Fault Tolerant Task Mapping in Many-Core Systems

Colin Andrew Bonney

PhD

University of York
Electronic Engineering

May 2016

Abstract

The advent of many-core systems, a network on chip containing hundreds or thousands of homogeneous processors cores, present new challenges in managing the cores effectively in response to processing demands, hardware faults and the need for heat management.

Continually diminishing feature size of devices increase the probability of fabrication defects and the variability of performance of individual transistors. In many-core systems this can result in the failure of individual processing cores, routing nodes or communication links, which require the use of fault tolerant mechanisms. Diminishing feature size also increases the power density of devices, giving rise to the concept of dark silicon where only a portion of the functionality available on a chip can be active at any one time.

Core fault tolerance and management of dark silicon can both be achieved by allocating a percentage of cores to be idle at any one time. Idle cores can be used as dark silicon to evenly distribute heat generated by processing cores and can also be used as spare cores to implement fault tolerance. Both of these can be achieved by the dynamic allocation of processes to tasks in response to changes to the status of hardware resources and the demands placed on the system, which in turn requires real time task mapping.

This research proposes the use of a continuous fault/recovery cycle to implement graceful degradation and amelioration to provide real-time fault tolerance. Objective measures for core fault tolerance, link fault tolerance, network power and excess traffic have been developed for use by a multi-objective evolutionary algorithm that uses knowledge of the processing demands and hardware status to identify optimal task mappings.

The fault/recovery cycle is shown to be effective in maintaining a high level of performance of a many-core array when presented with a series of hardware faults.

Contents

List of Tables

List of Figures

List of Equations

Acknowledgements

This thesis would not have been possible without the support of family, friends, supervisors and the Department of Electronic Engineering at The University of York.

First mention goes to Camilla Danese and Steve Smith who handled my application to study a PhD at the Department of Electronic Engineering of The University of York. Without the help of such dedicated and professional individuals the process of securing a PhD place would be a testing experience.

Camilla also works tirelessly on behalf of all the Electronis PhD students ensuring that essential record keeping is adhered to, often a thankless job when dealing with students who always think they have something better to do, be it research or recreational.

The University of York and the Department of Electronic Engineering that have provided the environment conducive to research. Both the department and university are brimming with individuals who are friendly and supportive. One could not wish for a better location, environment and social group within which to live and work.

Dr Gianluca Tempesti and Professor Jon Timmis who received my application enthusiastically. Without their support and enthusiasm this PhD will not have been possible. They both gave their time freely during my application and throughout my study, even when their personal timetables were busy beyond reason. I have been doubly favoured by fortune to have had two supervisors of the quality of Gianluca and Jon.

Particular thanks to Gianluca who has given me time in abundance to explore the concepts that form the core of this work. His knowledge of microprocessor hardware, processor arrays and FPGA's is inspirational.

Dr Yang (Jerry) Liu, a Research Associate on the Sabre project when I arrived at York, was always ready to enter into discussions on any subject whatsoever. His encyclopaedic knowledge of subjects within my research domain and of wider social issues meant that he was never lost for a word.

Dr Omer Qadir, unofficial social secretary, also with a huge knowledge base, who was ready to help when his busy life allowed.

Carolina, my wife, friend and companion, whose unceasing love and support has made this endeavour possible.

My mother and father who provided me with a loving home whose influence has enabled me to flourish.

My children Eloise, Rowan, Leona and Alison, who I have done my best to guide while watching them grow and blossom into adults. They have all nourished my quite pride of them with their own personal achievements.

My grandchildren Lina and Kaiden who are a constant source of joy and amazement.

Colin Bonney, May 2018

I dedicate this work to my family

Carolina
Eloise, Rowan, Leona, Alison
Lina, Kaiden
Cicley, John
Who fill my life with joy

All have achieved much in their lives

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Parts of this work have previously appeared in the publications listed below:

- C. Bonney, P. Campos, N. Dahir, and G. Tempesti, “Fault Tolerant Task Mapping on Many-core Arrays,” 2016 IEEE Symp. Ser. Comput. Intell. SSCI 2016, 2017.
- P. Campos, N. Dahir, C. A. Bonney, M. A. Trefzer, A. M. Tyrrell, and G. Tempesti, “XL-STaGe: A Cross-Layer Scalable Tool for Graph Generation, Evaluation and Implementation,” in IEEE Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2016, 2016.

Chapter 1

Introduction

In 1972 Intel released the 4004, a 4-bit microprocessor designed for the Japanese calculator manufacturer Busicom [1] which is credited as being the first commercially available microprocessor.

Early microprocessors had transistors numbered in the thousands, which restricted the functionality on a single device to a relatively simple microprocessor design (by today's standards) with additional functionality provided by external devices. With shrinking transistor sizes, the number of transistors on devices rose rapidly in line with Moore's Law [2]. The additional transistors were utilised to increase the complexity of the processor by increasing the word length, adding features such as floating point arithmetic and pipelining, adding memory cache and integrating functionality previously provided by external devices. However, there is a limit to the functionality and throughput that is possible with a single processor. In response to this, designs started to emerge in the 2000's with dual processors on a single chip followed in later years by quad-core and octa-core designs. These designs have become known as multi-core processors. Each core in a multi-core processor is complex with the design of the cores being closely related to the prevailing complex single-core processor designs.

In 2000 Hemani et al. [3] proposed the Network on Chip (NoC) architecture as a solution to a number of problems foreseen with the projected arrival, in the mid 2000's, of sub 100nm fabrication technology capable of producing one billion plus transistor devices [4]. Hemani's NoC design incorporates: processing cores, programmable logic, distributed memory and programmable I/O, all knitted together by a communication network enabling any element to communicate with any other element of the NoC. In 2007 Intel announced its intention to develop a many-core processor [5] proposing to integrate 100 medium 10M transistor processors, or even 1000 small 1M transistor processors, in a single device.

The difference between multi-core systems and many-core systems are not well defined but primarily relate to the number, complexity and level of independence of the cores. Multi-core systems quickly advanced from 2 to 32 or even 64 processors while many-core system will have cores numbering hundreds or thousands. The complexity of many-core system will typically be less than that of multi-core systems, sacrificing complexity of the processors in favour of greater numbers of processors.

In summary, the Many-Core paradigm offers the potential of increasing processing throughput compared to single and multi-core processors by spreading processing across the many, simpler cores, but requires the development of novel scalable solutions in hardware and software.

1.1 The Many-Core Paradigm

This section introduces the concept of the many-core system and related topics.

The basic architecture of a many-core array is generally represented as in Figure 1.1, with a lattice of interconnected routing nodes (the squares marked RN) that provide a communication fabric with each routing node having a directly connect processing core (the squares marked ’C’).

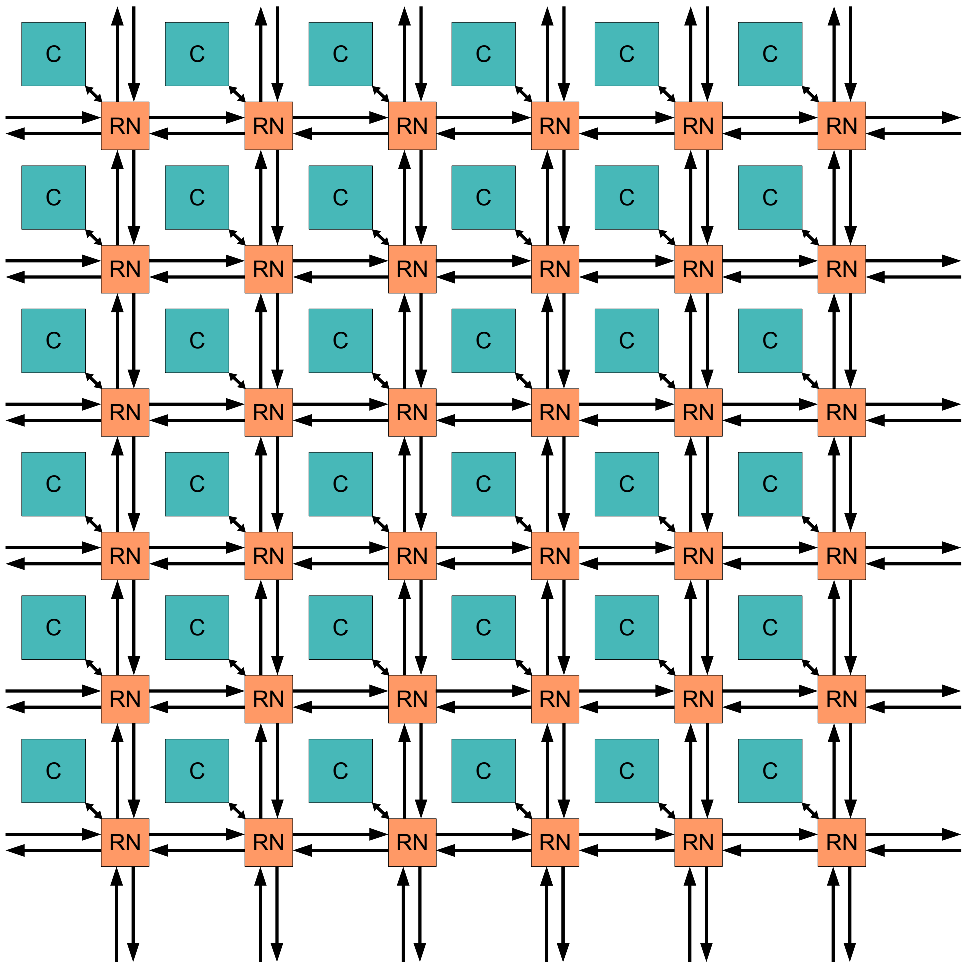


Figure 1.1 – **Many-Core Array**

While there are other possible arrangements of routing nodes and processing cores, for example the model proposed by Hemani et al.[3] uses a hexagonal arrangement of processing resources interleaved with a hexagonal communications matrix with each processing resource connected to multiple routing nodes, the model adopted for this thesis will use the square lattice arrangement of homogeneous processing cores, as shown in Figure 1.1.

Scale Independence

The many-core concept involves cores that number from hundreds to millions of cores which will require solutions for operation and control of many-core system to be *scale independent*.

Scale independence requires that there is no central control mechanism responsible for configuring or managing all the processors. A central control mechanism has the disadvantage of producing a single point whose failure would be catastrophic to the whole system and a single processor that would be required to undertake ever increasing amounts of work as the number of processors increased. The first of these problems is contrary to the concept of fault tolerance, while the second will place a limit on the maximum number of processing cores that can be utilised contrary to the concept of scale independence.

Scale independence therefore requires that monitoring and configuration are not centralised functions. The model that will be used in this thesis (see Subsection 2.4.5 *Core Regions*) is that a collection of processing cores which are physically close will be treated as a region, such that each region has independent monitoring and configuration and can communicate with the monitoring and configuration functions of neighbouring regions.

Dark Silicon

For many years the semiconductor industry has managed to keep pace with Moore's Law by doubling the number and density of transistors in devices every two years. However, reduction of feature size is no longer matched by a proportional reduction in transistor threshold and supply voltages [6] which is therefore increasing the power density of devices to the point where they can no longer be efficiently air-cooled. Post-Dennard Scaling means that because the leakage voltage cannot be ignored with device feature size of less than 65nm the voltage does not continue to scale down as the number of transistors increase. Consequently the power density increases in proportion to the increase in the number of transistors [7]. At a feature size of 22nm it is no longer possible to have the whole of the device active all the time requiring in the region of 20% of the chip to remain unused at any time [8]. To alleviate this, the concept of *dark silicon* where only a portion of the functionality available on a chip is active at any one time, has been proposed in order to keep the overall heat generation of the device to an acceptable limit. Many-core arrays can easily implement dark silicon by managing the number of cores that are active at any one time. The requirements of dark silicon coincides with the desire to provide fault-tolerance which also requires leaving a proportion of cores unused at any one time, as investigated in Chapter 4 *Core Fault Tolerance*.

Application Process Graph

An Application process graph (APG) is a graphical representation of an application broken down into processes represented as nodes and data transfers between processes represented by edges. In this research the graphs used to model applications are restricted to directed acyclic graphs (DAG) (see Subsection 2.4.2 *Application Process Graph*).

Processes are distinct parts of the application whose execution is dependent only on the data transfers modelled by the graph edges. This is a useful representation for mapping an application onto a collection of independent processors. An example application process graph with 8 nodes is shown in Figure 1.2. The edges of the graph in Figure 1.2 are annotated with numbers that represent the volume of traffic that is transmitted between the processes connected by the edge.

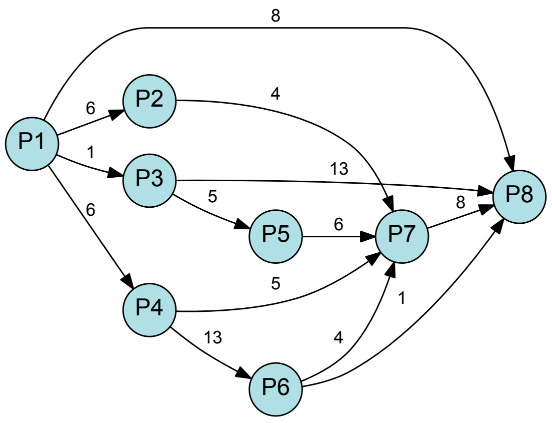


Figure 1.2 – **Example Application Process Graph with 8 nodes**

Task Mapping

Given an application process graph and a many-core array, *task mapping* allocates application processes as tasks to processing cores. The growth of computational time required to search all the possible solutions as the problem size increases places this problem in the NP-hard class of problems. Exhaustive searches are incapable of examining all the possible solutions in a reasonable length of time, so this research will use evolutionary algorithm techniques to search the problem space for solutions.

This research uses a multi-objective evolutionary algorithm develop with a set of objectives which will be used to guide the evolution of process mappings to optimize one or more objectives.

It is an axiom of this research that the mapping is not predetermined before operation of the system but evolves in response to the performance demanded of the system, the status of the system and the changing environmental conditions. To emphasise this point, there is no predefined initial mapping of any application process graph to the many-core system; the initial mapping will be determined by an initial placement algorithm (see Subsection 4.8.3 *Engineered Mappings*) based on the size of the application process graph and the size of the many-core array, after which alternative mappings will be evolved. This research will focus on determining in real time the suitable task mappings in response to the hardware status of the many-core system.

Fault Tolerance
The approach to fault tolerance taken in this thesis is to use the evolutionary objectives to search for process mappings that are inherently fault tolerant. A fault tolerance mapping is one which is either robust to faults so that the application can continue processing or can be ’repaired’ with minimal disruption to the processing of the application. In either case performance is likely to be compromised to some extent. This is described as *graceful degradation*.

The repair of fault will be followed by *graceful amelioration* which is search for new mappings that improve on the performance of the repaired mapping.

The whole process of fault detection followed by graceful degradation followed by graceful amelioration is described as a *fault-recovery cycle* which is discussed in detail in Section 7.2 *Fault/Recovery Cycle*.

Task Migration
Repair of a mapping after a fault and re-mapping during amelioration both require tasks to be migrated from one core to another. This research assumes that task migration mechanisms that can recover the state of a failed core, and mechanisms to recover lost packets in the case of a fault link, will be available.

1.2 Problem Description

This thesis sets out to:

"Demonstrate that a run-time fault-recovery cycle that implements graceful degradation and graceful amelioration along with a multi-objective search algorithm, is an effective strategy for maintaining task mappings that minimize the objective values and that the search for alternative mappings prolong the operational life of a many-core system compared to a static mapping that would quickly become non-viable after a small number of faults."

1.3 Related Work

There has been considerable research on design-time mapping [9–13] which in the vast majority of cases was applied to applications implemented using a collection of intellectual property (IP) modules for the design of NoCs consisting of heterogeneous processing elements (PEs).

Hu and Marculescu [9] analyse a generic video/audio MultiMedia System (MMS) application that includes video and mp3 encoders and decoders and partition it into 40 tasks which are assigned and scheduled onto 25 IP modules. A branch and bound algorithm is used to construct mappings of IP modules to tiles in an NoC architecture and minimize the total communication energy consumption calculated using an energy function that models the energy consumed by sending a single bit between two tiles of the NoC. The branch and bound algorithm is shown to be on average 72 times faster than a simulated annealing optimizer. The algorithm is used at design time to find mappings that are implemented and fixed pre-execution.

Lei and Kumar [10] use a two-step genetic algorithm to map an application represented by a task graph onto tiles on an NoC. The work concentrates on the static mapping of IPs at design time. The calculation of the optimization objective, the *average edge delay*, which is used to measure communication delay shares some commonality with the *network power* objective of Chapter 5 in this thesis, as both calculations make use of the distance between a pair of communication cores.

Murali and De Micheli [11] present a mapping algorithm they call PMAP for placing clusters onto processors, which produces mappings with lower communication cost than achieved by previous algorithms. They use bandwidth constraints as the optimization objective, which is also relevant to the *Excess Traffic* objective of Chapter 6. The PMAP algorithm uses Dijkstra's shortest path algorithm, applied to the quadrant graph (equivalent to a Com-Pair, see Section 5.1), to obtain the minimum path. The algorithm is used at design-time to find mappings that are implemented and fixed pre-execution.

Ascia et al. [12] investigate the mapping of 12 pre-designed, pre-verified modules in the form of intellectual property (IP) for an MPEG-2 encoder/decoder system, comprising DSPs, generic processors, embedded DRAMs and customized ASICs. A genetic algorithm is used to search for mappings that optimize performance and power consumption for a 5×5 array. The genetic algorithm is used at design-time to find mappings that are implemented and fixed pre-execution.

Derin et al. [14] use integer linear programming (ILP) to search for mappings that minimize the objectives of communication traffic and total execution time of an application in a mesh-based NoC with deterministic routing. The motivation is to combine the steps of allocating tasks to IPs to minimize execution time while at the same time minimizing communication traffic by allocating IPs to tiles. Derin et al. also implement a fault recovery mechanism for a single faulty core that would either reallocate tasks from a faulty core to IPs on other cores, or completely remap all tasks to optimize the objectives. These two approaches to recovery from a faulty core are analogous to the graceful degradation and graceful amelioration used in this thesis.

Sayuti and Indrusiak [13] use an approach that configure both task mapping and priority assignment, using a genetic algorithm to search for solutions. The optimization is performed at design time of a hard real-time embedded system based on a fixed priority pre-emptive NoC. As an alternative to searching for mappings with a GA, a constructive task mapping algorithm can construct task maps based on specific design properties [15].

Das and Kumar[16], Khalili and Zarandi [17], and Chatterjee et al. [18] all use a homogeneous NoC as the target platform and allocate multiple tasks to single tiles containing a

PE.

Das and Kumar[16] create a collection of task maps at design time, creating a mapping for each fault scenario which are saved so that they are available to at run-time. When a fault occurs, the map that whose scenario corresponds to the actual fault is used retrieved and used to re-map tasks.

The approach of Khalili and Zarandi [17] to fault tolerance is to allow the mapping of multiple applications, one at a time, and allocate a single spare core to each application during the mapping process. They use a function to calculate the criticality of a vertex in an *application core graph* which is influenced by the amount of traffic flowing into and out of the vertex, sorts the vertices by criticality, and then places the vertices to minimize the *Weighted Manhattan Distance*, which is the product of the distance between two communicating cores and the traffic volume between tasks located at the cores. A heuristic algorithm is used to first place the vertices onto cores and then allocate spare cores for fault recovery. The algorithm is used to map multiple applications onto cores of an NoC and allocates spares cores to specific applications during the mapping process. The algorithm has the effect of clustering the cores used for an application along with a single spare core while leaving the remainder of cores unallocated and available for additional applications. The application mapping is carried out at design time, before the application starts executing and remains fixed thereafter, while the migration of tasks when a PE becomes faulty is a run-time process.

Chatterjee et al. [18] developed an algorithm that provides a unified mapping and scheduling method for real-time systems, focusing on meeting application deadlines and minimizing communication energy while mitigating the effect of failure-prone processing elements. The platform is a NoC consisting of a lattice of homogeneous processing elements, each of which can be allocated multiple tasks by the mapping algorithm. The model determines the communication energy and communication time based on the Manhattan distance between the source and destination nodes. Fault tolerance is achieved by replicating tasks that have been allocated to PEs that are judged to be unreliable. The duplicate tasks either run in parallel with the original or stand by to start execution if the original PE should fail.

1.4 Novel Contributions of this Thesis

This thesis uses a lattice of homogeneous processing cores to a model a many-core array. In contrast to previous work, it is assumed that there will be sufficiently many cores that a single task can be allocated to a single core, obviating the need for scheduling multiple tasks to a processing core and an operating system to manage multitasking between multiple tasks. Also, in contrast to previous work, the philosophy of the use of spares cores is that 20%-25% of cores should be idle at any one time to fulfil the dual purpose of providing spare cores for fault tolerance and maintaining a percentage of the device inactive as dark silicon.

This required the development of a new model or the many-core system. The model used in this thesis includes a *hardware map* that is a representation of the processing resources and communication network of the many-core array. The hardware map records the fault status of every processing core and communication link, and also includes information regarding sources and sinks to resources external to the many-core array. The sources and sinks to external resources are regarded as the *environment* within which any particular process mapping exists. The environment affects the calculation of the metrics underlying the optimization objectives. At the time of writing, the aspect of the model that explicitly models links to external resources, was unique to this work and is regarded as a vital step from a theoretical model to a practical implementation.

The approach to fault tolerance is also fundamentally different from previous work. Fault tolerance requirements are represented as objectives which are an integral part of the search for task mappings so that the resulting task mappings are inherently fault tolerant, which is to say that the mapping is either robust when confronted with a link fault or, when a core fault occurs, can effect a repair by migrating the task to a near-by idle core. Core fault tolerance is achieved by distributing idle cores amongst the processing cores so that every processing core is in close proximity to an idle core. Link fault tolerance is achieved by minimizing the number of communicating core pairs that are dependent upon a single hardware link. No assessment of the reliability of individual cores is attempted; every core is assumed to be equally reliable until information to the contrary is collected from the functioning many-core array.

In summary, the novel contributions of this thesis are:

- The development of a *hardware map* that models interfaces to external resources and the fault status of cores and links in sufficient detail to enable the calculation of the metrics and objectives, while remaining computationally tractable.
- The formalization of the following objectives and underlying metrics for use by a multi-objective evolutionary algorithm to search for mappings:
  - Core Fault Tolerance
  - Link Fault Tolerance
  - Network Power
  - Excess Traffic
- The formalization of the concept of a pair of communicating cores, a ComPair, that underpins the calculation of metrics and objectives of the process maps.
- The developed of the concept of link criticality, used alongside the ComPair to determine how vulnerable a mapping is to the failure of a link.
- The presentation a mathematical proof for an algorithm that calculates the number of fault free paths in a network with an arbitrary number of faulty links.

- The implementation of a fault/recovery cycle that implements graceful degradation to effect a repair of a mapping to recover from core faults and, following a core or link fault, performs graceful amelioration using the evolutionary algorithm to search for alternative mappings that improve fault tolerance and/or performance of the repaired mapping.
- The implementation of a multi-objective evolutionary algorithm to search the solution space of any number of objectives to produce a Pareto Front of mappings used for graceful amelioration.

1.5 Research Outcomes

An evolutionary algorithm using the objectives for core fault tolerance and link fault tolerance was successful in finding mappings that were resilient to fault events within a defined computational budget. The results also demonstrate that the fault tolerance objectives can work successfully with performance objectives to produce Pareto Fronts that include mappings that range from weak fault tolerance in combination with good performance to strong fault tolerance with poorer performance (Subsection 6.8.6, Figure 6.15 and Figure 6.21).

Having developed a suitable evolutionary algorithm, a Monitor process was developed to implement graceful degradation and amelioration through a fault/recovery life cycle, consisting of the following steps:

- Normal operation
- Fault detection
- Graceful degradation
- Graceful Amelioration
- Return to normal operation

The fault-recovery cycle is designed to be used in real-time so that the many-core system can withstand a series of fault each being mitigated by repair followed by remapping, until the hardware resources are reduced to a level that can no longer sustain the processing required by the application.

The experiment results presented in Chapter 7 show that the cycle is effective in returning the post-fault system to a level of operation with a performance that is close to the pre-fault performance and prolong the operational life of a many-core system compared to a static mapping that would quickly become non-viable after a small number of faults. The fault-recovery cycle continues to be effective until there are too few processing cores to accommodate the application process graph processes, or too few remaining links in the communications array to allow communication between all ComPairs.

The results of the experiments confirm that the use of graceful degradation and amelioration, along with a Monitor to manage the recovery cycle, can maintain system performance, through a sequence of faults, at a level that would otherwise not be possible.

1.6 Thesis Structure

Chapter 2 *Many-Core Systems* reviews the development of microprocessors and network on chips leading up to the proposal of the many-core paradigm.

Chapter 3 *Fault Tolerance* reviews existing approaches to fault tolerance.

Chapter 4 *Core Fault Tolerance* presents the initial many-core model and single objective evolutionary algorithm to search for task maps that optimize the *Core Fault Tolerance* objective.

Chapter 5 *Network Power* introduces a second objective of *Network Power* along with a revised multi-objective evolutionary algorithm. The concepts of *ComPair* and *Link Criticality* are introduced.

Chapter 6 *Link Fault Tolerance and Network Traffic* adds two further objectives of *Link Fault Tolerance* and *Excess Traffic*. The many-core model is expanded further to to include the *Hardware Map* used to model faulty cores and links, and the *Environment* to model external data sources and sinks.

Chapter 7 *Graceful Degradation and Amelioration* implements a *Fault/Recovery Cycle* that uses *Graceful Degradation* and *Graceful Amelioration*.

Chapter 8 *Conclusions and Future Work* discusses the results of this thesis and suggests areas worthy of further research.

Chapter *The Many-Core Model* is a compilation of the complete many-core model developed in Chapters 4, 5 and 6.

Chapter *Metrics and Objectives* is a compilation of the four metrics developed in Chapters 4, 5 and 6.

Chapter 2

Many-Core Systems

2.1 Computer Architecture

This section reviews a range of computer architectures to provide a context for the understanding of the origin and motivations for the many-core model.

2.1.1 Evolution of Microprocessor Based Systems

Throughout the development of integrated circuits, feature size has been shrinking, allowing the number transistors on a chip to increase. In 1965 Gordon Moore, while employed at Fairchild Semiconductor, wrote a paper in which he identified the historical increase in complexity of components as doubling every year and stating that there was no reason why this trend should not continue [2]. In 1975 Moore revised his prediction for the doubling of transistor counts to take place every two years [19]. Intel processor designs kept up with Moore's Law over the 45 years from 1970 to 2015, during which time the transistor count increased from 2.5×10^3 to 5×10^9, an increase by a factor of 2×10^6.

The vast number of additional transistors have allowed designers to evolve the design of the single chip microprocessor in a number of directions. This examination focuses on four areas of design:

- Word size
- Pipeline stages
- Multi-core architectures
- Cache

Evolution of microprocessor design has tended to focus on one aspect of design at a time. The word length evolved to a size of 32 bits before pipelining evolved. When the number of pipeline stages reached was judged to be optimal, the emphasis moved to the number of cores. Caches are, the exception, first making a debut as small L1 caches at the same time as pipelines and then developed alongside pipelines and then multi-core processors which were accompanied by larger caches.

This staged evolution of one design element at a time is consistent with designers focusing on the design element that offered the best performance gain for the available transistors, optimizing one design element before moving on to the next. The exception of the development of caches can be explained by caches supporting and enhancing the performance of pipelines and multi-core processors.

Word Length The 8 bit 8080 processor uses an instruction set whose length is 1, 2 or 3 bytes, while the 16 bit 8086 uses an instruction set with instruction lengths varying from 1 to 6 bytes depending on the addressing mode. In both cases the majority of instructions require the fetching and decoding of multiple words, with the fetching and decoding of each word occupying a clock cycle. The multiple length instructions therefore reduce the effective speed of the processor when compared to instructions that occupy a single word of the machine's architecture and are fetched in a single cycle.

With the introduction of the i386 processor with a word length of 32 bits, the opcode, addressing mode information and register addresses could be contained within one word with additional words required only for addresses and immediate data, thus reducing the average cycles per instruction when compared to a 16 bit processor.

The benefits of a 32 bit word length compared to a 16 bit word length are considerable, especially when it is recognised that the longer word length also benefits the access of data as well as instructions, thus making this an obvious priority in the design evolution. Word lengths greater than 64 bits offer the option of encoding more than one operation in a single word, increasing the number of operations fetched per cycle to more than one.

Pipelining Having evolved the word length to 32 bits, the emphasis switched to the development of instruction pipelines. The motivation for pipelining is to increase the number of instructions that are executed per second by increasing the clock frequency.

Pipelining arises from the realisation that each instruction passes through a number of serialised stages and that it is not necessary for one instruction to have completed all stages before the next instruction enters the first stage. For example the Intel i486 breaks the instruction execution down into 5 stages [20]:

- FI: Fetch the instruction from cache
- D1: Main instruction decode
- D2: Secondary instruction decode, and memory address computation
- EX: First execution clock
- WB: Write results into the register file

Each stage takes place in a single clock cycle. Without pipelining an instruction that requires all five stages to be executed requires a long clock cycle to complete. With pipelining the i486 can execute five instructions simultaneously reducing the average execution time for some instructions to less than one clock cycle. Pipelining achieves this because each stage is relatively simple and so uses fewer logic gates, reducing the amount of time required for the logic to complete its task and therefore allowing the clock frequency to be increased [21].

The benefit of the i486 five stage pipeline can be illustrated by comparing the number of clock cycles required to complete execution of a selection of instructions to the equivalent i386 instructions [20]:

Instruction	Clock Cycles	
Type	i386	i486
LOAD	4	1
STORE	2	1
ALU	2	1
JUMP taken/not	9/3	3/1
CALL	9	3

In the Pentium 4 design the number of pipeline stages increased to 20 which, Intel reported, increased the relative frequency of the Pentium 4 when compared of the to the i486 by a factor of 2.5 [21].

A balance needs to be achieved between lowering the number of logic gates in a pipeline stage to the increase the frequency and increasing the number of pipeline stages which increases the total number of clock cycles required to complete the execution of an instruction. The optimum, based on Intel processor designs, appears to be in the region of 20 stages.

A further evolution of pipeline technology is the *superscalar* architecture, which is a design that has multiple pipelines working in parallel to increase the number of instructions executing simultaneously.

Processing Cores The continuing growth of the number of transistors available to designs next found a use is multi-core designs. Multi-core designs provide two or more identical and sophisticated processing cores on a single die. The sophistication and power of the cores is similar to the that of single processor designs. The processing cores are independent while sharing resources such as memory and communication buses.

- 2005 Intel released their first dual core processor, the Intel Pentium Processor Extreme Edition 840
- 2007 Intel released a Core 2 Quad Processor
- 2014 Intel unveiled its first 8 core desktop processor, the Intel Core i7-5960X processor Extreme Edition

- 2016 Intel released the 64 core Intel Xeon Phi Processor 7210
- 2016 Intel released the 72 core Intel Xeon Phi Processor 7290

On Chip Cache Memory is a relatively transistor hungry resource, using 50M transistors per megabyte of memory based on a design that uses 6 transistors per memory cell making it inevitable that the addition of cache memory to a microprocessor chip was lower down the list of processor features to take advantage of the increase in transistor numbers. Level 1 (L1) cache first appeared in the i486 at the same time as pipelining was introduced into the the processor design. Although the 8KB cache of the i486 seems small, especially when compared to the later multi-megabyte caches, it accounts for roughly 4% of the transistor count of the i486.

In single core designs L1 cache was on-chip and generally stayed below 1MB while L2 cache was generally, but not exclusively, off-chip. With the advent of dual core processors, most designs allocated a quantity of L1 cache for the exclusive use of each core and added on-chip L2 cache which is shared between multiple processors. As a result there was a step change in the size of on-chip caches from kilobytes to megabytes that coincided with the introduction of multi-core designs.

2.1.2 Network On Chip (NoC)

In 2000 Hemani et al. [3] proposed the Network on Chip (NoC) architecture as a solution to a number of problems foreseen with the projected arrival, in the mid 2000's, of sub 100nm fabrication technology capable of producing one billion plus transistor devices [4]. As transistor density has increased additional functionality has been added to single chips, resulting in system on chip (Soc), multi-core processors and NoC designs. However there is a limit to the level of functionality that a single processor can have [5]. A billion transistors cannot be effectively used by a single processor.

The development of multi-core processors and NoCs are an exercise in increasing the number of processors on a chip from one to many. Increasing the number of processors on a chip creates two problems that need to be solved to enable the processors to be used effectively: how to make effective use of multiple processors to solve a problem (parallel computing and process mapping) and how to efficiently route data between processors (routing algorithms).

2.1.3 System on Chip (SoC)

During the 1960's a number of companies were driving development efforts to produce digital watches [22]. Early watches were composed of many power hungry components and were unreliable due to the large number of electrical connections. In 1974 Intel hired Peter Stoll [23] to develop the 5810, an IC that contained all the electronics required for an electronic watch including analogue components, timing circuitry and LCD display driver logic. The 5810 is considered to be the first example of a system on chip (SoC). The SoC model is used in embedded devices which use a collection of heterogeneous systems. The SoC design approach is applied to systems with sub-systems that require the speed advantage of specialized hardware compared to the flexibility of a general purpose processor executing software code.

The development of SoCs have continued in parallel with the development of general purpose processors finding application in mobile phone, digital signal processing and wireless base stations.[24]

An SoC is a number of subsystems integrated on a single chip that would previously have been fabricated separately and then brought together on a circuit board. NoCs help improve physical connection and communication speeds between components that would previous have been separate devices connected via wire tracks on a circuit board. SoCs can suffer from bus overload and the inability to transmit the clock signal across the chip within a single clock cycle.

Guerrier and Greiner [25] propose an on-chip switched network to replace general purpose buses for inter-processor communications in for SoC designs.

2.1.4 A Processor Centric Perspective

It is useful to examine processing models based on the hardware configuration since it is the hardware design that limits or promotes possible processing models.

Single Processor This is the classic microprocessor design in which a single processor executes a sequence of instructions on a data set. Single processor designs work well for general purpose computing where many different tasks are being performed and the majority of functions do not involve performing the same processing on many data sets. If the same computation is required to be performed on many datasets then the processor will execute the same instructions many times over, once for each individual data set. The performance of single processor designs has been improving with the introduction of longer word lengths, FPUs, instruction lookahead, pipelining, memory caching and increasing clock frequencies as discussed in Subsection 2.1.1.

In the early years of microprocessor development the increasing transistor count, made available by shrinking feature size, was used to add design improvements to increase the speed of single processors. However there is a point where the increasing numbers of transistors cannot be used effectively to further improve the performance of single processor designs. The introduction of the multi-core design enables designers to make effective use of the available transistors.

Multi-Core Multi-core designs provide two or more identical and sophisticated processing cores on a single die. The sophistication and power of the cores is similar to that of single processor designs. The processing cores are independent while sharing resources such as memory and communication buses. Where cores have independent memory caches they must be kept consistent by implementing cache coherency.

The features added to single core processors have a direct effect on the speed of execution of all programs without the need for intervention by the operating system or any special coding by the application programmer. This is not the case for additional cores. Additional cores give the potential for increasing overall execution speeds which can be utilised by the operating system sharing work between cores or application programmers making specific use of additional cores by adding parallelization constructs to their code.

Multi-core designs are limited to a relatively small number of cores, typically 2-8 cores.

Tightly Coupled Processor Arrays Tightly coupled processor arrays are designs that incorporate many processor cores, typically more than 16, that are designed such that each processor can run the same instructions at the same time on different data sets. These processor arrays may also be referred to as coprocessors as they are under the control of a single controlling processor. They are capable of very high processing rates for problems that can be highly parallelized. Programs that take advantage of the large number of cores, need to be written with parallel specific code using a platform such as CUDA, OpenCL and OpenMp.

This class of processor arrays includes graphical processing units (GPU's) such as those produced by Nvidia and ATI Technologies and the Intel Xeon Phi coprocessors.

Many-Core Array A many-core array is an array of independent processing cores, on a single die, each of which can execute different program code to any of the other cores and processing data unconnected to the data processed by the other cores. A many-core processor will typically have cores numbering hundreds or thousands and the cores' complexity will be significantly reduced compared to cores in multi-core processors.

The Intel Xeon Phi coprocessors straddle the multi-core and many-core processor models. As a co-processor with in the region of 60 cores the Xeon Phi is not a multi-core processor. The design is closer to a processor array although it is possible to use the cores independently and therefore can mimick as a many-core processor.

von Neumann Architecture The von Neumann architecture is the design underlying almost all computers. It is a design consisting of a processor and memory; address bus, data bus and control bus; input and output devices. Despite its simplicity it is capable of processing any computable function. A von Neumann machine, in its original configuration, is a sequential processor that retrieves and processes a single instruction at a time. The von Neumann architecture suffers from the twin bottlenecks of execution of one instruction at a time and a single, limited capacity, data bus between the processing unit and the main memory that is used for both programs and data [26, 27].

Dataflow Machines have been researched since the 1970s and are architectures with parallel processing at the heart of their design [28, 29]. The fundamental philosophy of dataflow machines is that many instructions that are specified and executed sequentially in a von Neumann machine are not dependent on each other so can be executed in parallel. In a data flow machine instructions are not executed in sequence, but are executed when their required input values become available, allowing many instructions to be executed simultaneously by a number of functional units [28, 30].

Researchers have proposed various designs for dataflow machines [28, 31–34]. Initial attempts at dataflow machines were fine-grained, working at the instruction level parallelization. The the late 1980s and early 1990s the fine grained dataflow machines had largely been considered to be failures, not because the dataflow concept was considered flawed, but because the overhead of required to process individual operations in a fine-grained architecture. A many-core array can be used as a course-grained dataflow machine.

Sequential processing machines and fine-grained dataflow machines can be regarded as the extreme ends of a continuum both of which have inherent performance limitations. The success of the dataflow model will be determined by finding a suitable granularity of process such that the overhead of initiating the processes is insignificant to the gains of parallel processing.

Flynn's Taxonomy In 1972 Flynn defined four processing models based on the combination of types of instruction stream and types of data stream the the processor exploits resulting in the following four categories as described in [35]:

- SISD (Single Instruction stream, Single Data stream)
- SIMD (Single Instruction stream, Multiple Data streams)
- MISD (Multiple Instruction streams, Single Data stream)
- MIMD (Multiple Instruction streams, Multiple Data streams)

Flynn's taxonomy was developed to describe the execution of a single application so does not adequately describe many of today's systems. However, it is of historical interest and still provides a useful starting point.

In general terms Flynn's description of these models can be summarised as:

SISD (Single Instruction stream, Single Data stream) This model describes single core general purpose computers such as the microprocessor before it evolved into a multi-core design. There is a single processor which receives a single stream of instructions which operates on a single data set. There is no parallelism.

SIMD (Single Instruction stream, Multiple Data streams) When describing this model Flynn specifically mentions the SOLOMON Computer [36] which he describes as an array processor. The SOLOMON computer is an early example of a design which incorporates multiple processing units where the default configuration is for each processor to process the same instruction simultaneously on different data sets. In concept the SOLOMON Computer is similar to the GPUs (Graphics Processing Units) employed as co-processors for image processing that become available in the late 1990s and early 2000s from Nvidia and ATI Technologies.

MISD (Multiple Instruction streams, Single Data stream) This model is described by Flynn as:

> "[MISD] include specialized streaming organizations using multiple-instruction streams on a single sequence of data and the derivatives thereof."

A possible interpretation from Flynn's description is to regard MISD as an extension of SISD where the results of an action upon a data element remains in the processor to be acted upon my further processor instructions. Another interpretation is a machine that has multiple processors, the first of which processes a data element the results of which are processed by 2nd processor etc, thereby each data element is processed in some way by a sequence of processors, which is one of the ways in which a many-core system could be used.

MIMD (Multiple Instruction streams, Multiple Data streams) The notable property of MIMD systems is that two or more independent processors share some resource, usually memory, to increase processing throughput. In contrast to the SIMD model, the MIMD model introduces significant challenges for the coordination of the independent processors and the safe sharing of shared. The modern concept of multi-core and many-core system fits within this model. Limitations of memory data access may lead to cores remaining idle while waiting for data accesses to be completed. [37]

2.1.5 Amdahl's Law

While at IBM in 1967 Amdahl was asked by IBM to make a presentation with the purpose of comparing the computing power of a superscalar uni-processor computer to that of a quasi-parallel ILLIAC IV computer with 16 processors. [38, 39]. Amdahl made the observation that the speedup of a computed function is limited by the part of the function whose execution remains sequential. The consequence of this was to become known as Amdahl's Law and was stated by Getov [40] as:

> "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

Empirically, Amdahl, this reasoning was expressed as Amdahl's Equation which gives the processing rate of a parallel machine as a function of the processing rate of a sequential machine and the number of processors available in the parallel machine:

$$R = R_s + \frac{S + P}{S + \dfrac{P}{M}} \tag{2.1}$$

Where:

R	=	Processing rate of parallel machine
R_s	=	Processing rate of sequential machine
S	=	Proportion of workload that is processed sequentially
P	=	Proportion of workload that can be parallelized
M	=	Number of processors in the parallel machine

Rearranging as:

$$\frac{R}{R_s} = \frac{S+P}{S+\dfrac{P}{M}}$$

Where R/R_s is the relative speed of the parallel machine compared to the sequential machine and is often referred to as the speedup. If the sequential machine is given the processing rate of 1, then the equation simplifies to:

$$R = \frac{S+P}{S+\dfrac{P}{M}}$$

Where R is a value which represents the number of times faster the parallel machine processes the workload than the sequential machine.

Since Amdahl defines S and P as being the proportion of the total workload that, respectively, remain sequential and and can be parallelized, we have:

$$S+P=1$$

Giving the final equation of:

$$R = \frac{1}{S+\dfrac{P}{M}} \tag{2.2}$$

Amdahl gave an example of the proportion of a problem that remains sequential as 35% of the original problem, which is equivalent of $S = .35$ and $P = .65$. Amdahl originally developed this equation to model the speedup obtained from the ILLIAC IV with 16 processors and (generously) assumed all 16 processors could be utilised to maximum effect on the proportion of the workload that can be parallelized. Using these values the speedup of the 16 processor machine is given by:

$$R = \frac{1}{.35+\dfrac{.65}{16}}$$

$$R = 2.56$$

This gives a speedup figure of 2.56 for a parallel machine with 16 processors. Amdahl's argument was that this speedup is modest compared to the number of processors and therefore resources should be expended not only in parallelization bit also on speeding up sequential processing.

Since the original paper Amdahl's Law has been the subject of many research papers either by attempting to challenge the fundamental limit set by Amdahl [41], or extending it into new areas [7, 42–46] to mention only a few.

Gustafson [41] argues that Amdahl's estimate for the sequential element of the total workload of 35% is high and reports speedup of 1016-1021 using a 1024 processor machine. Gustafson's research uses the three problems of: beam stress analysis using conjugate gradients, baffled surface wave simulation using explicit finite differences and unstable fluid flow using flux-corrected transport. In these cases the number of computations that are parallelized is high so P is large compared to S which, from the reported results, we can infer is less than 0.1% of the total workload. According to Gustafson an important property of these problems is that the sequential element does not increase with an increase in total workload therefore the more parallelizable computations that need to be carried out the greater the benefit and the higher the speedup.

2.2 Hardware Considerations Related to Many-Core Systems

This section reviews some of the issues with single core processors that led to the conception of the many-core systems and some of the technical and design issues that need to be solved to implement many-core system.

2.2.1 Clock Propagation

Propagation of clock signals becomes problematic as die sizes and clock speeds increase. The speed of propagation of an electromagnetic signal in a medium is given by the equation:

$$v = \frac{c}{\sqrt{\varepsilon}} \tag{2.3}$$

Where:

v	=	signal velocity
c	=	speed of light (3 x 10^8 m/s)
ε	=	dielectric permittivity

Sylvester and Keutzer [47], [48] use a rearrangement of equation 2.3 to derive an equation for the Time of Flight (TOF); the time taken for a signal to travel a specified distance. They give:

$$TOF \quad = \quad 33.33\sqrt{\varepsilon} \quad ps/cm \tag{2.4}$$

Using equation 2.4 together with projections given in [4] of a die size of 750 mm^2 equivalent to a die edge of 27.4 mm, a feature size of 50nm a dielectric permittivity $\varepsilon = 1.5$, and restricting the longest path length, equal to 2 sides of the die, to 80% of the clock cycle, Sylvester and Keutzer give a maximum clock speed of 3.58 GHz.

The consequence of increasing die sizes and clock rates is that it is not possible for the whole chip to remain isochronous, leading to clock skew, as one clock cycle cannot propagate through the whole device before the start of the next cycle.

2.2.2 Globally Asynchronous Local Synchronous (GALS) Architecture

The concept of GALS made a debut in 1984 in a thesis by Chapiro [49]. A GALS architecture can be employed to overcome the problems of non-isochronous devices. In a GALS architecture, the device is designed as a number of subunits, each with a size of up to 3 x 2^5 transistors. Each subunit has a clock providing local synchronisation for the subunit, while communication between subunits is asynchronous.

Hemani et al. [50] (1999) claim that power consumption due to clocks, which are a major source of power consumption, can be decreased by 70% using a GALS architecture. In contrast the 2002 paper of Iyer and Marculescu [51], using simulations, claim that there is little power saving from clocks in a GALS architecture, although they identify power savings of 10% in a 5 domain GALS SoC while performance is reduced by 10-15%.

Muttersbach et al. [52] developed GALS architectures where a locally synchronous module is surrounded by an asynchronous wrapper using a pausable clock. The pausable clock was introduced by Yun and Donohue [53], while the asynchronous wrapper was proposed by Bormann and Cheung [54]. Muttersbach et al. present their work as a practical implementation of GALS that can be implemented in silicon along with a library of wrapper components to remove the burden of designing synchronous-asynchronous interfaces. Muttersbach et al. demonstrate the validity of their design by using it in the implementation of the Safer cryptoalgorithm in an ASIC.

Moore et al. [55] also present a mechanism for implementing communication between locally synchronous domains which rely on asynchronous logic techniques. The approach of Moore et al. is similar to that of Muttersbach et al. as they also use asynchronous wrappers and clock pausing. In addition both papers use a handshaking protocol as part of the communication process.

Dobkin et al. [56] examines two principle mechanisms used to achieve communication between modules in different clock domains. They look at mechanisms involving synchronisation of clocks using pausable delays and those using locally generated arbitrated clocks. Their analysis shows that there is a risk of synchronisation failure if clock delays are not taken into account. They propose a number of circuits designed to prevent metastability during data synchronisation.

Royal and Cheung [57] investigates the implementation of GALS designs on FPGAs. The work of Royal and Cheung carried out in 2003 was at that time incomplete. This is an interesting area of research as the ability to model GALS designs on FPGA's will be a critical step when investigating the effectiveness of novel designs.

Many-core systems are likely to use the GALS allowing each core's processing rate to be determined by the core's local clock while the node routing for synchronising communication with its neighbours.

2.2.3 Dark Silicon

For many years the semiconductor industry has managed to keep pace with Moore's Law by doubling the number and density of transistors in devices every two years. However, reduction of feature size is no longer matched by a proportional reduction in transistor threshold and supply voltages [6] which is therefore increasing the power density of devices to the point where they can no longer be efficiently air-cooled. Post-Dennard Scaling means that because the leakage voltage cannot be ignored with device feature size of less than 65nm the voltage does not continue to scale down as the number of transistors increase. Consequently the power density increases in proportion to the increase in the number of transistors [7]. At a feature size of 22nm it is no longer possible to have the whole of the device active all the time requiring in the region of 20% of the chip to remain unused at any time [8]. To alleviate this, the concept of *dark silicon* where only a portion of the functionality available on a chip is active at any one time, has been proposed in order to keep the overall heat generation of the device to an acceptable limit. Many-core arrays can easily implement dark silicon by managing the number of cores that are active at any one time. The requirements of dark silicon coincides with the desire to provide fault-tolerance which also requires leaving a proportion of cores unused at any one time, as investigated in Chapter 4.

2.3 On-Chip Communication Networks

The communication system is the collection of hardware and software that provides the functionality required to transmit data between cores, consisting of the physical communication channels, the routing nodes that manage the traffic, the network switching type and the routing algorithm.

2.3.1 Network Switching

This section will describe the network switching methods while section Subsection 2.3.3 will describe some routing algorithms that have been developed.

Switching describes the manner in which the network resources are acquired for the purposes of transmitting a message. Switching does not address the problem of how to choose a route. This section describes the primary models for switching relevant to NoC networks.

Circuit Switching Circuit switching allocates the resources for the entire end-to-end route for as long as it takes to complete transmission of the whole message. The quintessential and most familiar example of a switched network is the traditional land-based telephone network where the dialling of a number allocates the telephone line exclusively for the caller for the duration of the call.

Packet Switching Packet switching divides the message into packets which are sent over the network individually and then reassembled at the receiving location. Each packet carries the required target address information allowing each packet to be sent separately without reference to other packets. Each packet from a message may take a different route from the source to the destination. Each node on the route receives and stores a whole packet before determining the next leg of the route and sending it onwards requiring buffers that can store at least one packet [58].

Store and Forward Store and forward is where a whole message is stored at each node along the path to its destination. At each node the whole message is received and stored until the resources are available to transmit the message to the next node. It differs from packet switching in that the whole message is sent and stored as a unit [58]. Examples of store and forward networks are telegraphs and letters sent by post [59]

Virtual Cut-Through Virtual cut-through is a modified version of packet switching. When a packet is received at a node, the node does not wait for the whole packet to arrive before examining the target address and determining the next step on the route Kermani and Kleinrock [58]. If a route is available then the node will immediately retransmit the packet to the next node without storing the packet. If no route is available the packet is stored until a suitable route becomes available. Virtual cut-through can reduce latency compared to packet switching. Kermani and Kleinrock found virtual cut-through outperformed packet switching using the performance criteria of network delay, traffic gain and buffer storage requirements. Virtual cut-through requires sufficient buffer space at each node to store each packet in the event that the next channel is busy.

Wormhole Switching Wormhole switching, described by Dally and Seitz [60][61], often incorrectly referred to as wormhole routing, is similar to circuit switching in that the whole end-to-end route is allocated for the transmission of the whole message. In contrast to circuit switching, wormhole switching allocates the route as the message is being sent. Each message is divided into packets and each packet into flits (flow control bits). Only the header flit carries the target address. Each node that accepts a header flit is exclusively allocated for the transmission of the message until the whole message has been transmitted. As soon as a node successfully transmits a flit, it is sent the next flit. Routing nodes require buffers to store a small number of flits rather than whole packets, substantially reducing the buffer size requirements and latency over store and forwards methods. The flits of a message will become spread out along the communication channel, resembling a worm working it way through the network, hence the term wormhole.

Although the term wormhole was not used in Dally and Seitz's 1986 [60] paper, this is the first printed description of wormhole routing which was developed jointly by Dally and Seitz when Seitz was Dally's PhD advisor. Dally also explains that *"Wormhole routing is actually a misnomer - it refers to the flow-control used, not the routing."* (W. J. Dally, personal communication, September 18th, 2014).

2.3.2 Network Latency

It is important to reduce the latency of messages passing through the network to a minimum. Latency will depend on a number of properties of the network and the message including:

- size of message or packets
- time taken to allocate communication resources
- distance between source and destination nodes
- delay at routing nodes

Each of the switching models described above have their own latency profile.

Dally [62] compares the latency of *store and forward* and *wormhole* switching methods, giving the latencies as:

$$T_{SF} = \left(D + \frac{L}{W}\right) \tag{2.5}$$

$$T_{WH} = \left(D \times \frac{L}{W}\right) \tag{2.6}$$

Where:

T_{SF}	=	latency for store and forward switching
T_{WH}	=	latency for wormhole switching
D	=	distance
L	=	message length
W	=	data width

Felperin et al. [63] give a more detailed analysis of the latency in a wormhole switched network.

2.3.3 Routing

Routing is the mechanism by which a route through a network, for the transmission of a message or a message packet, is chosen. In principle any routing algorithm can be used with any of the switching methods describe above. Receiving, buffering and transmission of data is the responsibility of hardware. Routing decisions may be made by hardware, software or combination of both. All routing algorithms will interact with the routing hardware. The features provided by the routing hardware will determine how easy it is to implement an algorithm. A number of the routing algorithms are accompanied with a design for routing hardware.

A large number of routing algorithms have been developed for use in NoC some of which are reviewed in this section.

Routing Deadlock Routing deadlock is described by Kleinrock [64] as the inability of a communications network to transmit data between nodes. This can occur when two routing nodes have data to transmit to each other and the data buffers of each routing node are full with the data waiting to transmit. Neither node can transmit its data to the other until the data buffer of the target node is no longer full. Thus, neither node can transmit or receive data until some exception action is taken to free up one or other of the buffers. Routing protocols need to be designed to avoid deadlock situations.

xy Routing In [60][65] Dally and Seitz describes xy routing of packets used in the Torus Routing Chip. Although described as xy routing Dally and Seitz's presents a multidimensional model of arbitrary dimensions with each packet containing the n dimensional address of the destination. xy routing become popular and can, perhaps, be considered as the benchmark against which all other routing strategies are measured.

xy Deadlock Free Routing Dally and Seitz [61] describes a deadlock free xy routing algorithm based on virtual channels. Dally and Seitz defines a *"routing function"* and a *"channel dependence graph"* and gives a proof that if a channel dependence graph has no cycles then the associated routing function is deadlock free. Virtual channels are used to construct a cycle free channel dependence graph. Each virtual channel is associated with its own queue ensuring blocked messages do not hinder the progress of other messages [61][66].

Multi-Cast Routing Lin and Ni [67] develop a multicast wormhole routing strategy to provide deadlock free routing of messages to multiple nodes. The method demands that networks have a *Hamilton path* which is a path through the network where every node is visited once and only once. Lin and Ni state that almost all topologies being studied, including 2D-mesh and hypercubes, have Hamilton paths. A given network may have more than one Hamilton path. From the point of view of the performance of this routing method, not all Hamilton paths are equal, some being more efficient than others.

Turn Model Glass and Ni [68][69][70] present the Turn Model, a deadlock free adaptive routing algorithm for 2D & 3D mesh without using additional physical or virtual channels. The crux of their work is the realisation that prohibiting one quarter of the possible turns in these meshes is a necessary and sufficient condition for preventing deadlock. The increased number of turns available compared to xy routing allows the development of adaptive routing strategies without the need for adding physical or virtual channels. In the case of a 2D mesh there is a clockwise cycle of turns and an anticlockwise cycle of turns. To achieve deadlock free routing it is necessary to prohibit one turn from each cycle. For a 2D mesh Glass and Ni define three scenarios where two turns are prohibited giving rise to routing algorithms they call, *west-first*, *north-last* and *inverse-first* routing algorithms. All other combinations of prohibiting one turn from each cycle are equivalent to one of the above three scenarios.

Odd-Even Turn Model The adaptiveness of the turn model is uneven, producing only one minimal path for at least half of the source-destination node pairs [71] resulting in *unfairness* and an inability to deal with traffic congestion. To resolve these problems Chiu developed the Odd-Even Turn Model. This model allows all the available turns, but instead restricts the nodes where certain turns can be made. Given columns are labelled with integers starting with zero all columns can be considered to be either even or odd. The algorithms prohibits EN and ES turns in an even column and NW and SW turns in an odd column.

Proximity Congestion Awareness (PCA) Nilsson et al. [72] investigate memoryless switches to transmit messages through a network. To avoid congestion messages can be sent in a non-ideal direction. Nilsson et al. propose a technique called Proximity Congestion Awareness (PCA) to increase the maximum load that the network can cope with. PCA is implemented by transmitting control information, which they describe as a stress value, from each communication node to its neighbours to improve their own routing decisions. Each node can examine the stress value from its neighbours to help decide the best node for routing a data packet. Nilsson et al. report that PCA provides a 20% improvement over making a random choice.

DyXY Routing Li et al. [73] develop a routing strategy called Dynamic XY Routing (DyXY) which provides both deadlock and livelock free routing while outperforming xy and odd-even routing. This solution includes the hardware design of a router to support the routing algorithm. The important attribute of the routing method is for each router node to inform each neighbour of its *stress* level. It this case the stress is a measure of the utilisation of the input buffer.

MD Routing Ebrahimi et al. [74] observe that DyXY routing [73], in the presence of faulty communication channels, can send a packet into communications cul-de-sacs from which the packet needs to be returned before attempting a new route. Ebrahimi et al. extends the

DyXY routing algorithm creating a fault tolerant algorithm which they call MD routing. The innovation is for the each node to transmit, to its neighbour nodes, the status of itself and each of its own neighbours. This is in contrast to DyXY routing which only transmits data about a node's own status. The additional information allows sending nodes to avoid dead ends. This algorithm makes use of virtual channels. In their paper they state that when all available routes have similar congestion status their algorithm favours sending packets down the longest route. This is a useful principle that ensures that, at each step, there are the maximum possible shortest paths to the destination and deserves rigorous study and associated proof.

Hot Potato Hot Potato policy describes the idea of a node passing on a packet as soon as it arrives, without any buffering [59]. The original model was proposed by Baran [59] and further developed by Caragiannis et al. [75] and others.

Lifetime Aware Routing Reliability of NoC routers has a strong correlation with the routing algorithms used by the NoC. A lifetime aware routing acts to route traffic around hotspots to balance the load more evenly between routers and extend the lifetime of the routers and the NoC[76, 77].

2.3.4 Network Topology

The network topology is the arrangement of cores within a network and the interconnections between the cores, both of which will influence the performance of the network. There are numerous possible arrangements each with its own unique properties. The diagrams in this section describe a number of possible topologies represented by nodes used to model processing cores and edges modelling communication channels.

Mesh architectures do not rely on global buses to carry information. Information is transmitted through the mesh from node to node, which replaces the bus overload problem with the challenge of designing efficient routing algorithms.

Square Lattice Dally and Towles [78] propose a grid structure with an Internal routing network which they estimate has a 6.6% area overhead. Kumar et al. [79] propose a square lattice arrangement where each communication node is connected to four other nodes. Each node is also connected to a single processor providing the processor with a connection to the communications grid. This arrangement is often referred to as a *mesh network* or *mesh topology*. Mesh networks are the topology of choice for most research into NoC.

The square lattice is convenient to work with as it is a simple and familiar geometrical arrangement that also translates well to digital circuit designs which also often use layouts that follow a square grid arrangement.

When communication nodes are arranged in a two dimensional mesh and the probability of any pair of resources communicating with each other is a constant, then more messages

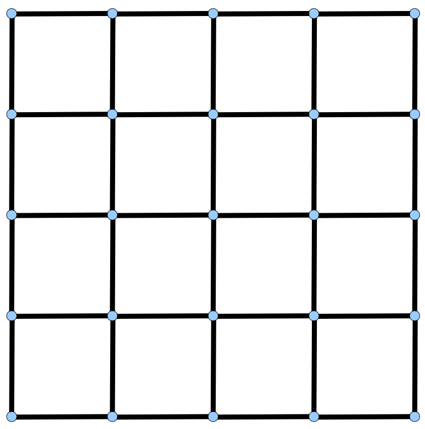


Figure 2.1 – **Square Lattice**

will be routed through the central region of the mesh creating a hot spot of traffic [72].

Rhombic Lattice The rhombic lattice can be constructed using an equilateral triangle grid which is composed of equally spaced nodes such that any group of three equally spaced nodes form an equilateral triangle. The diamonds are formed by connecting nodes using two sets of parallel lines oriented at an angle of 30° to each other as shown in Figure 2.2. The resultant arrangement of nodes and connections is similar to a square mesh placed at a 45 degree angle to the horizontal, but without the uniform spacing between nodes.

The difference between a diagonal lattice and a square lattice can be appreciated by studying Figure 2.2 and 2.1 respectively. These two lattices have the same number of nodes.

In the diamond lattice the distance from the central node to a corner node is the same as the distance from the central node to each of the edge nodes and in this case is 3 steps. In the square lattice the nodes closest the the middle of an edge are closest to the centre. The distance of a node from the centre node increases as nodes are traversed from the middle of an edge, which is 2 steps from the centre, to a corner node, which is 4 steps from the centre.

The corners and edges of the diamond lattice are less connected than in the square lattice; in the diamond mesh the corners are connected to only a single node and the edges to two other nodes, while in the square mesh the corners are connected to two other nodes and the edges are connected to three other nodes.

Hexagonal The hexagonal lattice is constructed using the same equilateral triangle grid described in the Rhombic Lattice section. It is more highly connected than either the square or rhombic lattices and shares similar properties to the rhombic lattice regarding to distance between a central node and the edge and corner nodes.

Hemani et al. [3] propose an alternative configuration of processing units based on a honeycomb configuration. Processors are arranged in hexagonal rings with a communication node in the centre of each ring. In this arrangement each processor is directly connected

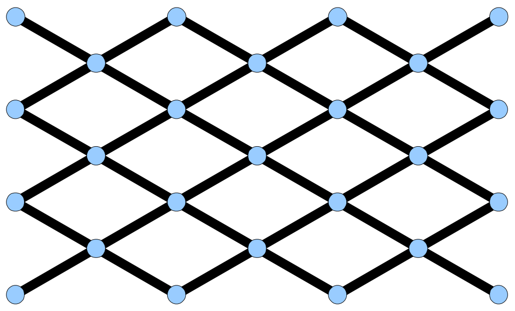


Figure 2.2 – **Rhombic Lattice**

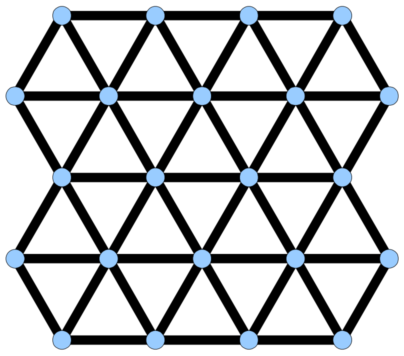


Figure 2.3 – **Hexagonal Mesh**

to its three nearest neighbouring processors and three central communication nodes. Connecting each processor to more than one communication node provides resilience against failure of a single node and the connections between the processor and the node. This resilience comes at a price of increased complexity of communications nodes and routing algorithms.

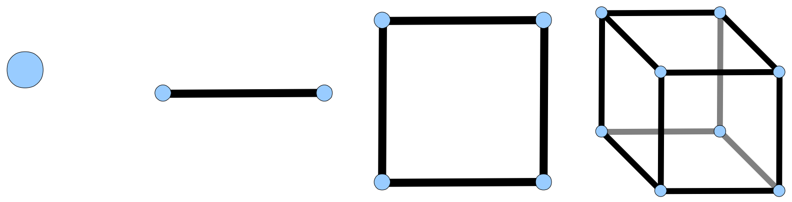
Star The Star topology is discussed in [80] in which they discus Cayley graphs which are defined using a set of generators. They give an example of a 24 node graph which is edge symmetric - that each edge looks identical and is toroidal in nature.

Toroidal A torus can be constructed from a plane surface where the opposite edges are rolled round so that they are connected. Consider a square sheet of paper which is rolled so that the top edge meets the bottom edge and then take the two ends of the cylinder and join them.

Converting a plane lattice to toroid in this manner ensures that all nodes are equally connected which means that there are no edges in the sense that planes have edges. Each node is therefore equally viable for connection to external data sources.

Any of the 2-dimensional lattice structures discussed above can be made into a torus.

Polytopes Polytopes are the generalized set of n-dimensional geometric objects with flat sides. A polytope in n-dimensional space is termed an n-polytope. A 0-polytope is an object with zero dimensions which is a point or vertex, a 1-polytope is a one dimensional object which is a straight line or edge, a 2-polytope is a two dimensional polygon also called a face and a 3-polytope is a three dimensional polyhedron also called a cell.



(a) 0-Polytope Point or Vertex **(b) 1-Polytope Line or Edge** **(c) 2-Polytope Polygon or Face** **(d) 3-Polytope Polyhedron or Cell**

Figure 2.4 – **Polytopes**

As an example the square lattice can be extended into 3-dimensions producing 3-polytope or a cube lattice as show in Figure 2.4d.

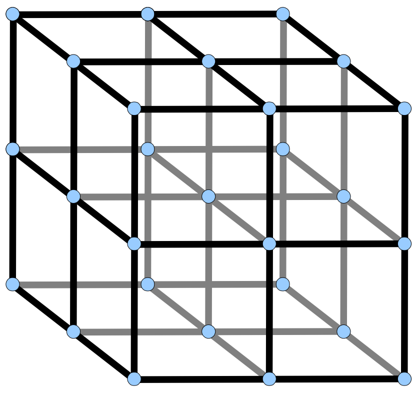


Figure 2.5 – **Cube Lattice**

Since each hypercube is an object that exists in n dimensional space, hypercubes with dimensions greater than 3 need to be mapped to 3-dimensional space which can be achieved by using the hypercube graph representation of the n-dimensional object, denoted by Q_n.

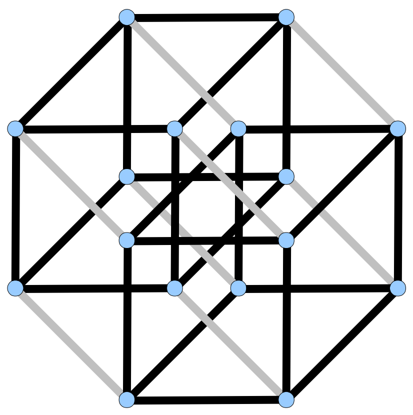


Figure 2.6 – Q_4 **Hypercube Graph**

2.4 Software Considerations in Many-Core Systems

The successful implementation of many-core system is not just an issue of hardware design, of equal importance is the ability to make effective use of the processors through the design and implementation of programs.

2.4.1 Scale Independence

The many-core concept involves cores that number in the hundreds, thousands, hundreds of thousands and millions. The number of cores on a single die will depend on the feature size, the complexity of the processors and the amount of cache memory that is desired, but is likely to be in the tens or hundreds. Systems requiring larger numbers of cores will utilise many interconnected many-core chips. Since the scale of many-core systems may vary in orders of magnitude ranging from 10^2 to 10^6 the design, operation and control needs to be *scale independent*.

Scale independence requires that there is no central control mechanism responsible for configuring or managing all the processors. A central control mechanism has the disadvantage of producing a single point whose failure would be catastrophic to the whole system and a single processor that would be required to undertake ever increasing amounts of work as the number of processors increased. The first of these problems is contrary to the concept of fault tolerance, while the second will place a limit on the maximum number of processing cores that can be utilised contrary to the concept of scale independence.

Scale independence therefore requires that monitoring and configuration are not centralised functions. The model that will be used in this thesis, as discussed in Subsection 2.4.5, is that a collection of processing cores which are physically close will be treated as a region, such that each region has independent monitoring and configuration and can communicate with the monitoring and configuration functions of neighbouring regions. The size of the regions will not be fixed but may be constrained within limits that are proven to be effective during the research.

2.4.2 Application Process Graph

An Application process graph (APG) is a graphical representation of an application broken down into processes with processes represented as nodes and data transfers between processes represented by edges. A constraint is imposed on the construction of APGs that they must be directed acyclic graphs (DAG), a simplification that ensures that there are no closed loops between any arbitrary group of processes which in turn ensures there is no possibility of deadlocks between processes. An application process graph is a standard representation that models processes as nodes and data transfers between processes as edges.

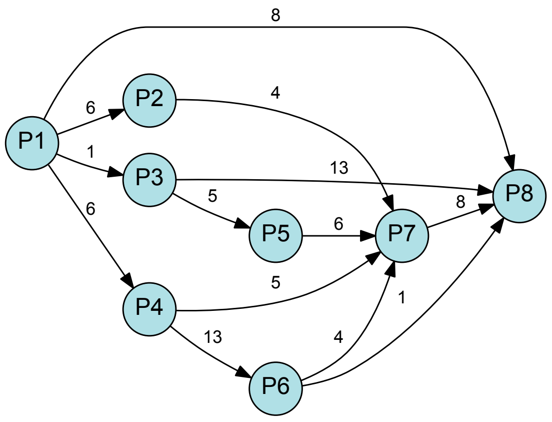


Figure 2.7 – **Example Application Process Graph (APG)**

A Directed Acyclic Graph(DAG) is, by definition, directed, meaning that an edge between two nodes originates from one node and terminates at the second and allows flow from the originator to the terminator and not visa versa. A DAG is, also by definition, acyclic, meaning that there are no close paths through the graph. A consequence of a graph being directed and acyclic is that the nodes can be arranged such that all edges are directed from left to right. Each node can then be assigned an *order* which describes how many nodes are to its left.

Consider an application that is required to process a large number of similar, but different independent data sets, such that on a sequential processor the application will retrieve a data set, process the data set and store the results of processing the data set, and then move on the the next data set. Suppose the application is sufficiently complex that it is possible to break it down into a number of functions, some of which can run concurrently with other functions, then these functions can be represented as processes in the APG.

Now consider the example APG in Figure 2.7, that has processes $P1$ - $P8$ and processes an arbitrarily large number of data sets DS_1 to DS_n. When the application begins, process $P1$ starts by retrieving and processing dataset DS_1. Once process $P1$ has completed its processing of DS_1, including the communication to processes $P2$, $P3$ and $P4$, it is free to start processing the next data set DS_2, so at this point process $P1$ will be processing data set DS_2 and processes $P2$, $P3$ and $P4$ will be processing dataset DS_1. Repeating the same reasoning across the remainder of the graph, it can be seen that four data sets can be in different stages of processing at the same time. The simultaneous processing of multiple data sets is illustrated in the APG Execution Chart as shown in Figure 2.8.

		Time Segment							
		T1	T2	T3	T4	T5	T6	T7	T8
Data Set	1	P1	P2 P3 P4	P5 P6	P7	P8			
	2		P1	P2 P3 P4	P5 P6	P7	P8		
	3			P1	P2 P3 P4	P5 P6	P7	P8	
	4				P1	P2 P3 P4	P5 P6	P7	P8

Figure 2.8 – Example APG Execution Time Chart

Note that the application process graph gives no information as to whether process $P7$ can start processing a data set immediately when it receives data from process $P4$ or $P5$, whichever it receives data from first, or whether is must wait for data from some or all of its inputs before processing can begin; in either case $P7$ cannot complete its processing until it has received data from both $P4$, $P5$ and $P6$. In Figure 2.8 process $P7$ is shown to start processing only when it receives data from process $P4$, $P5$ and $P6$ have all completed processing the data set.

Assuming that all the processes must be completed on every data set, the length of the time segments will have to be no less than the time taken for the slowest process to complete. Processes that take less time to complete will be delayed whilst waiting for the slower processes to compete. As a consequence, the most efficient throughput will occur when the execution time of each of the processes is identical which is an idealisation that, although unlikely to be achieved, is worth pursuing. Or, alternatively, we can say that when designing an APG, care must be taken to ensure that the execution time of the processes is balanced such that the overall waiting time of the processes, as a whole, is reduced to a minimum.

There are similarities between an APG and a staged pipeline used in processor design although the APG is more complex in that there is a greater number of inter-process dependencies and multiple processes can be processing the same data set in the same time segment.

The description of the processing of data sets given above is an example of a coursed grained dataflow machine as discussed earlier in this chapter.

2.4.3 Task Mapping

Task mapping is the process of mapping processes of an APG to processing cores in the many-core system. This research is concerned with investigating methods to efficiently achieve a task mapping of an APG to a many-core system such that the resultant mapping makes efficient use of the resources available in the many-core system[81]. The phrase *efficient use* is subjective and can change from time to time and therefore needs to be able to be influenced by the initial configuration determined by the designers of a system, the current processing demands placed upon the system and the current environmental conditions.

Given an application process graph and a many-core array, *task mapping* allocates application processes as tasks to processing cores. For a graph with n nodes being mapped to a square array of dimension a with $c = a^2$ cores, the number of solutions is proportional to the number of permutations of n from c:

$$^{n}P_{c} = \frac{c!}{(c-a)!} \tag{2.7}$$

The growth of computational time required to search all the possible solutions as the problem size increases places this problem in the NP-hard class of problems. Exhaustive searches are incapable of examining all the possible solutions in a reasonable length of time, so this research will use evolutionary algorithm techniques to search the problem space for solutions.

It is an axiom of this research that the configuration of the system is not predetermined before operation of the system but evolves in response to the performance demanded of the system and the changing environmental conditions. To emphasise this point, there is no predefined initial mapping of any APG to the many-core system, the initial mapping will be determined by the many-core system itself following boot up of the many-core system after which alternative mappings will be evolved. Mapping will take place in real time in response to the hardware status of the many-core system

There has been considerable research on design-time mapping [9–13] which in the vast majority of cases was applied to applications implemented using a collection of intellectual property (IP) modules for the design of NoCs consisting of heterogeneous processing elements (PEs).

Hu and Marculescu [9] analyse a generic video/audio MultiMedia System (MMS) application that includes video and mp3 encoders and decoders and partition it into 40 tasks which are assigned and scheduled onto 25 IP modules. A branch and bound algorithm is used to construct mappings of IP modules to tiles in an NoC architecture and minimize the total communication energy consumption calculated using an energy function that models the energy consumed by sending a single bit between two tiles of the NoC. The branch and bound algorithm is shown to be on average 72 times faster than a simulated annealing optimizer. The algorithm is used at design time to find mappings that are implemented and fixed pre-execution.

Lei and Kumar [10] use a two-step genetic algorithm to map an application represented by a task graph onto tiles on an NoC. The work concentrates on the static mapping of IPs at design time. The calculation of the optimization objective, the *average edge delay*, which is used to measure communication delay shares some commonality with the *network power* objective of Chapter 5 in this thesis, as both calculations make use of the distance between a pair of communication core pairs.

Murali and De Micheli [11] present a mapping algorithm they call PMAP for placing clusters onto processors, which produces mappings with lower communication cost than achieved by previous algorithms. They use bandwidth constraints as the optimization objective, which is also relevant to the *Excess Traffic* objective of Chapter 6. The PMAP algorithm uses Dijkstra?s shortest path algorithm, applied to the quadrant graph (equivalent to a ComPair, see Section 5.1), to obtain the minimum path. The algorithm is used at design-time to find mappings that are implemented and fixed pre-execution.

Ascia et al. [12] investigate the mapping of 12 pre-designed, pre-verified modules in the form of intellectual property (IP) for an MPEG-2 encoder/decoder system, comprising DSPs, generic processors, embedded DRAMs and customized ASICs. A genetic algorithm is used to search for mappings that optimize performance and power consumption for a 5×5 array. The genetic algorithm is used at design-time to find mappings that are implemented and fixed pre-execution.

Derin et al. [14] use integer linear programming (ILP) to search for mappings that minimize the objectives of communication traffic and total execution time of an application in a mesh-based NoC with deterministic routing. The motivation is to combine the steps of allocating tasks to IPs to minimize execution time while at the same time minimizing communication traffic by allocating IPs to tiles. Derin et al. also implement a fault recovery mechanism for a single faulty core that would either reallocate tasks from a faulty core to IPs on other cores, or completely remap all tasks to optimize the objectives. These two approaches to recovery from a faulty core are analogous to the graceful degradation and graceful amelioration used in this thesis.

Sayuti and Indrusiak [13] use an approach that configure both task mapping and priority assignment, using a genetic algorithm to search for solutions. The optimization is performed at design time of a hard real-time embedded system based on a fixed priority pre-emptive NoC. As an alternative to searching for mappings with a GA, a constructive task mapping algorithm can construct task maps based on specific design properties [15].

Das and Kumar[16], Khalili and Zarandi [17], and Chatterjee et al. [18] all use a homogeneous NoC as the target platform and allocate multiple tasks to single tiles containing a PE.

Das and Kumar[16] create a collection of task maps at design time, creating a mapping for each fault scenario which are saved so that they are available to at run-time. When a fault occurs, the map that whose scenario corresponds to the actual fault is used retrieved and used to re-map tasks.

The approach of Khalili and Zarandi [17] to fault tolerance is to allow the mapping of multiple applications, one at a time, and allocate a single spare core to each application during the mapping process. They use a function to calculate the criticality of a vertex in an *application core graph* which is influenced by the amount of traffic flowing into and out of the vertex, sorts the vertices by criticality, and then places the vertices to minimize the *Weighted Manhattan Distance*, which is the product of the distance between two communicating cores and the traffic volume between tasks located at the cores. A heuristic algorithm is used to first place the vertices onto cores and then allocate spare cores for fault recovery. The algorithm is used to map multiple applications onto cores of an NoC and allocates spares cores to specific applications during the mapping process. The algorithm has the effect of clustering the cores used for an application along with a single spare core while leaving the remainder of cores unallocated and available for additional applications. The application mapping is carried out at design time, before the application starts executing and remains fixed thereafter, while the migration of tasks when a PE becomes faulty is a run-time process.

Chatterjee et al. [18] developed an algorithm that provides a unified mapping and scheduling method for real-time systems, focusing on meeting application deadlines and minimizing communication energy while mitigating the effect of failure-prone processing elements. The platform is a NoC consisting of a lattice of homogeneous processing elements, each of which can be allocated multiple tasks by the mapping algorithm. The model determines the communication energy and communication time based on the Manhattan distance between the source and destination nodes. Fault tolerance is achieved by replicating tasks that have been allocated to PEs that are judged to be unreliable. The duplicate tasks either run in parallel with the original or stand by to start execution if the original PE should fail.

2.4.4 Task Migration

Task migration is the action of moving a process from one core to another and involves loading a new core with the appropriate program to continue the processing and where possible transfer of the current state of processing to the new core.

The nature of task migration is different if it is triggered by a hardware fault which requires immediate migration of the process from the faulty core or triggered by the desire to ameliorate performance. If the migration is due to the failure of a core then loss of the current state of processing is usually inevitable in which case the processing that has been lost will need to be repeated or abandoned completely. The migration needs to be immediate to minimize the disruption caused by the loss of processing. Whether the processing is repeated or abandoned depends on whether the application is robust in the sense that it can cope with some level of lost processing or whether it requires all processing to be completed even if this delays the final result. Migration in response to a hardware fault will often only involve the migration of single process.

Implementation of a new mapping to achieve amelioration may require multiple task migrations to complete, in which case the migration of processes will take place in a controlled manner and can be achieved without loss of processing or significant delay to the execution of the process. Task migration is a non-trivial undertaking requiring the loading of the required program to the new core, transfer of the existing execution status, and notification to all interested parties of the new location of the process. Task migration is expensive in terms of processing resources, communication bandwidth and time. The total cost of multiple task migrations required to implement a new mappings needs to be balanced against the long term benefit of the new mapping.

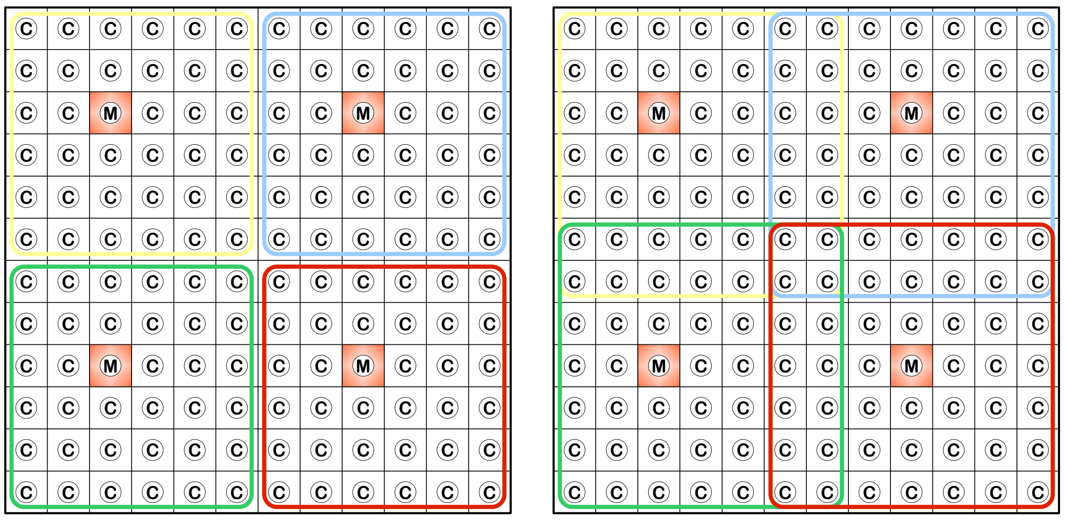
2.4.5 Core Regions

The concept of a monitoring core was introduced in Subsection 2.4.6. The philosophy of this thesis is that the monitoring and management of the many-core system should be distributed function for many-core arrays that have more and a few tens of cores. To preserve scale independence it is proposed that the many-core array is logically divided into a number of regions that are independently managed. A *region* will consist of a collection of processing cores that are physically close to each other and are monitored and configured independently of other regions. Neighbouring regions will interact with each other enabling communication between processes in different regions and allowing migration of processes from one region to another. The size of the regions will not be fixed but may be constrained within limits that are proven to be effective during the research.

The regions that an array of cores is divided into can be disjoint as in Figure 2.9a or overlapping as in Figure 2.9b.

2.4.6 Monitoring

A many-core system is ideally suited to be a dynamic system which adapts to changes to the processing requirements placed upon the system, changes to the system itself and changes in its operating environment. Graceful degradation and graceful amelioration



(a) Disjoint regions **(b) Overlapping regions**

Figure 2.9 – A 12x12 Lattice divided into regions

are achieved through the management of the workload amongst the processing cores and in response to changing conditions that are assessed through the monitoring of the processing cores. Monitoring and management can be a centralised function that has a holistic view of the many-core system however, as stated in Subsection 2.4.1, this produces a single point of failure. In contrast, the approach taken by this thesis is that monitoring and management should be a distributed function carried out by one or more cores within the many-core array.

A *Monitor* process is responsible for collecting performance data and managing the resources of the many-core array to comply with the performance parameters supplied to the Monitor. Examples of the functions that the Monitor node is responsible for are:

- Collecting traffic data
- Collecting thermal data
- Controlling voltage and frequency of individual cores
- Detecting fault conditions
- Managing the fault-recovery cycle
- Maintaining the hardware map with core and link faults
- Maintaining the process map
- Maintaining the application process graph with actual process-process traffic volumes
- Informing routers of the location of processes and faulty links
- Managing process migration

- Managing evolutionary runs to search for process mappings
- Maintaining Pareto Front *Pf0* between evolutionary runs
- Selecting suitable mappings from Pareto Front *Pf0*
- Communicating with adjacent regions (where they exist)

Monitoring of the status of the cores in the many-core system provides the raw data that will be used to determine if there are conditions adversely affecting performance and determine if performance can be improved by reallocating processes to cores in a different configuration. The specific core or cores that undertake the monitoring and management functions is not predetermined.

The system can be designed to allow monitoring for example:

- Processing core faults
- Routing node faults
- Communication channel faults
- Processor bottlenecks
- Communication bottlenecks
- Heat/power hot spots
- Failure to achieve required throughput
- Lack of fault tolerance

2.5 Memory Subsystem

The purpose of a processor is to execute programs to manipulate data. For a processor to function efficiently it must be able access programs and data in a timely manner. This is the role of memory and storage devices.

The size of memory and speed of access to the contents must be compatible with the needs of the processor, consequently memory technology has developed alongside the evolution of processors and computing systems.

This sections reviews the memory subsystems and their development with relevance to many-core system architecture.

2.5.1 Memory Hierarchy

It has been a constant feature of electronic computers that processor speeds are many times faster than prevailing bulk memory systems and the cost of available memory technology increases with the speed of access the memory technology offers. Both of these challenges are addressed by implementing multi-level memory systems. Multi-level memory systems have been a feature of system design from early in the development of computer systems, as is evidenced by Burks et al. [83] who in 1946 proposed a three level storage hierarchy based on criteria of speed, cost and size; the same considerations that are important today [83].

In general the closer memory is to the processor the smaller, faster and more expensive it is, although the technology used in each level has changed over time.

Burks et al. discusses a hierarchical memory system with the top level storage using a cathode ray tube based storage device called a Selectron, the second level using on-line magnetic tapes and the third level using off-line magnetic tapes. Burks et al. also describe a Selectron Register built of flip-flops, presumably using thermionic vacuum valves (since this was before the invention of the first transistor by William Shockley and John Bardeen in December 1947), that is used to store the value retrieved from the Selecton for use by the arithmetic unit. Incidentally the use of flip-flops for main memory was dismissed as impractical, presumably because of the complexity, cost and physical size it would require.

In today's systems the generally implemented hierarchy of memory is: processor registers, on-chip cache memory using SRAM technology (which is often also multi-level), DRAM solid state memory and a hard disk mass storage device. Processor registers and on-chip cache use an SRAM design which is sufficiently fast that a memory access can be completed within a single clock cycle. The SDRAM design uses six transistor per bit and is therefore expensive in terms of space when compared compared to the DRAM used for main storage. Main storage is provided by external memory modules managed by the memory management unit (MMU). The access time to main memory is greater than for cache memory because the memory is off-chip, the distance from the processor is greater and DRAM technology is slower. The DRAM design uses a single transistor and single capacitor per cell compared to the six transistors of the SRAM design. The result is that DRAMS have a higher density i.e. more memory locations for a given die size and ultimately a lower cost.

Cache memory has an access time in the order of a single clock cycle, size in the region of MBs and is relatively expensive. Main memory has an access time in the region of a few 10s to 100s of clock cycles, has size in the region of giga bytes and is medium cost being implemented using DRAM external to the processor. Hard disk storage has access times in the millions of clock cycles, size in the region of tera bytes and is relatively inexpensive. The size of each level in the hierarchy differ by roughly a factor of 10^3.

2.5.2 Cache

Cache memory is a mechanism designed to increase the effective bandwidth of data between the processor and the mass memory and so reduce the amount of time the processor has to wait for data to be retrieved from memory. Addition of cache into processor designs followed the realisation that data access is both temporal and spatial in nature.

To illustrate the nature of the temporal and spatial nature of data access we can use the following small C++ program that uses a loop to sum the corresponding elements of two arrays and place the results in a third array.

Listing 2.1 – Loop program code in C++

```
void ArraySum::sum() {

  for (int x = 0; x <= 10; ++x) {
    arrayC[x] = arrayA[x] + arrayB[x];
  }
}
```

Temporal Proximity Temporal proximity of data access means that memory locations that have recently been used are more likely to be used again in the near future than memory locations that have not been recently accessed.

Consider the program code of Listing 2.1 consisting a simple loop which will be executed once for each element of the arrays. When execution is initiated the program is loaded into main memory from where the processor will retrieve each instruction is in turn. A simple processor that can execute only a single instruction at a time will retrieve, decode and execute an instruction before progressing to the next instruction. The loop test condition and the code internal to the loop will be executed many times over, once for each array element. Each time through the loop each of the instructions are read from memory even though they may have been read only a few clock cycles earlier. This is an example of temporal proximity.

Spatial Proximity Spatial proximity is the property that data that is located in memory close to data that has recently been used is more likely to be used in the near future than more distant memory locations.

Again using the program of Listing 2.1 the elements of each array will typically be stored in adjacent memory locations, and the arrays stored contiguously in memory. As the program progresses through the arrays it will be accessing memory locations adjacent or close to the last memory locations used. This is an example of spatial proximity.

The Cache Model

Single Level Cache: When a processor requests data from a memory location that is not already in cache, the request will trigger the retrieval of not just the memory location being requested but of a *block* of memory containing the desired location. When the requested data is loaded into cache, it is then transferred to the processor which can continue its processing whilst the MMU completes the transfer to the whole block of data. As the processor continues execution of the program it will read further data elements and by virtue of temporal and spatial proximity it is likely to find that the required data is already in cache.

Multi-Level Cache: Cache memory can, and often does, have multiple levels that mirror the memory hierarchy in that cache closer to the processor is smaller, faster and more expensive than cache further from the processor and can be on-chip, off-chip or a combination of both. Cache closest to the processor is referred to as Level 1 or L1 cache, followed by L2 and L3 caches. Processor registers are integrated into the design of the processor so there is effectively zero distance between the registers and the ALU and are smometimes referred to as level 0 cache. On-chip L1 cache, measured in kilo bytes, is placed close to the processor to maintain single clock cycle access times while L2 cache, measured in mega bytes, is placed on the periphery of the processor die and will have multi-cycle access times.

Early microprocessor designs such at Intel's processors up the the i386 did not have any on-chip cache. Even when there is no on-chip cache it is possible to implement off-chip cache using smaller and faster memory devices than those used for main memory, although this would have to be designed into the MMU. As discussed in Section C.3, on-chip cache first was introduced into the Intel processor design in the i486 processor.

Unified and Split Caches The von Neumann architecture has a single system bus used for both addressing and data and a single main storage that contains data and programs while the Harvard architecture has separate data and instruction buses and correspondingly separate data and instruction memories. The Harvard architecture benefits from the use of dedicated buses by being able to simultaneously transfer instructions and data between processor and memory. The design is also more flexible since word lengths for data and instructions do not have to be the same.

In practice, systems are designed with dedicated address, data and control buses where the data bus is used for both instructions and data; which are stored in a single, mixed purpose, main memory. Instructions and data are only separated in L1 cache and CPU internal registers dedicated either to data or instructions while L2 cache is usually a unified instruction and data cache.

Write Strategies When a data item is modified in a cache there are options for how to manage the update to the main memory. These options are referred to as write strategies which are write-through and write-back.

Write-through: A write-through policy immediately updates main memory when a value is modified in the cache ensuring that the value in main memory and the cache are consistent. Each data item in the same cache block that is modified triggers a new update to main memory, creating a significant overhead which reduces the benefit of the cache, especially when multiple modifications are made to the same piece of data.

Write-back: Write-back updates the cache and sets a *modified status bit* linked to the cache entry to indicate that there is now a difference between the cache value and the main memory value. Main memory is not updated until the cache block is removed at which point main memory will be updated with all outstanding data modifications in the cache block. Write-back is faster than write-through because it avoids unnecessary updates to main memory.

2.5.3 Multi-Processor Memory Architecture

Shared Memory In a shared memory system all processors use the same physical address space which refers to a single unified memory through a single interconnect, sometime referred to as a shared memory multiprocessor (SMP) system. The memory management unit coordinates access requests by the processes to the memory. Communication between processors is through shared variables or memory buffers, both of which are located within the shared memory. Applications are responsible for ensuring that they use mechanisms such as memory locks or semaphores to ensure proper synchronization of memory updates.

Distributed Memory In a distributed memory system each processor has its own private physical memory that cannot be directly accessed by other processors. This is typical of clusters comprising many independent machines, each with its own disk drives. Access to by a processor to another processor's memory must be made through messages with the first processor making a request for data and the second processor returning the requested data. Communication between processors via messages requires intervention of the operating system, packaging of the message for transmission, transmission of the message and unpacking, which is inherently a slow process when compared to memory accesses in a SMP system.

Distributed Shared Memory A hybrid arrangement is possible where the memory of a physical address space is distributed among the processors. In this case the memory access times will vary depending of how far the memory is from the processor that is accessing it. This gives rise to non-uniform memory access (NUMA) is contrast to the uniform memory access (UMA) of an SMP system [82].

Shared Memory with Message Passing Another possible memory arrangement for many-core system would be to have a shared memory which is equally accessible by all cores and additional private memory for each individual core sufficiently large to accommodate the program and data for the process. This would avoid repeated accesses to shared memory. Processes that are part of the same application process graph would communicate through message passing so that the shared memory would not be a factor or bottle neck when a data set passes from one node to the next.

2.5.4 Memory Coherence in Many-Core Systems

Each core in a multi-core or many-core system will usually have its own dedicated cache, introducing the possibility of the same data being located in more than one cache at the same time. Copies of a data element that is in multiple locations can become different when one copy is modified by one of the cores, a problem described as memory coherency. Often individual processors will have a dedicated L1 cache and then a group of processors will share a L2 cache. If the number of processors is small then they may all share a single L2 cache, for large numbers of processors it may be desirable to have multiple L2 caches with the L2 caches sharing an L3 cache.

Coherency has been defined by Censier and Feautrier [84] as:

> "A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address."

Maintaining coherency in a multi-core or many-core system necessitates the development of controls and mechanisms to ensure that a modified value is reflected accurately in main memory and that when a process modifies a value it is given exclusive rights to do so until the update is complete.

There are many possible schemes to maintain coherency in multi-core systems. One scheme that has been widely used is based on the write-back mechanism described earlier in this section, with additional flags and signals to indicate when a cache entry has been updated and has not been written to main memory which is then described as 'dirty'. Coherency is maintained by each cache monitoring all data requests from all caches. If a cache identifies a request for an entry in its own cache that has been marked as dirty then it will signal this to the MMU so that its value is used in preference to the value in the next lower memory level [85, 86]. In a many-core array snooping requires all transactions to traverse the entire network, increasing the traffic in proportion to n^2 of the number of nodes which grows rapidly with number of nodes making it unsuitable for large many-core arrays.

An alternative to the above scheme is a directory based approach where the MMU maintains a list of which caches have copies of memory blocks. When a write request is sent from a cache to the main memory the directory is used to inform all other caches using the same block that their data is now invalid [87] [84]. The scheme reduces the traffic compared to the snoopy schemes as only affected caches need to be informed when data is modified in main memory. Although an improvement of the the snoopy scheme the directory based approach has its own limits because the sizes of the directories will limit the number of L1 caches that can be watched.

A further development of the directory based approach is using distributed directories [88–92] where directories are maintained in the network nodes instead of centrally with the main memory. In Eisley et al. the directory is maintained as a linked tree between the nodes that have of copy of the same data [91].

2.6 Programming Models

Mainstream traditional programming is primarily based on imperative programming languages that were designed for writing programs for the single instruction execution stream of computers with a von Neumann architecture. However imperative languages are not well suited to writing programs that involve parallel processes. This chapter review a range of programming paradigms that have been developed to overcome the limitations of imperative languages.

2.6.1 Evolution of Computation

The development of mechanical computation goes back to the 15^{th} century with Wilhelm Schickard (1592-1635), Pascal (1923-1662), Gottfried Wilhelm Leibniz (1646-1716) who all designed mechanical calculation machines. Leibniz, a mathematician and philosopher, developed a system of calculus at the same time as Newton and it is Leibniz's notation that is now used. Leibniz work on mechanical computation and recognition that mathematical algorithms are a list of steps that can be followed mechanically inspired him to posed the philosophical question of whether there is an algorithm that could decide the truth of statements in number theory. This together with Hilbert and Ackermann's 1928 Decision Problem, the Entscheidungsproblemn in his native German [93], was the catalyst for later generations of mathematicians to develop formal methods for computation. In the process of attempting to answer these problems many researchers in the field of mathematics developed calculus and conceptual computing machines. The 1930s and 1940s were a particularly productive for research into formal systems with the publication of Gödel's 1931 paper [94–96] where he sets out the theory of recursive functions, now called primitive recursive functions following the work of Kleene; Turing's 1936 paper on Turing Machines [95, 97]; Post's 1936 paper on finite combinatory processes [98] and Church's 1941 book "The calculi of Lambda-Conversion" [99]. Shepherdson and Sturgis in 1963 developed the Unlimited Register Machine (URM) [100] which can be viewed as a modern equivalent of Turing Machines that allows functions to be expressed in a very simple programming language making it easier to work with than Turing Machines. These alternative approaches to formal systems and computability have all been shown to be equivalent and in particular equivalent to simple recursive functions, which is now referred to as the "Church-Turing Thesis".

The importance of all these systems to computing is that computers, however sophisticated, are also equivalent to Turing Machines. Shepherdson and Sturgis's URM needs only four operations to be able to specify any function. The same is true for real computers; any additional instructions or complexity of design do not increase the computational capability of the machine but rather make the machines easier to program and more efficient.

The theory of primitive recursive functions gives formal proof of the validity of concatenation, substitution and recursion of functions all of which underpin all programming languages while Church's Lambda Calculus is fundamental to the theory and design of functional and dataflow languages.

2.6.2 Imperative Languages

In 1978 Backus described languages that execute a single instruction at a time in the sequence specified by the programmer as 'von Neumann languages" [26], although are more accurately described as *imperative languages*.

The first stored program computers were single processor computers that executed a single instruction at a time described as the stored-program computer or von Neumann architecture [101]. As a consequence, the first programs were written to execute a single instruction at a time in a specific order to achieve the desired effect. Both procedural languages and object-oriented languages are sequential. The methods of a class in an object-oriented language contain sequential code as does the code within functions and procedures of procedural programming languages. There is nothing inherent in these languages that promotes parallel processing. Where parallelism is required it is provided by incorporating application program interface (API) directives to an external application with provides multiprocessing support.

It has been argued that procedural programming constrains the programmer to thinking an coding solutions in terms of the von Neumann architecture [26] and that a functional programming approach would free programmers from this restraint.

2.6.3 Functional Programming

In his 1978 paper Backus highlighted the problems of imperative languages and proposed a functional style programming model. Backus's approach is very mathematically orientated being based on Church's Lambda calculus.

The properties of a functional language that distinguishes it from an imperative language are that functions are:

- independent: they compute irrespective of other computation states
- stateless: each action is unrelated to any previous action
- deterministic: the same inputs will return the same outputs

Functional programming languages attempt to adhere as closely as possible to these properties. However in real systems, the purpose of programs is to change the state of the system. For example, a function that updates database records is changing the state of the system.

A number of languages fit into the functional language category: Haskell is strongly based on Lambda calculus and is one of the purest functional languages. Python is a widely used, popular language that has features which can be described as functional.

2.6.4 Dataflow Programming

Functional programming and dataflow programming are closely related in that they both promote the same above properties. Functional languages more closely follow the mathematical ideal, whereas dataflow languages were designed for programming dataflow machines with an emphasis on parallel programming.

The roots of dataflow programming can be traced back to the 1960s and 1970s when several groups of researchers developed dataflow architectures [28, 33, 102, 103] and dataflow languages [104–108]. The motivation for dataflow machines was to overcome the bottlenecks inherent in the von Neumann architecture by designing machines capable of instruction level parallelization. The name dataflow arises from the change of emphasis from instructions requesting the data they require during execution to instructions being triggered when the required data is available for their execution. Any instruction can be executed by any available processing element as soon at the required data becomes available. This allows multiple instructions that have no data interdependencies to be executed simultaneously.

The dataflow programming model uses a directed acyclic graph, sometimes referred to as a dataflow graph [28, 29, 104, 107, 109–111] to represent program code as it would be executed on a dataflow machine. The earliest description of a dataflow graph, by Karp and Miller, is a directed graph of nodes and edges that Karp and Miller refer to as a *Computation Graph* [104]; the edges are augmented by four integers representing information about the input and output data queues of the nodes and the data consumed and created by the nodes and are not acyclic. Dataflow graphs are not a programming language, they are a visualization that can be derived from the program code, representing what the program will do.

With each new machine design, a corresponding programming language was required. While each machine may have required its own language to best exploit its architecture, all languages shared common fundamental features explicitly expressed by Ackerman [27] and summarised as:

1. Freedom from side effects
2. Locality of effect
3. Equivalence of instruction scheduling constraints with data dependencies
4. Single assignment of variables
5. An unusual notation for iterations due to features 1 and 4
6. A lack of history sensitivity in procedures

Conway in 1963 set out conditions for separating a program into processing modules [112] that are the same conditions that are required of a DAG and a pipeline dataflow [113]:

1. the only communication between modules is in the form of discrete items of information
2. the flow of each of these items is along fixed, one-way paths
3. the entire program can be laid out so that the input is at the extreme left, the output is at the extreme right, and in between all information items flowing between modules have a component of motion to the right.

Although Conway was not designing a dataflow language his aim was to decouple modules such that the output of a module was dependent only on the input it received, which is a essential property for nodes to process in a DAG and could be added as a fourth requirement to Conway's list.

The overhead that accompanies parallelization is considerable and in the case of instruction level parallelization can be significant when compared to the gains. This, coupled with the rapid improvements in the designs of classic von Neumann machines, resulted in the dataflow machine architectures of the 1960s and 1970s falling short of the anticipated benefits to the extent that the research on dataflow machines was eventually abandoned. Architectural gains of von Neumann machines is finite; clock speeds have levelled off, pipelines are near optimal, and cache can be as large as required. Parallelization is still required to realise higher processing throughput so the search for methods to achieve parallel processing continued. The focus of research moved from instruction level parallelization first to the parallelization of loops [114–117]. Individual iterations of loops can be run in parallel if the iterations are independent. However this is not always the case making it difficult to increase processing throughput using multiple processors. Research next looked at the gains that may be achieved with threads [118–120].

Each stage of research has increased the granularity of the units of code being parallelized. As already noted instruction level parallelization was found to be too fine grained. The same is largely true of isolated loops, with processor clocks running at 3 GHz and pipelines achieveing near single cycle instruction execution, many millions of loop iterations can be executed in a very quickly.

Early dataflow machines with a fine-grained instruction level parallelism and the sequential processing von Neumann machines can be viewed as the extreme ends of a continuum. Both, in their pure form, have performance problems. There is now an expectation that the best performance will be achieved somewhere between the two. Current research into parallelization can be viewed, in this context, as a search for the appropriate level of granularity to achieve useful levels of parallelism while keeping the related overhead small compared to the gains [29]. As hardware continues to evolve and programming models mature, the granularity the produces the best results may also change.

The continued hardware evolution that now allows many-core system to become a reality gives a new opportunity for the dataflow programming model to realise the ambition of its original researchers.

2.6.5 Parallel Programming Models

Explicit parallel programming, where programmers have to add constructs into code to enable it to run in parallel, has been available for some time with platforms such as OpenMP and CUDA. An alternative approach is that of implicit parallelism which infers from the code where parallelism can be implemented.

OpenMP OpenMP is a set of directives that can be used within a supported language such as C++ that enables to programmer to explicitly designate code to run on multiple processing cores and handle synchronisation issues. OpenMP as its name suggests, is an open standard.

CUDA CUDA is a collection of compiler directives and language extensions to languages such as C++ that allows programmers to make explicit use of the multiple processing units within NVIDIA GPU cards. Originally intended for graphics programming CUDA is also used to access the power of the GPU cards for highly parallelizable computations.

Implicit Parallelism Hwu et al. promote an implicit parallel programming model in which the compiler is responsible for identifying parallelism in contrast to the OpenMP approach in which programmers explicitly include directives to enable code sections to run in parallel [121].

The use of existing traditional languages do not adhere closely enough to the properties described by Ackerman and Conway to be amenable to implicit code parallelism. Implicit parallel coding will require new languages which are more rigorous such as Haskell which has been designed to closely implement Lambda Calculus.

2.6.6 Code Parallelization

There are at least three conceivable ways in which parallelization of processing can be achieved:

Instruction Level Instruction level parallelism is currently implemented through super-scalar and pipeline architectures. Early dataflow machines were designed as instruction level parallel machines before the realization that this was too fine-grained to be practical.

Array Processing Using an array of processing cores that simultaneously process the same instruction/program on different data sets. Commonly used in specialized coprocessors such as GPUs and useful primarily when many datasets need to be processed by the same function simultaneously. The processing elements run the same code at the same time, with possibly small variations.

Multi-Thread Processing This is the model implemented by systems such as OpenML, where the same code is run on many processors as threads. The threads are more independent that for array processing and much more flexible in the code that can be run.

A Course-Grained Dataflow Using a many-core array and an APG to achieve pipeline style processing of multiple data sets simultaneously. This is similar in concept the the original instruction level dataflow model but with a much more coarse grained structure.

2.7 Summary

This chapter has reviewed the evolution of computer processing paradigms, technological advances and challenges that resulted in Hemani et al. proposing the many-core model in 2000. Feature sizes are now small enough that single devices can contain billions of transistors, making available the technology to fabricate many-core systems with hundreds or thousands.

A major challenge for system designers using devices with feature sizes below 22nm is the variability of operation and reliability of individual transistors and the need for dark silicon to ensure that the device does not overheat. The variability and reliability of individual transistors will increase the occurrence of faults, some of which may be transient while others will be permanent, causing loss of the processing unit affect by the fault.

The large number of processing cores envisaged in many-core systems present an opportunity to help alleviate the problem of reliability due to failure of processing cores or communication links. Appropriate management of the use of individual cores can also be effective in management of dark silicon.

This thesis demonstrates one possible approach to the management of the use of cores to implement fault tolerance and management of dark silicon.

Chapter 3

Fault Tolerance

Fault tolerance has been, and continues to be, an important element of designing reliable computer systems. Avizienis in 1978 [122] provides the following definition of fault tolerance:

> "Fault-tolerance is the architectural attribute of a digital system that keeps the logic machine doing its specified tasks when its host, the physical system, suffers various kinds of failures of its components."

Since Avizienis gave the above definition the scope of fault tolerance has been widened to include software as well as hardware faults. Avizienis and Kelly [123] make the distinction between a *fault* that can exist within hardware or software without manifesting itself and an *error* which is the result of a fault producing an incorrect output.

Continually diminishing feature size of devices increases the probability of fabrication defects and variability of performance of individual transistors [124, 125]. Use of technology in hostile environments continues to increase with a corresponding increase in the likelihood of single event upsets (SEU) and damage to individual components leading to failure of complete systems. Inaccessibility of systems can make maintenance impossible or prohibitively costly. All of the above scenarios can benefit from the application of fault tolerant strategies.

Microprocessors are now being fabricated with transistor counts in the range of 10^8 and 10^9 per device. In these devices the failure of a single transistor during operation may render the whole device unusable, requiring its replacement. As manufacturers continue to strive to obtain higher transistor densities and lower power consumption, the feature size is continually being reduced. For example, feature sizes of 32nm appeared in commercial devices in 2010 in which the size of a single transistor is in the region of 60 atoms of the silicon lattice. Components of this size reduce the predictability of the behaviour of the device and increase the likely hood of damage to transistors causing device failure. In 2005 it was recognised by Borkar [125] that reducing feature size would cause problems:

> "As technology scales further we will face new challenges, such as variability, single-event upsets (soft errors), and device (transistor performance) degradation - these effects manifesting as inherent unreliability of the components,

posing design and test challenges."

3.1 Reliability

The existence of soft errors or single event upsets (SEUs) in which a single bit of information is inverted due to external energetic particles entering the device have been recognised since the 1970s [126]. The source, for example, can be alpha particles from radioactive contaminants in the packaging, high energy cosmic rays and low energy cosmic rays [127]. Decreasing feature size magnifies the effect of energetic particles, increasing the frequency of SEUs and in some cases causing multi-bit upsets (MBUs).

Detection and correction of single and multi-bit errors can be achieved in hardware using a variety of techniques such as error correcting codes [128] and radiation hardened designs [127]. As feature sizes decrease the effect of process variations increase, leading to higher variation of the properties of the individual transistors in devices. Process variations and other manufacturing defects, also accentuated by reduced feature size, act to decrease the yield of monolithic processor designs. Many-Core Arrays on a single die are tolerant to the failure of individual cores causes by manufacturing defects. The same tolerance during the lifetime of the device will allow the many-core array as a whole to continue to perform useful work even when a significant proportion of individual cores fail.

3.2 Error Detection and Correction Codes

One of the earliest forms of fault tolerance was the devising of codes to represent information with error detection properties. The simplest error detection code is parity, which can detect one single bit error. More sophisticated codes have been developed to detect more than one error or to correct errors.

Alt (1948) [129, 130] wrote two papers describing the design of "A Bell Telephone Laboratories' Computing Machine" in which he describes 2 of 5 and 3 of 5 codes that are capable of identifying the occurrence of single bit errors. The computer was designed to stop when errors were identified to allow intervention to correct the problem.

Hamming's 1950 paper [131] gives a mathematical treatment of error detecting and error correcting codes. Hamming defines the *distance* between two codes, which we now know as the *Hamming Distance*, and the relationship between various distances and the ability to detect or correct a given number of single bit errors.

Fontainet and Gallagert [132] analyse communication data and made the observation that errors are not evenly distributed but occur in bursts. Fontainet and Gallagert conclude that, for the data under scrutiny, error correction was impractical whereas error detection was practical with a negligible probability of undetected errors. They also state that the choice of error detection and correction codes should be determined by the profile of errors within the data.

Bell Labs applied for a patent *Pulse code communication* which included the coding scheme now referred to as *Gray Codes* [133] and now classified as a canonical binary single-distance code.

Berger and Mandelbrot [134] propose two models for describing distributions of errors while Mandelbrot [135] gives a self similar model of the distribution of errors, noting that the distribution of errors appears the same regardless of the scale used to examine the distribution. The property of self similar scale independence is the same property possesed by fractals, one of the earliest and most well-known of which is the Mandelbrot Set. Another conclusion of Mandelbrot is that errors are unavoidable so that effort should be directed at detection and correction as well as reduction.

Peterson and Brown [136] present a mathematical treatment of cyclic codes, now referred to as *Cyclic Redundancy Check (CRC) Codes* using generator polynomials to define codes.

3.3 Hardware Redundancy

Redundancy is a widely used approach to fault tolerance which takes many different forms [137]. The primary objective is to ensure that in the event of the failure of a module, there is a backup module that can replace the faulty unit. Some systems require the replacement to be carried out manually, while more sophisticated approaches make the transition automatically.

3.3.1 Double Modular Redundancy (DMR)

Double modular redundancy is an approach based on designing systems with a primary module and a secondary module which can replace the primary module in the event of a fault. It protects against single module failures in the primary module. Typically, the secondary modules have no active part in normal operation, other than to ensure that they are in a position to take over operation from the primary module. This model was used from 1976 by Tandem Computer Systems in its range of commercial computers [138].

Duplicate modules are designed as an integral part of the system and remain in situ during normal operation. Duplicate modules are passive backup modules as they remain in a standby mode taking no active part in the normal operation of the machine. Each primary module gives regular updates of its status to its corresponding backup module so that the backup module can resume operation starting from the last status update should the primary module fail. The design relies on monitoring software and hardware that can detect failure of a module and switch processing to the backup module. When a fault has been detected processing is automatically continued by the backup module to enable the machine to maintain normal operation. Where the system is accessible an engineer is alerted who can replace the failed unit without nay downtime of the system. The primary aim is to reduce the mean time to repair from hours to milliseconds and increase the overall availability of the system.

The Voyager spacecraft had three dual-redundant computer systems [139]. Voyager's computer architecture illustrates the need for testing and monitoring the health status of primary systems and a mechanism to switch to backup systems. In Voyager's case the modules carried out self test which generated status signals, the CCS monitored the health of the modules via the status signals and was also responsible for switching to backup systems.

Figure 3.1 – An NASA artist concept of Voyager 1 and 2 which used dual-redundant computer systems

3.3.2 Triple Modular Redundancy (TMR)

The philosophy behind TMR systems is fundamentally different to that of DMR systems. TMR systems are usually based on using three replicas of a module, carrying out the same processing. The outputs of the three modules are routed through a simple majority voter module to determine the definitive output.

TMR has been implemented in real applications such as the Saturn V launch which uses TMR with voting elements in the design of the central processor [140] and the fly-by-wire primary flight computer for the Boeing 777 aircraft [141].

The majority voter is based on Von Neumann's work [142] where he defines an automaton which he calls a '*majority organ*' that gives a single output which is the same value as the majority of three inputs as defined by the boolean equation 3.1.

$$m(a, c, b) \quad \equiv \quad ab + ac + bc \quad \equiv \quad (a + b)(a + c)(b + c) \tag{3.1}$$

The basic TMR architecture protects against a single fault in any one of the three duplicated modules as illustrated in figure 3.2.

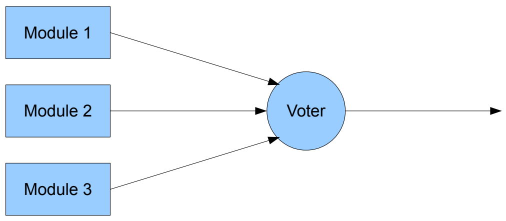


Figure 3.2 – Triple Modular Redundancy arrangement using a von Neumann inspired voter

Lyons and Vanderkulk [143] examine TMR for use in applications in challenging environments such as those encountered in space exploration and the military. Lyons and Vanderkulk look exclusively at permanent component errors, giving a mathematical treatment of TMR to show that TMR can increase the overall reliability of a system. They note that TRM works with systems that already display a high level of reliability - the higher the reliability of the non-redundant system, the more beneficial is the effect of implementing TMR. The effect of TMR when used with unreliable components is reduced and can, in some cases, reduce the overall reliability. In particular if the original component has a reliability probability of less that 0.5, then TMR cannot improve the reliability of the system.

The basic implementation of TMR, shown in figure 3.2, moves the problem of a single point of failure from the replicated module to the voter module. A more sophisticate arrangement is given by Lyons and Vanderkulk [143], figure 3.3, which also triplicates the voter modules. The diagram shows the outputs of stage one being routed through triplicated voters. The outputs of the voters are used as the inputs to the next module which is also triplicated. This arrangement can be repeated for each triplicated module. However, if the system is not closed so that there is a final output, the voter for the last output will still become a single point of failure as it cannot be triplicated. Lyons and Vanderkulk 's improved arrangement is the focus of their paper [143].

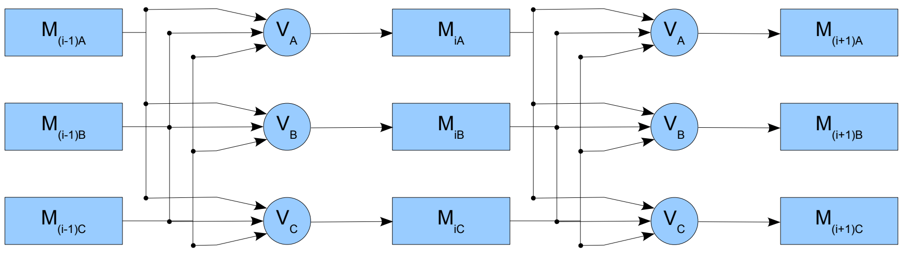


Figure 3.3 – Triple-modular-redundant configuration.

The traditional implementation of TMR uses bit-by-bit voting schemes. When the redundant modules have n outputs bits, there are n voters, with the i^{th} voter calculating the majority vote for the i^{th} bits of the modules. Bit-by-bit voters can only correct single module errors. If errors occur in more than one module such that any two modules produce an error on the same bit then the bit-by-bit voter will not detect the error and produce an incorrect output.

Mathur and Avizienis [144] give a detailed mathematical treatment of N-tuply modular redundant system i.e. systems with N copies of the module. Mathur and Avizienis also discuss a hybrid(N, S) system where there is an N-tuply modular redundant system and S spares which can be switched in when one of the N redundant modules fail.

Abraham and Siewiorek [145] partition a network into cells and calculate the reliability of each cell in order to simplify the calculation of the reliability of the the whole network, which they state as improving the accuracy of calculations over previous work.

To improve the response to multiple module errors, Mitra and McCluskey [146] developed the TMR architecture further by proposing a word-voter. In the word-voter additional circuity is added which examines the output words of each module. If two modules are in error and produce different errors, the outputs of all three modules will be different to each other. In this case the word-voter produces an error condition. While it is not possible to determine what the correct value should be it is possible to say that an error has been detected and report the condition. This word-voter does not help identify errors when two modules are in error and produce identical errors. In this case the word-voter will still produce an erroneous value and will not produce an error condition.

Commercial off-the-shelf (COTS) FPGAs are an attractive choice for development of systems because of the low cost and mature development compared to an ASIC, even in hostile environments such as space-based systems[147]. SRAM based FPGAs are, however, vulnerable to single event upsets (SEUs) which can be caused by radiation in space-based processing [148]. Implementations using FPGAs in such an environment need to use hardened designs and verification to protect against SEUs [149, 150]. Quinn et al. [147] investigate the limitations of TMR when presented with multiple-bit upsets.

Wakerly (1976) [151] shows that careful use of TMR can improve the reliability of microcomputer systems, although he notes that because of the complexity of microprocessors the additional TMR circuitry could be more unreliable than the microprocessor itself. This research was carried out when the Intel 8080 microprocessor was popular. The Intel 8080 was one of the early 8 bit microprocessors containing in the region of 6,000 transistors. Modern microprocessor systems such as the Intel Itanium, released in 2001, contains 3.1 billion transistors and do not use TMR, instead relying on error detection and correction mechanisms [152].

3.3.3 Synchronised Redundant Systems

An important point to note with voting systems is that the modules need to maintain synchronisation with each other. Independent systems can quickly fall out of synchronisation simply due to small variation of clock rates.

The designers of the Space Shuttle's primary computer systems took a different approach when using multiple redundant systems to provide fault tolerance [139]. The original design was to use five separate systems all carrying out the same processing with a voting mechanism. This configuration would allow two failures and still provide a majority voting with the three maintaining systems [139].

During development of the systems the configuration evolved into four computers running the same programs and maintaining synchronisation with the fifth computer providing a backup that could be switched in to replace one of the others. A major issue with this configuration was maintaining synchronisation. Varying clock rates between the computers resulted in a rapid loss of synchronisation which required more frequent synchronisation than originally envisaged. When events such as input, output or completion of a software module occurred the computers would attempt synchronisation.

3.4 Fault Tolerance in Memory

Various schemes have been used to increase the yield of semiconductor memory devices, from error correcting codes, single cell replacement during production, to 1-D and 2-D built in self repair (BISR) strategies.

Chen and Hsiao (1984) [128] give a review of the state-of-the-art error correcting code mechanisms in use at that time. Error correcting codes (EXCs) can extend the life of a memory device by detecting and correcting data errors. The device will only fail when the quantity of errors exceeds the ability of the ECCs to compensate.

In the early 1990's all memory devices were designed with redundancy so that faulty cells could be replaced using lasers and fuse/antifuse techniques within the production process [153]. Chen and Sunada [153] present a structure that allows the memory to self test and self repair outside of the production environment and without the need for external equipment for intervention. Aichelmann [154] makes a review of ECCs and other fault tolerant design strategies.

Kim et al. [155] propose a design where additional columns are added along with built in self test (BIST) and built in self repair circuitry. When faulty cells are detected spare rows can be brought into use by employing multiplexers. This 1-D self repair model has been an active area of research by others [155–158].

The 1-D model has been further developed into a 2-D model where yield of semiconductor memory devices is increased by adding spare rows and columns which can be used to replace faulty cells in the main fabric of the device. When a primary memory cell is faulty, either the row or the column of which it is a member is made obsolete and a spare row or column configured to take its place. As the number of faulty cells increases the problem of determining if there is a mapping of the spare rows and columns that can cover all fault cells simultaneously is an NP-Complete Problem [159]. Various algorithms have been developed to to determine an appropriate mapping [159–166].

3.5 Fault Tolerance in Disks

Patterson et al. [167–169] present the case for RAID (Redundant Arrays of Inexpensive Disks) as an alternative to the prevailing model of SLED (Single Large Expensive Disks). They note that while the capacity of large disks had steadily increased there had been only moderate increase in performance. The RAID concept was based on making use of disks designed for the personal computer market.

The major issue with RAID disks is the failure rate as measured by MTTF. Patterson et al.'s calculations show that an array of 100 disks have an MTTF of 300 hours, less than three weeks, which only gets worse as more disks are added. This makes fault tolerance an essential part of the design of the RAID model.

Gibson et al. [170] investigate various redundancy codes for use with disk arrays. In their paper they note that disk support hardware will be subject to failure and may support a bank of disks. They note that if disk parity groups are arranged orthogonally to disk groups connected to common support hardware, then this will protect against failure of the support hardware.

Blaum et al. [171] propose a scheme they call EVENODD which can be implemented as standard RAID 5 disk controllers which protects against the simultaneous failure of 2 disks.

3.6 Fault Tolerance in Many-Core Systems

As discussed in Section 3.3 designers have used a variety of strategies to protect single and multi-core processors from faults including double modular redundancy, triple modular redundancy, synchronised redundant systems and error detection and correction mechanisms.

Homogeneous many-core systems offers the opportunity to develop and implement novel fault tolerant strategies by utilising the inherent redundancy offered by the availability of large number of identical processing elements on a single chip.

Lei et al. [172] propose a redundant core arrangement they call $N + M$ which describes a many-core array with a physical arrangement of N processing cores and M redundant cores. The aim is to use firmware level reconfiguration of the available cores to present a consistent and regular logical core topology of N cores to the operating system and programming level. When all of the N processing cores are fault free none of the M cores are utilised. In the event of one of the N cores failing, the cores would be reconfigured to bring into use one of the spare M cores. This technique increases yields of many-core chip, presenting N functional cores, when up to M processing cores are faulty.

3.6.1 Fault Recovery

In the context of this thesis fault recovery will involve the smooth migration from an existing process mapping to a new mapping to achieve a performance improvement while minimizing disruption to the processing of the application. For fault recovery to be successful the recovery mechanism will need to:

- Monitor performance of nodes and processors.
- Identify when a fault occurs.
- Migrate tasks from a faulty core to a spare core.
- Search for new mappings.
- In real-time migrate a process from one core to another to implement the new task mapping.

3.6.2 Response - Slow Versus Fast

One of the objectives of the search algorithm will be to find mappings that exhibit a level of fault tolerance. Fault tolerance in the case the many-core systemis protection against various hardware failures. This is achieved by ensuring that unused *idle* cores are placed close to every active processing core.

When fault is detected with a processing core the monitoring node will initiate a task migration to the closest idle core. There is no need to search for alternative configurations at this time, the only requirement when a core fails is to migrate the task to the nearest available Idle core as soon as possible. This is a quick fix, but because the idle cores should be evenly distributed across the core array, the task mapping should not appear radically different.

Once the task migration has been completed resolving the immediate issue of fault recovery the system can move on the the next stage which is searching for a new long-term solution. Since the topology of the system has changed it is possible that there may be more efficient alternative mappings. The evolutionary algorithm will search for new mappings using the current monitoring data and network topology. If a suitable alternative mapping is found then the Monitor will initiate a series of task migrations to implement the new mapping.

3.6.3 Graceful Amelioration and Degradation

Hardware faults in processors, communications channels, and communication routers are likely to cause immediate disruption to one or more processes. In these situations the desire is for the many-core system to minimize the disruption to the system as a whole and take steps to reconfigure to compensate for the failure to enable the system to continue functioning even if the overall performance is reduced. This is described as *graceful degradation*.

Other fault conditions such as poor performance or high power usage indicate that parts of the system, if not the whole system, is under stress. It is possible that these conditions can be improved by suitable reconfiguration. For example a hot spot can be alleviated by moving a task to a processor in a less used region of the array or sharing the work among multiple processors. Migrating to a configuration that improves some aspect of performance is described as *graceful amelioration*.

Many-core systems will have cores numbering hundreds or thousands. With so many cores available, the objective is not to use all of the cores all of the time but rather to maximise the performance of the system as a whole. Mappings of the application network to the many-core array will not use all available cores ensuring that there will be spare cores to recruit when a core fails. The spare cores provide the raw material for implementing strategies to enable graceful degradation and graceful amelioration. This philosophy of the managed underutilization of cores coincides with the need to maintain dark silicon.

3.7 Conclusion

This chapter has reviewed a range of approaches to fault tolerance used for a variety of hardware.

This research will concentrate on the recovery phase making, the assumption that for each scenario the fault has been detected, diagnosed and identified. Examples of possible scenarios are the complete failure of a single core, failure of a communication link, failure of a routing node, intermittent core faults and anticipated loss of a functioning core.. Examples of performance related scenarios are communication bottlenecks, high power consumption and sub-optimal performance.

This research will use a wider definition of faults than has been customary in previous work. As well as actual hardware faults, conditions that can be interpreted as stress to the system and can be alleviated by a change in configuration will also be considered to be fault conditions.

Faults can be either hardware related faults such as:

- Processor hardware faults
- Communication channel faults
- Communication router faults

or performance related faults such as:

- Slow performance

- Excessive use of power
- Generation of heat spots
- Processor bottlenecks
- Reduced fault tolerance

Errors in software are not included as faults in this research.

Many-core systems could be used to mimic approaches such as dual modular redundancy or triple modular redundancy, however both these approaches require the backup modules to be active, so increasing power consumption. The requirement for dark silicon is a result of the need to reduce heat generation of the many-core system, so solutions that increase power consumption are incompatible with this requirement.

The approach to fault tolerance of this thesis, which places spare cores strategically among processing cores, also requires the spare cores to remain in a low-power standby mode when not in use.

Chapter 4

Core Fault Tolerance

This chapter will explore the core fault tolerance objective using a single objective evolutionary algorithm to search for fault tolerant task mappings. Although any suitable search algorithm can be used to search the solution space, the evolutionary algorithm approach has been chosen because it has proved to be effective for searching the multi-objective solution spaces that dominate this research.

The concept of core fault tolerance in a many-core array is to leave a proportion of the cores unused, referred to as *spare* or *idle* cores so that when a core that is processing a task, becomes faulty, the task can be migrated to one of the spare cores. The presence of spare cores also create dark silicon, introduced in Subsection 2.2.3.

4.1 Assumptions

Migration of a task to a spare core is a non-trivial task which is not within the scope of this research, however it is assumed that the mechanism required to migrate a task will be available. Since task migration will require the transmission of the status of the process and associated local data to a spare core, the *target spare core*, and the cost of transmission of information is proportional to the number of routing nodes that the information passes through, it is assumed that the cost of migration is approximately equal to the number of routing nodes between the failed core and the target spare core.

It is assumed that minimizing the distance between a failed core and a spare core will also minimize the duration taken to migrate the task and restore processing thus minimizing disruption to the processing.

Since the cost of migration is proportional to the distance between a failed core and the target spare core, the cost of migration of the task from a processing core to a spare core can be minimized by evenly distributing spare cores amongst the processing cores thus minimizing the distance between each processing core and its nearest spare core.

This chapter will introduce a simple many-core model, evolutionary algorithm and supporting algorithms to place spare cores while minimizing the cost of migration.

4.2 The Many-Core Array

The many-core architecture is a square lattice arrangement of R rows and C columns of homogeneous computing nodes. Each *node* consists of a *router* that communicates with its nearest orthogonal neighbours and a *core* attached to the router. Communication between two nodes is via a *link*. Links are required to show the connections between nodes, however the details of the communication links between nodes is not important when considering the placement of spare cores. Communication links will be considered in more detail in the next chapter.

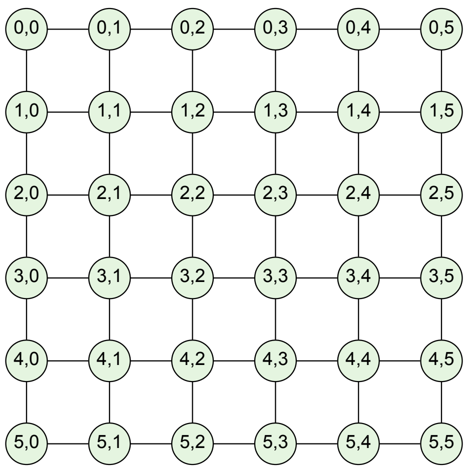


Figure 4.1 – A 6×6 many-core array showing the coordinates of each node.

The many-core array $\mathcal{A}$ is represented as a tuple of the set $\mathcal{V}_a$ of nodes and the set $\mathcal{E}_a$ of links:

$$\mathcal{A} \coloneqq (\mathcal{V}_a, \mathcal{E}_a) \tag{4.1}$$

Given that there are R_a rows and C_a columns, the number of nodes and links are defined as V_a and E_a:

$$\begin{aligned} |\mathcal{V}_a| &= V_a \\ &= R_a C_a \end{aligned} \tag{4.2}$$

$$\begin{aligned} |\mathcal{E}_a| &= E_a \\ &= R_a(C_a - 1) + (R_a - 1)C_a \\ &= 2R_a C_a - R_a - C_a \end{aligned} \tag{4.3}$$

The set of nodes is defined as:

$$\mathcal{V}_a = \{v_1, \ldots, v_m \mid m = V_a\} \tag{4.4}$$

With a node defined as an ordered tuple consisting of the row and column coordinates of the node within the many-core array, which are also referred to as its location:

$$\begin{aligned} v_n &= (r, c) \\ &= loc \end{aligned} \tag{4.5}$$

Where:

$$loc = (r, c) \tag{4.6}$$

and:

r	Is the row coordinate of node $v_n \mid 0 \leq r < R_a$.
c	Is the column coordinate of node $v_n \mid 0 \leq c < C_a$.

Row and column coordinates are defined above as beginning at 0 and location $(0, 0)$ is arbitrarily assigned to the top-left hand core of the many-core array and location $(R_a - 1, C_a - 1)$ refers to the bottom-right hand core of the many-core array.

The set of links is defined as:

$$\mathcal{E}_a = \{e_1, \ldots, e_m \mid m = E_a\} \tag{4.7}$$

A link is defined as a set of ordered tuples consisting of a source node and a target node:

$$e_n = (s, t) \tag{4.8}$$

Where:

e_n	is a link from the set $\mathcal{E}_a \mid 1 \leq n \leq E_a$.
s	is the source node of link e_n from the set $\mathcal{V}_a \mid 1 \leq s \leq V_a$.
t	is the target node of link e_n from the set $\mathcal{V}_a \mid 1 \leq t \leq V_a$.

Since the definition of the node includes its location, the source and target nodes of a link also specify the locations of each end of the link. Figure 4.1 illustrates a 6×6 many-core array showing the coordinates of the location of each node and the links between nodes.

4.3 Application Process Graph

The Graceful Project concept, that this work uses, requires that an application program is divided into a number of processes that can run independently with data flowing between the processes. The resulting program model is an *Application Process Graph* which is an

Directed Acyclic Graph, where the processes are represented by nodes and data flows between processes are represented by edges.

An example of an application process graph with 26 processes application processes is illustrated in Figure 4.2.

Since the application process graph is a directed graph, each edge has an arrow head which specifies in which direction the data flows between the processes. Each node can have data receiving *inbound* edges which are edges where the arrowhead of the edge is attached to the node, and data sending *outbound* edges which are edges where the tail of the edge is attached to the node. A node must have at least one edge which can be either an inbound or an outbound edge; a node with no edges would be disconnected from the application process graph so would not be part of the graph. For an application process graph a *source node* is a node that has no inbound edges while a *sink node* is a node that has no outbound edges. In this example there is a single source node and single sink node.

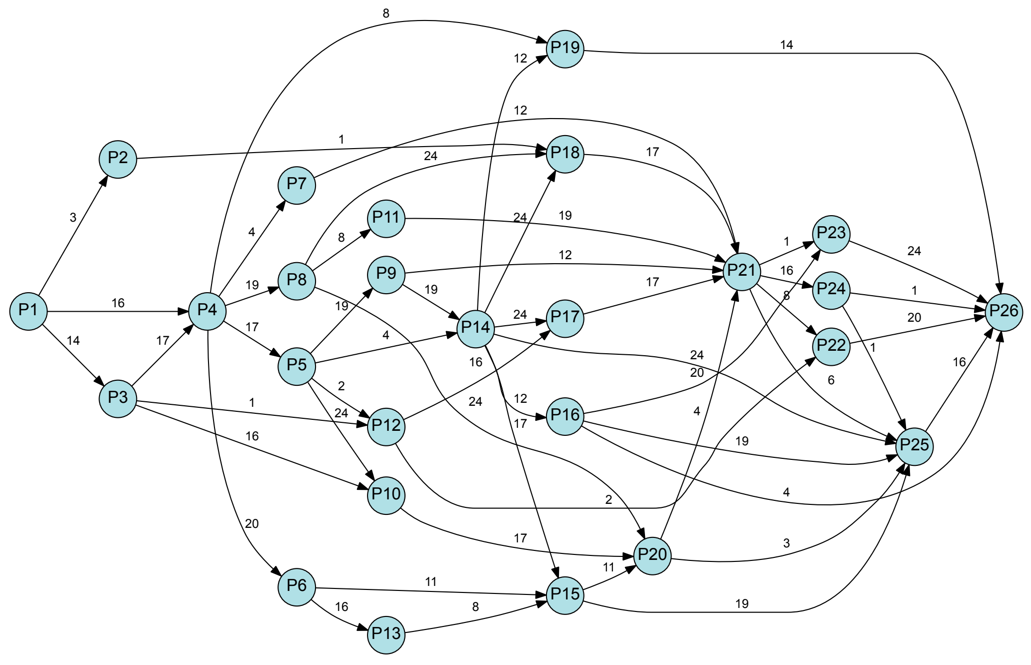


Figure 4.2 – An application process graph for a 26 node APG in a 6x6 Many-Core Array

Nodes are arbitrarily arranged on the graph so that source nodes are on the extreme left and sink nodes are on the extreme right, and data flows between nodes from left to right. Nodes are grouped by *ranks* from left to right, each rank containing nodes which are the same distance from a source node in terms of the largest number of nodes from the source node to the nodes in the rank. Ranks are used to improve the aesthetic quality of the visualisation of the graph but otherwise have no effect on the mapping process from the application process graph to the many-core array. The graph in in Figure 4.2 has 12 ranks.

The graph generator (see Appendix A) assigns each node in a graph a unique identifier, beginning with "P" followed by a sequential identifier, starting at 1, to ensure that each node has a unique identifier representing the specific process which will be used throughout the model.

Each edge has a *source node* and *target node* and is annotated with an integer value that represents the traffic volume as a percentage of the bandwidth of the hardware link. It is allowed for the traffic volume to be greater than 100, in which case at least two hardware links will be required to carry the traffic between the processes.

The graph $\mathcal{G}$ is represented as a tuple of the set $\mathcal{V}_g$ of nodes and the set $\mathcal{E}_g$ of edges:

$$\mathcal{G} := (\mathcal{V}_g, \mathcal{E}_g) \tag{4.9}$$

The number of nodes and edges are defined as V_g and E_g:

$$V_g = |\mathcal{V}_g| \tag{4.10}$$

$$E_g = |\mathcal{E}_g| \tag{4.11}$$

The set of nodes are defined as:

$$\mathcal{V}_g = \{v_1, \ldots, v_n \mid 1 \leq n \leq V_g\} \tag{4.12}$$

Where each node v_n has a single value, the *process name* that is the string 'Pn' which is the concatenation of the letter 'P' and the value of n, that uniquely identifies the APG process.

The set of edges are defined as:

$$\mathcal{E}_g = \{e_1, \ldots, e_n \mid 1 \leq n \leq E_g\} \tag{4.13}$$

A node is a set of ordered 3-tuples consisting of a source node, a target node and an edge value representing the volume of traffic that will be generated by the source for transmission to the target:

$$e_n = (s, t, d) \tag{4.14}$$

Where:

e_n	is an edge from the set $\mathcal{E}_g \mid 1 \leq n \leq E_g$.
s	is the source node of edge e_n from the set $\mathcal{V}_g \mid 1 \leq s \leq V_g$.
t	is the target node of edge e_n from the set $\mathcal{V}_g \mid 1 \leq t \leq V_g$.
d	is an integer representing the traffic volume on edge e_n as a percentage of the bandwidth of links in the many-core array.

The traffic volume d, will be used in the calculation of the power metric described in Section 6.5 *Network Power Metric and Objective*.

4.4 Process Map

A *process map* or *mapping* is a data structure that, for each processing node of the application process graph, gives the location of the core in the array that will run the process. Cores that are running a process are given a task name that is the process name from the application process graph. Many-core array cores that are not allocated a process are spare cores and are considered to be idle so are given a task name of 'i', while failed cores are given a task name of 'f'. Figure 4.3 illustrates a 6×6 many-core array.

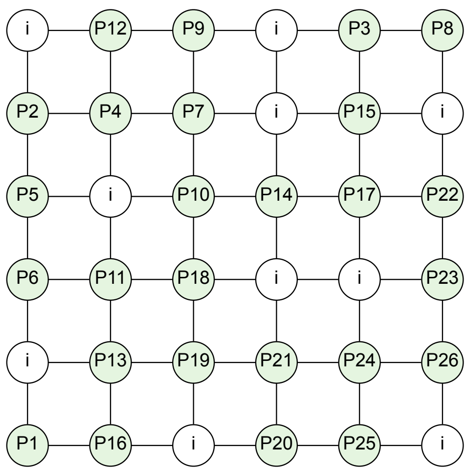


Figure 4.3 – A Process Map for a 26 node APG in a 6x6 Many-Core Array

Figure shows a process map for the application process graph in Figure 4.2 mapped to the many-core array of Figure 4.1

The process map has the same dimensions as the many-core array, given as R rows and C columns defined in Section 4.2.

The process map $\mathcal{M}$ is represented as a set of nodes $\mathcal{V}_m$:

$$\mathcal{M} := \mathcal{V}_m \tag{4.15}$$

The number of nodes is defined as V_m and is the same as V_a:

$$\begin{aligned} V_m &= |\mathcal{V}_m| \\ &= V_a \\ &= R_m C_m \end{aligned} \tag{4.16}$$

Given that there are R_m rows and C_m columns, the set of nodes are defined as:

$$\mathcal{V}_m = \{v_1, \ldots, v_n \mid 1 \leq n \leq V_m\} \tag{4.17}$$

A node is a set of ordered tuples consisting of a location which is the coordinates of the corresponding node in the many-core array and a process name:

$$\begin{aligned} v_m &= (r, c, p) \\ &= (loc, p) \end{aligned} \tag{4.18}$$

Where:

$$loc = (r, c) \tag{4.19}$$

and:

v_m	Is a node of the set $\mathcal{V}_m$.
r	Is the row coordinate of the node $v_m \mid 0 \leq r < R_m$, where row 0 is the top row.
c	Is the column coordinate of the node $v_m \mid 0 \leq c < C_m$, where column 0 is the left most column.
p	Is the process name of the APG process mapped to the many-core array node at location (r, c) or, if no process is mapped to the many-core array node, the value i representing a spare core $\mid p \in \mathcal{P}_g \cup \{i\}$.

The location coordinates are illustrated in Figure 4.1.

4.5 Metrics and Objectives

Metrics are measurements of fundamental properties of the system that is being studied. Here we are interested in measuring properties of a mapping of an application process graph onto a many-core array. Objectives are used by a search algorithm to make a comparative measure of fitness between mappings and often use a metric in the calculation of the value of the objective.

For example, the *rectilinear distance* between two nodes in a many-core array is the sum of the number of horizontal and vertical edges between the two nodes; this is a metric. A corresponding objective could be to minimize the sum of the rectilinear distance between all pairs of communicating cores. In this case, the metric measures the actual distance for a single pair of communicating cores, while the objectives uses the metric to calculate a single value for all pairs of communicating cores in the mapping.

This thesis will present four objectives along with their supporting metrics. These are:

- Core fault tolerance

- Link fault tolerance
- Network power
- Excess traffic

This is only a selection of possible objectives for optimization of mappings for many-core arrays selected to illustrate the work of this thesis.

The core fault tolerance objective is the subject of this chapter. The other objectives will be presented in subsequent chapters.

4.6 Core Fault Tolerance Metric and Objective

Metrics are measurements of fundamental properties of the system that is being studied. Here we are interested in measuring properties of a mapping of an application process graph onto a many-core array. Objectives are used by a search algorithm to make a comparative measure of fitness between mappings and often use a metric in the calculation of the value of the objective.

For example, the *rectilinear distance* between two nodes in a many-core array is the sum of the number of horizontal and vertical edges between the two nodes; this is a metric. A corresponding objective could be to minimize the sum of the rectilinear distance between all pairs of communicating cores. In this case, the metric measures the actual distance for a single pair of communicating cores, while the objectives uses the metric to calculate a single value for all pairs of communicating cores in the mapping.

4.6.1 Core Fault Tolerance

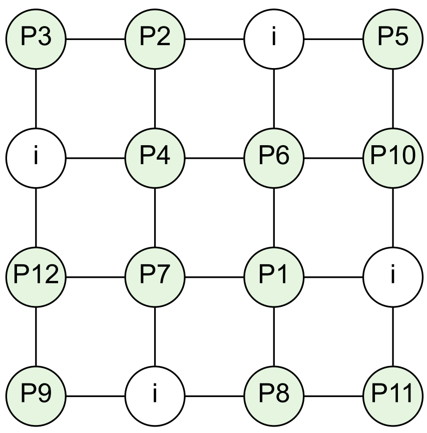
Problem Description
This section develops the metric for measuring the distance between cores and the objective, which makes use of the metric, that is used by the evolutionary algorithm.

Given a fault free many-core array with R rows and C columns and an application process graph with V_a processes where $V_a < RC$, arrange the spare cores to minimize the cost of task migration in the event of the failure of a processing core.

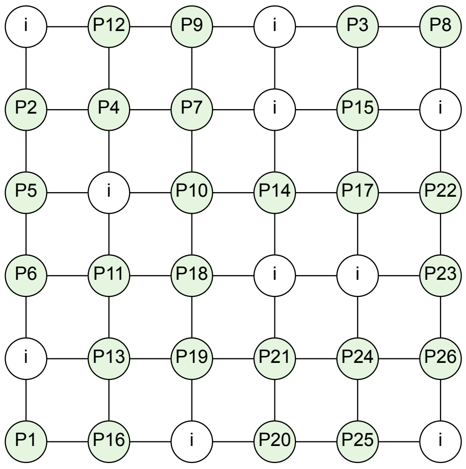
The cost of task migration will be defined explicitly in Section 4.5.

4.6.2 Distance to Nearest Idle Core Metric

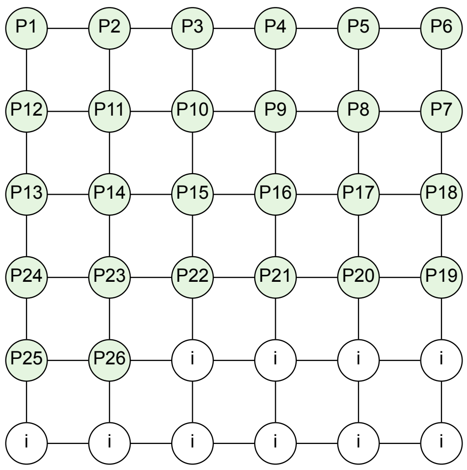
The distance to nearest idle core metric will be a measure of the distance between an individual processing core and its nearest idle core. As stated in Section 4.2, given an array of R rows and C columns, each node's location can be given by a tuple, (r, c) representing the row r and column c of the node relative to the top left-hand corner of the array. The



(a) Distribution on a 4x4 Array



(b) Distribution on a 6x6 Array



(c) Mapping on a 6x6 array with weak core fault tolerance

Figure 4.4 – Examples of Distribution of Spare Cores

distance to nearest idle core metric, $Mdnic_{(r,c)}$, is the distance between a processing core at location $p_{loc} = (p_r, p_c)$ and its nearest idle core at location $i_{loc} = (i_r, i_c)$ and is defined as the rectilinear distance between the location of the processing core and the location of the idle core, given by Equation 4.20.

$$Mdnic_{(r,c)} = |(p_r - i_r)| + |(p_c - i_c)| \tag{4.20}$$

Where:

$Mdnic_{(r,c)}$	=	The distance to nearest idle core metric for the process located at (r, c).
p_r	=	The row of the location of the process.
p_c	=	The column of the location of the process.
i_r	=	The row of the location of the nearest idle core.
i_c	=	The column of the location of the nearest idle core

The minimum value of the nearest idle core metric is 1 which is the distance when a processing core is adjacent to an idle core.

4.6.3 Core Fault Tolerance Objective

The fault tolerance objective is to minimize the sum of the distances between each processing core and its nearest idle core. When an idle core is adjacent to a processing core, i.e. one step away, the fault tolerance objective value is defined as zero. When t steps are required to reach the nearest idle core the fault tolerance objective value will be $t - 1$ (In other words, to obtain an adjacent idle core objective value of zero the metric value of each process-idle core pair is reduced by one). This is an arbitrary choice made to ensure that the lowest possible fault tolerance value for any array size is zero.

For example, Figure 4.4 (a) and (b) illustrate respectively, for 4×4 and 6×6 arrays, arrangements of idle cores among processing cores where every processing core is adjacent to at least one idle core, while Figure 4.4c illustrates a mapping which has weak fault tolerance due to some processing cores having no adjacent idle core. The mapping 4.4a and 4.4b, are assigned an objective value of zero, by the objective function, as no other distribution of idle cores is considered to be any better for the purposes of core fault tolerance.

A fault tolerance cost of zero for the task map as a whole indicates that each processing core is adjacent to at least one idle core. A processing core that has more than one adjacent idle core is not regarded as having any additional benefit from the additional adjacent idle cores. The core fault tolerance objective $Jcore$ is given as:

$$Jcore = \sum_{r=0}^{R-1}\sum_{c=0}^{C-1}\begin{cases} Mdnic_{(r,c)} - 1, & \text{if core}(r,c) \text{ is a processing core} \\ 0, & \text{otherwise} \end{cases} \tag{4.21}$$

4.7 Nearest Spare Core Search

The optimization process of the core fault tolerance objective for a mapping will require searching for the nearest idle core to each processing core in each candidate mapping examined by the search algorithm. For an array size of 6×6 and 28 processes and 1000 generations with a population of 100 then the number of nearest core searches during an evolutionary run will be $28 \times 1000 \times 100 = 2.8$ million searches where each search could make as many of $28 \times (6 \times 6 - 1) = 1008$ comparisons.

Searching for the nearest core with a specific property is therefore worthy of examination to develop an efficient algorithm. This section will discuss possible search methods.

4.7.1 Problem Statement

Given a processing node in a square lattice, develop an algorithm for efficiently finding the nearest spare core the the processing core.

4.7.2 Method 1 - Exhaustive Search with Early Termination

An exhaustive search takes a processing core, searches through all the remaining cores for idle cores and calculates the distance to each idle core, retaining the location with the minimum distance. The search ends when an idle core with a distance of one has been found or all cores have been examined. Using this method it is not possible to know if the nearest idle core has been found until all cores have been searched unless an idle core with a distance of one has been found, which is the minimum possible distance.

If it is assumed that the exhaustive search algorithm would process cores from the beginning to the end of the data structure used to represent the cores then, in general, the algorithm gives no preference to cores it is examining based on the distance of cores from the processing core. If there are c cores in the array of which p cores are processing cores, then $p * (c - 1)$ core pairs will be examined unless there is an idle core adjacent to the processing core. The number of comparisons will increase in proportion to increases of both c and p.

4.7.3 Method 2 - Concentric Diamond Search

A more efficient search strategy to find the nearest idle core to an processing core is to examine all cores at a distance of one step away from the processing core, followed by all cores with a distance of two steps and so on until all cores have been examined. In a two dimensional mesh all cores that are a distance of n steps from the processing core form a diamond shape centred around the processing core. Starting from the processing core, the search proceeds by examining each core in a succession of concentric diamonds of increasing size until either an idle core is found or all cores have been examined. The worst case scenario is where the processing core is at one corner of an array and the only idle core is in the opposite corner.

When discussing the diamond search pattern it is convenient to refer to the number of steps from the processing core to the cores of a diamond as the radius of the diamond. The coordinates of the cores in a diamond of radius n relative to an processing core can be calculated using only the radius n.

To determine the coordinates of all the cores in a diamond the mathematical concept of a partition of an integer can be employed. A partition of an integer n is a set of integers whose sum is n.

For example, if $n = 3$ then the partitions are {3}, {2,1} and {1,1,1}.

The number of integers in a partition are described as parts; so {3} is a partition of one part, {2,1} is a partition of two parts and {1,1,1} is a partition of 3 parts. For this discussion it is convenient to regard the partition of {3} as having two parts written as {3,0}.

If the order is important, then the partition is described as a composition. The compositions for $n = 3$ are {3,0}, {2,1}, {1,1,1}, {1,2} and {0,3}. In this case we are working with a two dimensional array so we are only interested in the two part compositions. Note, that this method can easily be extended to d-dimensional lattices by using compositions with d parts.

The two part compositions of 3 are {3,0}, {2,1}, {1,2} and {0,3} which can be interpreted as coordinates of the side of the diamond that is in quadrant I of the Cartesian plane. The Cartesian plane can be divided into four quadrants referred to by the Roman numerals I, II, III and IV starting with quadrant I where x and y coordinates are positive and then working anti-clockwise, where quadrant II has positive y and negative x, quadrant III has negative y and negative x and finally quadrant IV which has negative y and positive x.

Given the two part compositions of 3 that are located in quadrant I of the Cartesian plane, the coordinates of the points on the diamond in the other three quadrants of the Cartesian plane can be obtained from the permutations of positive and negative values of the parts of each composition. The negative and positive permutations for $n = 3$, treating zero as a positive value, give quadrant I coordinates of (0,3), (1,2) and (2,1), (3,0), quadrant II coordinates of (-1,2), (-2,1) and (-3,0) quadrant III coordinates of (-2,-1) and (-1,-2) and quadrant IV coordinates of(0,-3), (1,-2) and (2,-1).

In general, for a diamond of radius n there are $4n$ points which are a distance of n steps from the core at the centre of the diamond.

If each of the twelve coordinates are added, in turn, to the coordinates of the processing core then the coordinates of all cores on the diamond of radius 3 are obtained as shown in Figure 4.5.

The 2 part compositions of a given integer can easily be obtained by use of a simple counting loop. The permutations of negative and positive values of each compositions are also simple to derive. An algorithm can therefore easily calculate the relative coordinates of all points on a diamond with radius n.

Since the above obtains coordinates of points in a 2-dimensional space it is easy to apply to a 2-dimensional data structure representation. The relative coordinates can also be applied to a 1-dimensional data structure representation of the 2-dimensional space with appropriate mapping calculations.

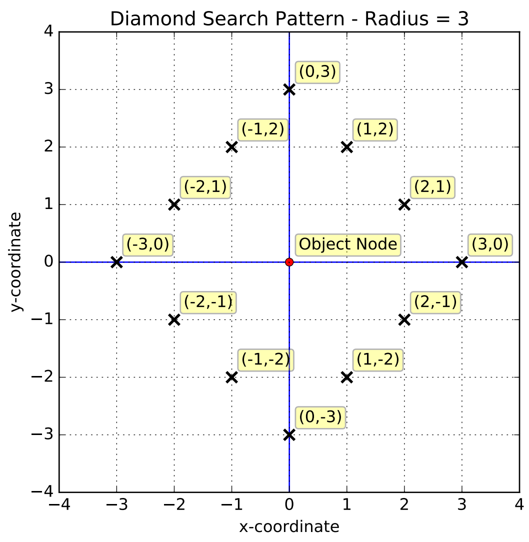


Figure 4.5 – **Diamond Search Pattern - Radius = 3**

In the worst case scenario where the processing core is at one corner of an $n \times n$ array and the only idle core is in the opposite corner then $2n^2 + 2n$ candidate positions would be calculated of which only $n^2 - 1$ are valid core coordinates that have to be examined.

The search ends when an idle core is identified or all cores have been examined.

4.8 Many-Core Evolutionary Algorithm

Previous sections (Section 4.2 describing the many-core model and Section 4.5 describing the metrics) are applicable to any search method suitable for exploring the solution space of process maps.

The solution space for mappings using a single objective of core fault tolerance consists of all the possible arrangements of V_g processes in an array with V_a nodes. The size of the search space is the given by the number of permutations of V_g processes within an array with V_a nodes, given by Equation 4.8:

$$P(V_a, V_g) \equiv \frac{V_a!}{(V_a - V_g)!} \tag{4.22}$$

An exhaustive search of the arrangements of processes in an array falls within the category of NP-hard problems, so for even a moderately sized problem of 28 processes within a 6×6 array, the search cannot be completed using an exhaustive search.

At this point in the research core fault tolerance is the only objective, however, the remainder of the research will use multiple objectives required the exploration of multi-dimensional solution spaces.

Evolutionary algorithms have proved to be an effective tool for searching solution spaces of a size that cannot be exhaustively searched in polynomial time. The process allocation problem is NP-hard with the complexity increasing as a factorial of the size of the array [173, 174]. An evolutionary algorithm is, therefore, a suitable tool for exploring mappings of application process graph processes to a many-core array.

This chapter describes the evolutionary algorithm that has been designed to search the solution space for many-core array mappings.

The evolutionary algorithm is composed of a collection of processes that run in a cycle to manipulate the population of one generation to produce the population of the next generation. Each of the processes will be explored individually as they are independent of each other. With the exception of the process that generates the initial population, each process takes as input a population which it manipulates to create a new population as its output.

The individual elements of the evolutionary algorithm, listed below, will be discussed in the remaining sections of this chapter.

- The Evolutionary Cycle
- Phenome and Genome
- Population Dynamics

4.8.1 Evolutionary Cycle

An evolutionary algorithm is a process by which a population is progressively changed from one generation to the next to improve the fitness of individuals within the population. The section explains how changes to the population, from one generation to the next, are influenced through a collection of parameters.

4.8.1.1 The Cycle

The evolutionary cycle is implemented as a series of populations, summarised in Table 4.1.

The evolutionary algorithm is implemented as an *evolutionary cycle*, illustrated in Figure 4.6, which show the populations in the cycle and the processes that transform one population into another. At the end of each evolutionary cycle, or generation, a new primary population is produced which is also the starting point for the next evolutionary cycle. The transformation processes and termination of the evolutionary cycle are controlled by the evolutionary algorithm parameters.

Table 4.1 – Evolutionary Algorithm Populations

Population	Description
Initial	The initial population which is provided to the evolutionary cycle as a starting point for evolution.
Generation Zero	The first primary population created by evaluating and sorting the initial population and is the starting point for evolution.
Primary	The sorted population that is the product of each evolutionary cycle (and the starting point for the next cycle).
Intermediate	The population created through cloning and genetic manipulation of the primary population of the previous evolutionary cycle.
Fitness Evaluated	The intermediate population after the objectives have been evaluated.

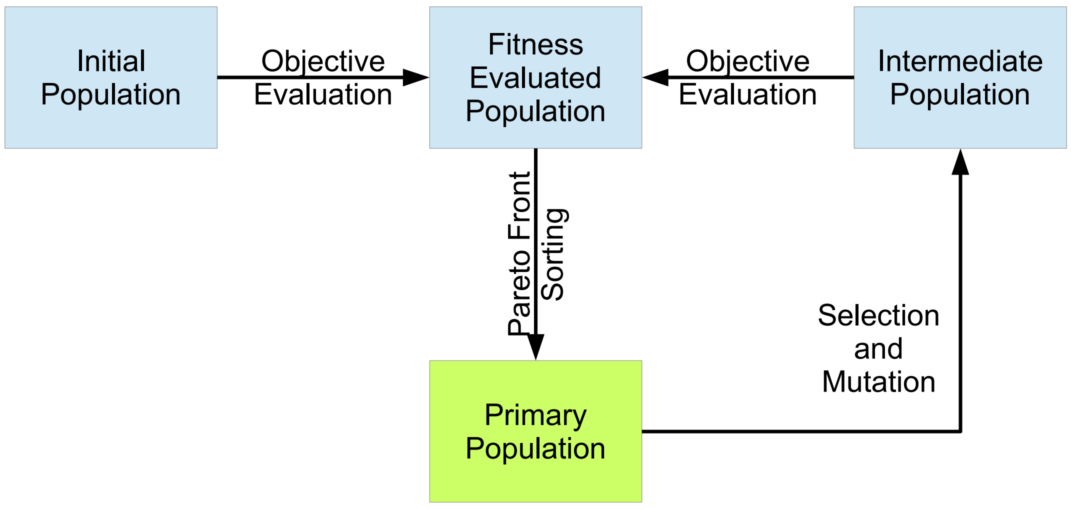


Figure 4.6 – Evolutionary Algorithm Cycle

4.8.1.2 Initial Population

The initial population is composed of a combination of engineered mappings (discussed in Subsection 4.8.3) and random mappings. The initial population is located in the top left of Figure 4.6. Once the individuals of the initial population have been created their metrics are evaluated, producing a fitness evaluated population and sorted to produce the *generation zero* population, the first primary population. The size of the initial population is governed by the population size parameter, P_z.

4.8.1.3 Generation Zero Population

The generation zero population is the result of the evaluation and sorting of the initial population and is the first primary population which is used as the starting point of the evolutionary process. The name generation zero is given to this population to distinguish it from all other primary populations that are created through the evolutionary process.

4.8.1.4 Primary Population

The *primary population* is the sorted population produced at the end of each evolutionary cycle and is also the population that is produced on completion of the evolutionary algorithm. The primary population is the population that is used as the starting population for each evolutionary cycle. In Figure 4.6 the primary population is represented as the green box at the bottom of the diagram. Outside of this section the primary population will normally be referred to simply as *the population*.

The size of the primary population, P_z is set to 10. On each iteration the primary population is replaced by a new population of the same size.

4.8.1.5 Intermediate Population

The intermediate population is created by selection and mutation of individuals from the primary population. The creation of the individuals in the intermediate population, illustrated in Figure 4.7, is controlled by the following parameters, choose to create a new population with the same size of the primary population (P_z = 10).

Elite Individuals
The elite parameter, P_e set to 2, is used to specify the minimum number of fittest individuals of the input population that are copied directly to the new intermediate population.

Descendants
The number of individuals specified by the *Parents* parameter, P_p set at 4, are selected from the input population. The *Descendants* parameter, P_d set to 4, determines how many times each parent is copied and mutated before being added to the intermediate population.

The size of the intermediate population is determined by the evolution parameters using Equation 4.23, which limits the intermediate population to be the same size as the primary population.

$$Q_z = P_e + (P_p \times P_d) \tag{4.23}$$

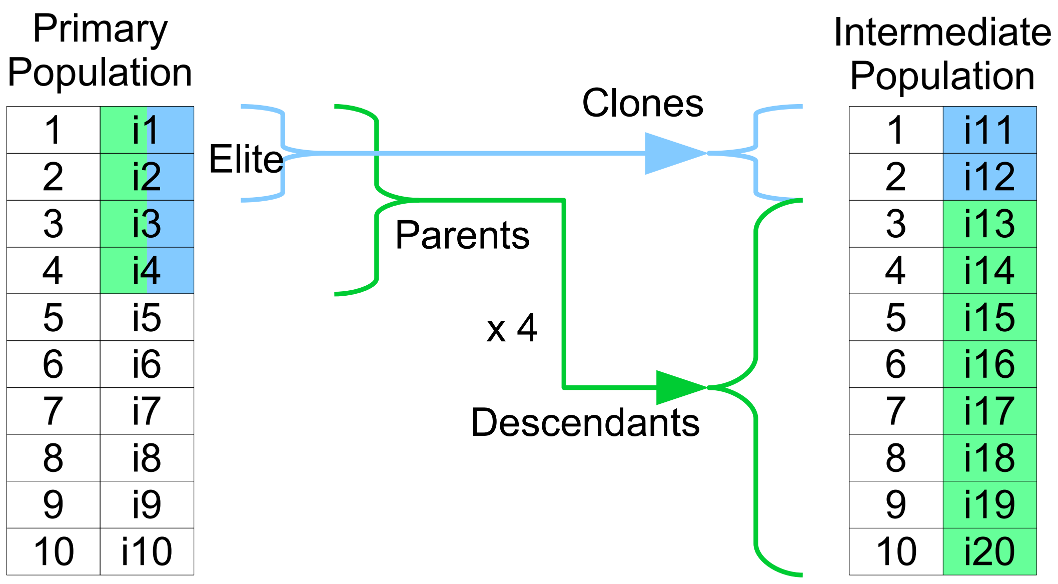


Figure 4.7 – Population Selection and Mutation, using 2 elite individuals, 4 parents mutated twice each to create a population of size 10.

4.8.1.6 Fitness Evaluated Population

Once the intermediate population has been created, the objectives for each individual need to be evaluated in preparation for sorting the population. The *fitness evaluated population* contains the same individuals as the intermediate population with the evaluated objectives added to each individual.

4.8.1.7 Population Evolution Parameters

The size of the primary population and the intermediate population are determined by the properties and evolution parameters given in Table 4.2.

Table 4.2 – Population Properties and Parameters

Parameter	Description
P	The primary population
Q	The intermediate population
P_z	Primary population size
Q_z	Intermediate population size
P_e	Number of elite Individuals
P_p	Number of parents
P_d	Number of descendants

4.8.2 Phenome and Genome

The *phenome* is a representation of where processes and idle cores are placed within the environment of the many-core model. The *genome* is a representation used to carry out manipulation to produce new individuals from existing individuals. The genomic representation can either be a direct representation of the phenome or an indirect representation that requires a mapping, through the process of expression, to produce the phenome. Each representation has associated benefits and disadvantages.

A particular genomic representation will have a its own *correlation profile* with respect to the phenomic representation. *Correlation* describes how a mutation in the genome is exhibited in the phenome. A good correlation is one where mutations in the genome will be exhibited by corresponding changes to the phenome. A poor correlation is one where mutations in the genome cause the phenome to exhibit changes that do not correspond well with the changes in the genome, for example, where a single change to the genome results in multiple and chaotic changes to the phenome. A good correlation makes the changes to the phenome via the genome more controllable.

An advantage of using a coded representation is that random generation of genomes and the genetic manipulation of genomes can be much simpler than manipulating the phenome directly. In contrast the expression of a genome using a coded representation requires a mapping to be carried out to express the phenome which needs to enforce any constraints required by the phenome.

There are three properties of uniqueness that must be present in a phenome for it to represent a valid mapping of an application process graph to a many-core array. The first property is that each process is represented exactly once in the phenome; the second property is that at most one phene is mapped to each node of the many-core array; and the third property is that processes must only be mapped to non-faulty nodes. These requirements must be enforced when creating phenomes either directly or through the expression of a coded genomic representation.

4.8.2.1 Phenome

In the same way that a genome consists of a collection of genes, the phenome can be modelled as a collection of phenes. A natural and programmatically simple method to represent a phenome is to use a two dimensional array of phenes where the location of the phene in the array is analogous to the location of the node in the many-core array so that it is not necessary to specifically code the location of the node within the phene.

Each location in the array contains a tuple representing the status and name of the node. The valid values for the status of the node is given in Table 4.3

Irrespective of the genomic representation the fitness is calculated using the phenome after

Table 4.3 – Node Status Values

Status	Description	Name
i	Idle node	i
f	Failed node	f
p	Processing node	Process name

expression from the genome to the phenome has taken place.

Table 4.4 lists the properties of phenome and genomes.

Table 4.4 – Phenome & Genome Properties

Parameter	Description
P	The phenome composed of the set phenes $\{p_1 \dots p_{\mathcal{P}}\}$
$\mathcal{P}$	The number of phenes $\lvert P \rvert$ in the phenome
P_{rows}	The number of rows of nodes in the phenome
P_{cols}	The number of columns of nodes in the phenome
$p_{(r,c)}$	The phene at location (r, c) in the phenome
r	The row where phene $p_{(r,c)}$ is located
c	The column where phene $p_{(r,c)}$ is located
G	The genome composed of the set of genes $\{g_1 \dots g_n\}$
g_n	The n^{th} gene of the genome
A	The set of alleles of a genome

4.8.2.2 Genome - Phenomic Representation

In order to keep the evolutionary algorithm simple, the genome will be a direct representation of the phenome. The next chapter will compare a variety of genomic representations to establish which representation gives the best overall performance.

A genome that uses a phenomic representation has a one-to-one correspondence between the genes in the genome and the phenes in the phenome which, in this case, is a two dimensional array of genes. The expression of the phenome from the genome is a trivial one-to-one mapping from the genes to the phenes.

The genetic alphabet consists of $P_z + 1$ alleles, one for each processing node of the application process graph and one to represent the idle condition. Each gene can be any one of the alleles, but the genome as a whole must conform to the constraints for a valid phenome defined in Subsection 4.8.2.

Permutation

For a genome using a direct phenomic representation, the constraints defined in Subsection 4.8.2 need to be enforced by the genetic operators used to manipulate the genome. Permutation involves swapping two randomly selected genes one of which must be a process and second must be either a process or an idle core, which excludes the possibility that either gene is in the failed state. The constraint that at least one gene must be a process ensures that time is not wasted on exchanging two idle cores, which has no effect on the genome. Assuming that the parent gene is valid, swapping a pair of genes in the manner described will guarantee that the new genome will be valid.

Mutation
Mutation is problematic because the genome can only have one copy of each allele. Mutation, by definition, will transform one allele into another resulting in a genome that will be missing one allele while having two copies of another. A repair mechanism would be required to remove duplicates and add missing alleles. The complication of adding a repair mechanism makes mutation unsuitable for a phenomic representation.

Crossover
Crossover has similar problems to mutation, although via a different route. If two parents are chosen and portions of each parent are mixed to produce a genome of the same length then it cannot be guaranteed that all the alleles will be unique. A repair mechanism would be required to remove duplicates and add missing alleles. This makes mutation unsuitable for a phenomic representation.

Correlation
As there is a one-to-one mapping between the genome and phenome, every change to the genome will be mirrored in the phenome producing a perfect correlation between them.

Random Generation of Genomes
Random generation of genomes must enforce the constraints defined in Subsection 4.8.2. This is achieved by first creating a genome which is filled with idle genes while ensuring that any failures in the hardware map are reflected in the genome. The processes are retrieved from the application process graph one at a time and replace a randomly selected idle core. Taking processes from the application process graph ensures that each process is represented in the genome exactly once. Ensuring a randomly selected gene is in the idle state ensures that only one process is mapped to each node in the underlying hardware map.

4.8.3 Engineered Mappings

Before evolution begins an initial population is required. An obvious method of producing individuals for generation zero is to randomly map processing nodes to nodes in the many-core array. While this may be an acceptable method in many situations, the aim here is to improve upon random generation, motivated by the desire to improve the quality of mappings obtained for a given computational effort. This section will present a collection of deterministic algorithms to generate *engineered* mappings for inclusion in generation zero. In general, generation zero will be a mixture of engineered solutions and randomly generated solutions.

There are two properties of mappings that the engineered mappings will attempt to exploit. The first is that, to attain good core fault tolerance, the processing nodes need to be evenly distributed across the many-core array; the second is that the power metric, introduced in the next chapter, is minimized when pairs of communicating cores are close to each other. As noted in Section 4.3, application process graph nodes are numbered one rank at a time, and since data flows from the low order ranks to high order ranks, preserving the order of the nodes may help keep pairs of communicating cores close together therefore producing mappings with lower than average power metrics.

The engineered solutions attempt to capitalise on these two observations by using algorithms that control the distribution of processing nodes and the order in which array nodes are selected for allocation of a process.

4.8.3.1 Random Placement

To create a random mapping, each processing core of the application process graph is taken in turn and a random number generator is used to produce coordinates for an array node. If the array node has an idle status then the processing node is mapped to the array node. If the status of the array node is not idle then new coordinates are generated until an array node one is found that is idle.

No specific knowledge of the application graph is required as each process can be placed without reference to the other processes. The random placement of processes means that pairs of communicating cores are as likely to be on opposite sides of the network as they are likely to be next to each other. The average distance between v is expected to be higher than the engineered solutions.

It is assumed throughout this section that there are at least as many idle cores in the many-core array as there are application processes in the application process graph.

4.8.3.2 Engineered Mapping Concept

The concept for producing engineered mappings is to first determine a *process distribution pattern* of processing nodes using a one-dimensional list, and then fill the nodes of the array from the elements of the list using an *array node ordering*. The following sections will first describe the the distribution pattern and the the array node ordering which when used together produce a collection of engineered mappings.

4.8.3.3 Process Distribution Pattern

The process distribution pattern determines how the application process graph processing nodes are distributed along a one dimensional list.

The algorithm starts with a one dimensional *node distribution list* that has one element representing each node in the many-core array with each element initially representing an idle core as shown in Figure 4.8a. Then it uses a distribution method to place each processing node from the application process graph into the node list to produce a distribution pattern as shown in Figure 4.8b and Figure 4.8c.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

(a) All idle Cores

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	i	i	i	i

(b) 12 Clustered processes

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1	P2	P3	i	P4	P5	P6	i	P7	P8	P9	i	P10	P11	P12	i

(c) 12 Distributed Processes

Figure 4.8 – String Length of 16 Cores

Distribution
The distribution method can be either *clustered* or *distributed*:

Clustered means that each processing node is placed in the node list in consecutive positions with no gaps until the list of application process graph processing nodes has been exhausted. The result is shown in Figure 4.8b.

To distribute processing nodes evenly across the many-core array a calculation can be made using the number of available idle cores in the many-core array and the number of processing nodes from the application process graph to determine a *cluster size* of consecutive list nodes that have processing nodes placed in them and the number of idle cores between each cluster. The result is shown in Figure 4.8c.

4.8.3.4 Array Node Ordering

The placement algorithm will control the order in which nodes in the array are visited for the purpose of mapping processing nodes from the node distribution list. Throughout this section the default first node to be considered is taken to be the top left-hand cornet of the array number 1 in Figure 4.9b and Figure 4.10.

There are four parameters, listed in Table 4.5, designed to determine the order in which nodes in the many-core array are visited. The parameters are used in combination to produce a variety of orderings of the array nodes.

Table 4.5 – Array Node Ordering Parameters

Parameter	Description
Traversal	Defines if nodes are scanned horizontally or diagonally
Transition	Method of moving from one scanned line to the next
Direction	Direction in which the scan starts
Orientation	Positioning of first node

Orientation

Orientation defines the direction of lines of array nodes for scanning. The orientation is either horizontal, as illustrated in Figure 4.8, or diagonal, as illustrated in Figure 4.10.

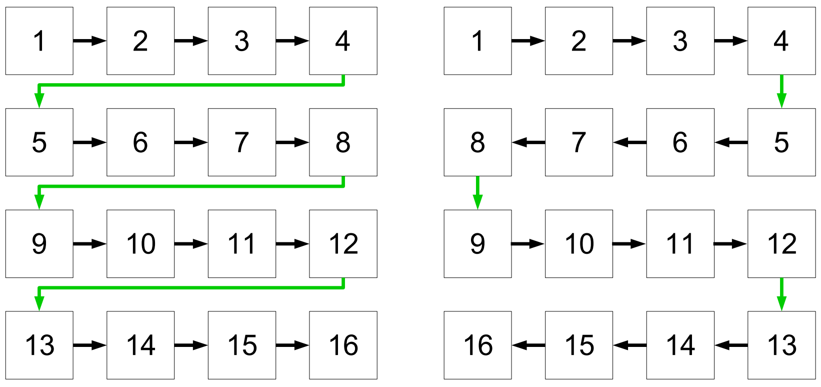
Transition

The ordering of array nodes is based on scanning lines of nodes, one line at a time. The *transition* parameter determines how the scan proceeds from one line to the next when the scan reaches the end of a line. Transition and can be either *raster* or *snake*.

Raster

The term *raster* is used here to refer to the process of scanning lines of nodes, one line at a time and scanning each line in the same direction, for example from left to right. The array in Figure 4.9a illustrates a raster scan with horizontal traversal, while Figure 4.10a illustrates a raster scan with a diagonal traversal.

If nodes are numbered in the order in which they are visited, then in each line neighbouring nodes will have consecutive numbers. However, there will be a discontinuity at the end of each line because the node with the next consecutive number is at the beginning of the next line.



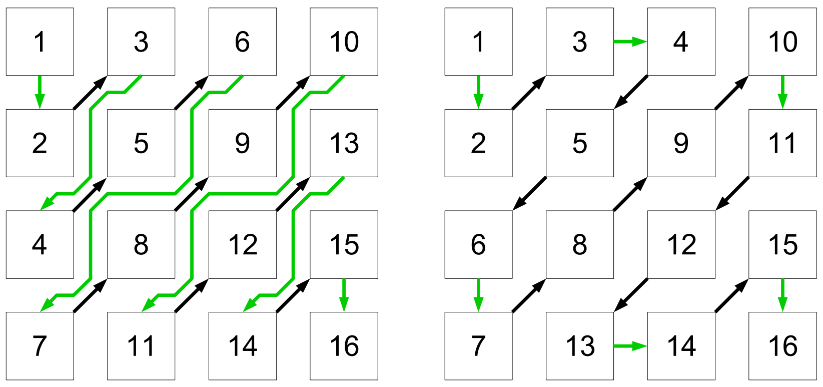
(a) Raster Transition (b) Snake Transition

Figure 4.9 – Scan transition for horizontal traversal

Snake

A snake scan differs from a raster scan in that, when the end of the line being traversed is reached, the next node to visit is the node immediately below the one just visited and the next line is traversed in the opposite direction to the one that has just been completed.

This method of scanning lines ensures that each node visited is a neighbour of the previous visited node as illustrated for a horizontal traversal in Figure 4.9b, and a diagonal traversal in Figure 4.10b.



(a) Diagonal Raster Transition **(b) Diagonal Snake Transition**

Figure 4.10 – **Scan transition for diagonal traversal**

Direction

The direction of a scan from the starting point can either be clockwise or anti-clockwise. Changing the direction from clockwise to to anti-clockwise will not change the relative position of mapped processing nodes, assuming all other parameters are unchanged, but will produce a mirror image of the clockwise mapping. For the application process graph defined in Section 4.3 and process map defined in Section 4.4 a pair of mappings that are different only by symmetry will have the same metrics. However, when sources and sinks external to the many-core array are added to the application process graph (see Section 6.1) symmetrical mappings will no longer have the same metrics.

Rotation

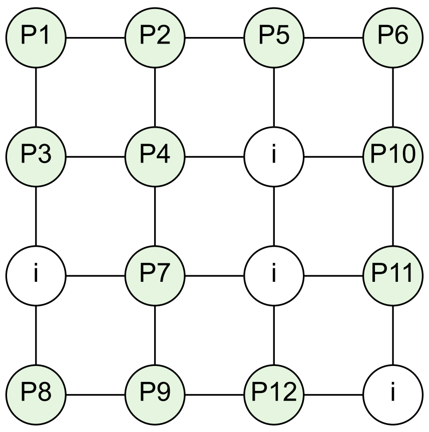
Rotation determines where the starting corner is by specifying a rotation in as either 0, 90, 180 or 270 degrees. When the starting position is the top-left corner, the rotation is 0. A rotation of 90 will move the starting position the top-right corner. As discussed above symmetry has no effect on the metrics when using the application process graph defined in Section 4.3 but will effect the metrics when sources and sinks external to the many-core array are added to the application process graph in Section 6.1.

4.8.3.5 Example Engineered Mappings

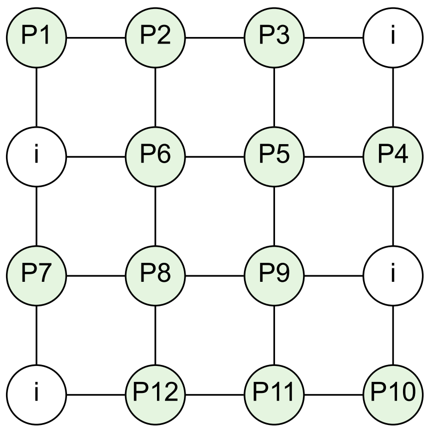
Combining the distribution patterns of Subsubsection 4.8.3.3 with the array node orderings of Subsubsection 4.8.3.4 produces a total of 64 different arrangements of processes within the many-core array. Due to rotations and reflections some of these arrangements will be topologically identical, as discussed above. The arrangements can, in principle, be applied to any size of array, however the resultant diversity of mappings will not be evident in small array such as 2×2 array.

Figure 4.11a shows the results of combining the distributed pattern of Figure 4.8c with an array node ordering define by a diagonal orientation, snake transition, zero rotation and clockwise direction of Figure 4.10b. The process of combining the node distribution list with the array node ordering, is the simple process of matching the reference numbers of the node distribution list with the reference numbers of the array node ordering and then placing the process from the list element into the array element.

Figure 4.11b shows the results of combining the distributed pattern of Figure 4.8c with an array node ordering define by a horizontal orientation, raster transition, zero rotation and clockwise direction of Figure 4.9a.



(a) Diagonal Snake



(b) Horizontal Rasta

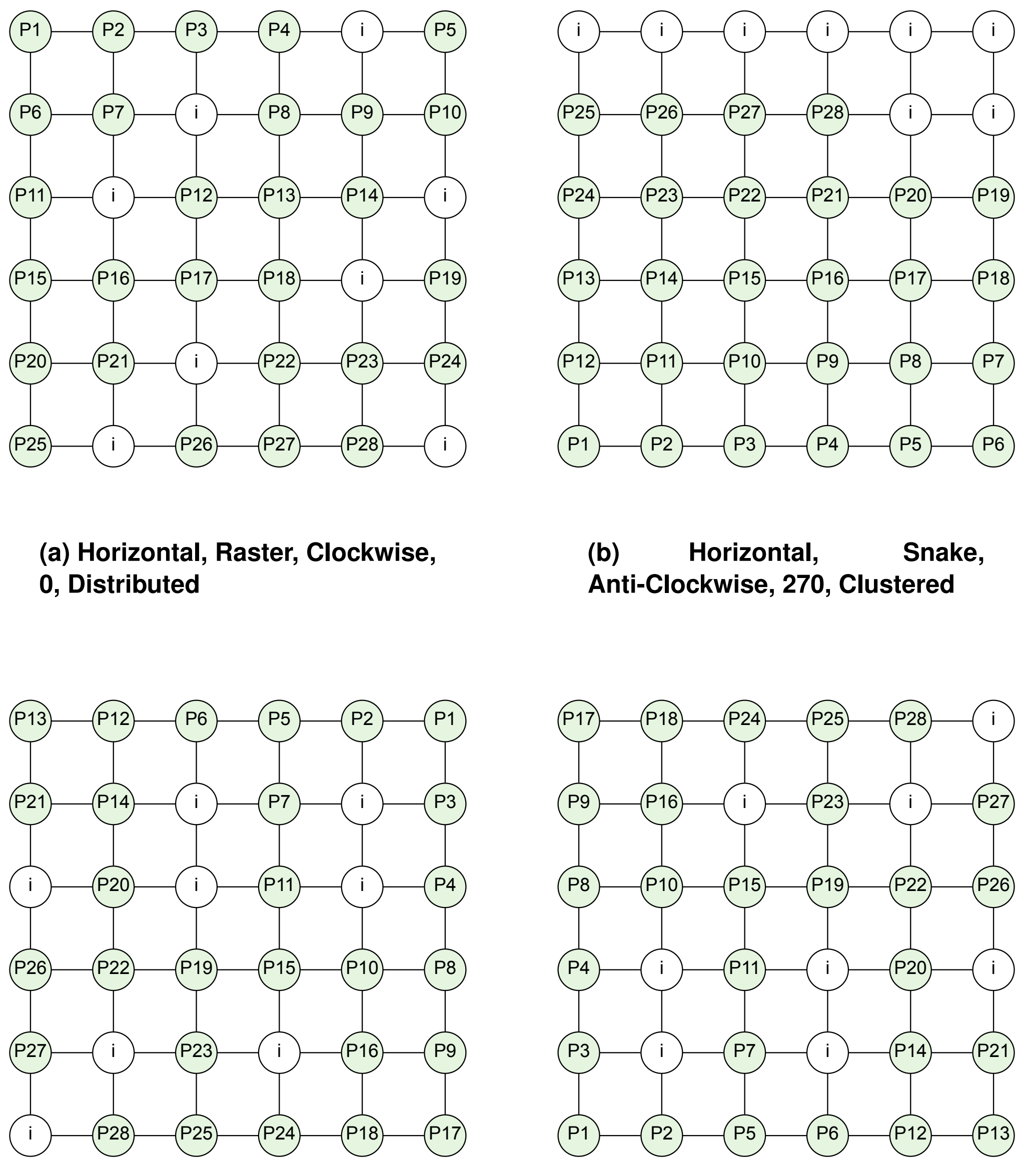
Figure 4.11 – **Mappings generated by initial placement algorithms**

The arrangement of processes in Figure 4.11 are examples of mappings without sources and sinks so any rotations and reflections that produce topologically identical arrangements will have identical objective values.

The mappings in Figure 4.12 show 4 examples of the possible 64 engineered mappings for the application process graph illustrated in Figure 4.2 with 26 processing nodes. The parameters used to generate each mapping are given in the caption of the figures.

4.9 Experiments and Analysis

The experiments presented in this section are designed to ensure that the model and evolutionary algorithm defined in this chapter can produce mappings that exhibit good core fault tolerance for a range of array sizes, each with a range of application process graph



(a) Horizontal, Raster, Clockwise, 0, Distributed

(b) Horizontal, Snake, Anti-Clockwise, 270, Clustered

(c) Diagonal, Snake, Clockwise, 90, Distributed

(d) Diagonal, Snake, Anti-Clockwise, 270, Clustered

Figure 4.12 – Examples of Engineered Mappings

sizes, as explained in Subsection 4.9.2. A piece of information, of particular interest, that is obtained from these experiments is the *minimum discovered fitness* for each combination of array size and application process graph size and the computational effort required to find the minimum discovered fitness. It should be noted that, due to the size of the search space and the nature of evolutionary algorithms, the minimum discovered fitness may not be the actually minimum fitness. For each minimum discovered fitness, statistics are collected from multiple evolutions to determine how quickly the evolutions found the minimum discovered fitness, in terms of computational effort. The results of these experiments will be used to determine suitable sizes of array and application process graph for experiments in the following chapters. Finally, there is an exploration of a variety of methods for determining when the evolutionary runs should be terminated.

Specifically, the experiments will:

- Determine the minimum discovered fitness for each combination of array size and application process graph size.
- Discover the optimal application process graph size (see Subsection 4.9.1) for a range of many-core array sizes.
- Examine the computational effort required to find the minimum discovered fitness.
- Investigate the effectiveness of a variety of termination conditions for the evolutionary algorithm.

4.9.1 Description and Assumptions

The effect of array size on the computational effort required for the evolutionary algorithm to find a good, if not optimal, solution will be investigated; the results of which will be used to determine practical array sizes for later experimental phases.

For any given size of array there is an maximum size of application process graph that can be mapped to the processing cores while ensuring that each processing core is protected by being adjacent to at least one idle core. The size of such an application process graph can be considered *optimal* in the sense that smaller application process graphs will leave more cores idle than are necessary for core fault tolerance and larger application process graphs will leave some processing cores unprotected. For some optimal application process graph sizes it will be possible to arrange processing and idle cores such that each processing core is adjacent to exactly one idle core, while in other cases some processing cores will be doubly protected by being adjacent to more than one idle core.

As the number of nodes in the application process graph decreases from the optimum number of nodes the problem of finding a mapping, where all processing cores are protected, becomes increasingly trivial. An extreme example is when there are only two processes which will both be protected when placed anywhere within an array with a minimum dimension of 2×2. When the number of processing nodes in the application process graph is close to the number of cores in the array then there will be a corresponding small number of idle cores. When the number of processes in the application process graph is greater than the optimal for the array, some of the processing cores will be left unprotected while the smaller number of idle cores reduces the number of unique combinations of locations where they can be placed, although these combinations are repeated many times. Neither of these scenarios pose interesting problems because finding an optimum placement for the idle cores becomes increasingly trivial as difference between the number of processes in the application process graph and the optimal application process graph size increases.

Consider the case, where the number of processing nodes in the application process graph is 50% of the number of cores, then the processing cores and idle cores can be arranged in a chequered pattern. In the case each processing core is adjacent to two, three or four idle cores depending whether the processing core is located in a corner, on an edge or internally. This illustrates that with 50% of the number of cores being idle, all processing cores will be protected by at least 2 idle cores, giving an initial lower bound the the optimal application process graph size. As graph size increases above the 50% of the number of cores, the mapping problem will become increasingly difficult. This analysis has been used to guide the initial selection of graph sizes for each array as being from 50% of the number or cores to a size of 2 nodes less than the number of cores.

4.9.2 Experiment Parameters

Array Size

Experiments will be conducted with array sizes from 4×4 to 8×8. The lower bound of this range has been chosen on the basis that an array of size 3×3 does not present an interesting problem for this research. By inspection, a placement of idle cores on a 3×3 array such that each processing core is adjacent to an idle core requires a minimum of three idle cores for which there is only one possible arrangement while leaving only 6 cores available for processing. A 3×3 array is therefore limited both in terms of the size fo application process graph that can be accommodated and the number and arrangement of idle cores.

The upper bound has been chosen with respect to the processing time required to find good solutions. The size of the problem, measured by the number of permutations of placing a graph of V_g nodes in an array of size $R \times VC$ is proportional to $RC/(RC - V_g)$, which increases rapidly as the array size and graph size increase. Since the basis of this research is to develop an algorithm for manageable-sized regions that interact with each other, the upper size of the problem has been chosen as an 8×8 array which presents a sufficiently challenging problem for this research while keeping the computation effort within acceptable limits.

The actual array size of a real many-core array will be determined at design time. It is not expected that arrays of size greater than 8×8 will be materially different, from a scientific point of view, from the array sizes studied, although the computation effort will be significantly greater.

Application Process Graph Size

As discussed in Subsection 4.9.1 the size of application process graphs has been chosen to be in the range of $(RC/2)$ nodes to $(RC - 2)$ nodes. Experiments in this phase will be used to explore whether these initial limits can be refined by increasing the lower bound limit and reducing the upper bound limit, without loss of generality.

4.9.2.1 Test Parameters

The following parameters were used for these tests:

Table 4.6 – Fault Tolerant Single Objective Test Parameters

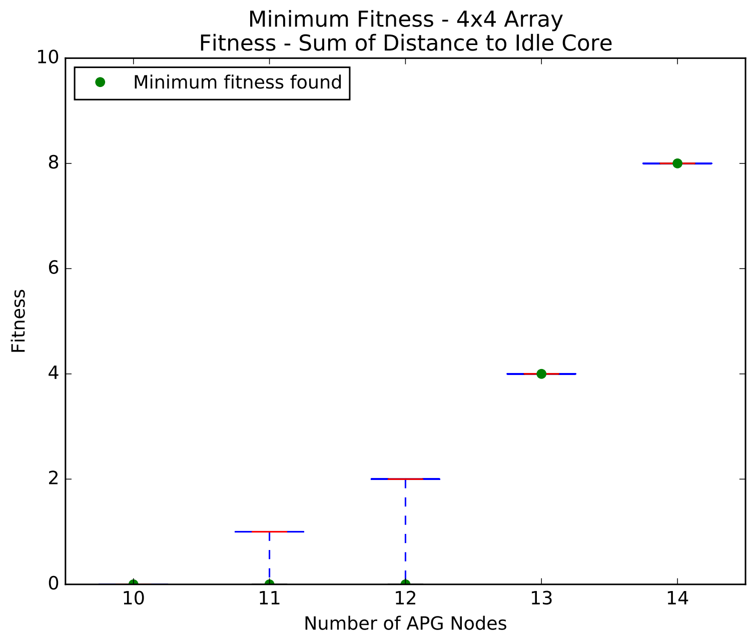
Parameter	Value
Array Size, $R \times C$	4×4 to 8×8
Graph size, V_g	$\frac{RC}{2}$ to $RC - 2$
Number of evolutions	100
Static generations	$\frac{R+C}{2} \times V_g$
Population size	10
Elite	2 fittest individuals cloned
Parents	4 fittest individuals used as parents
Descendants	2 from each parent via permutation
Mutation Rate	Fixed single mutation

4.9.3 Minimum Discovered Fitness

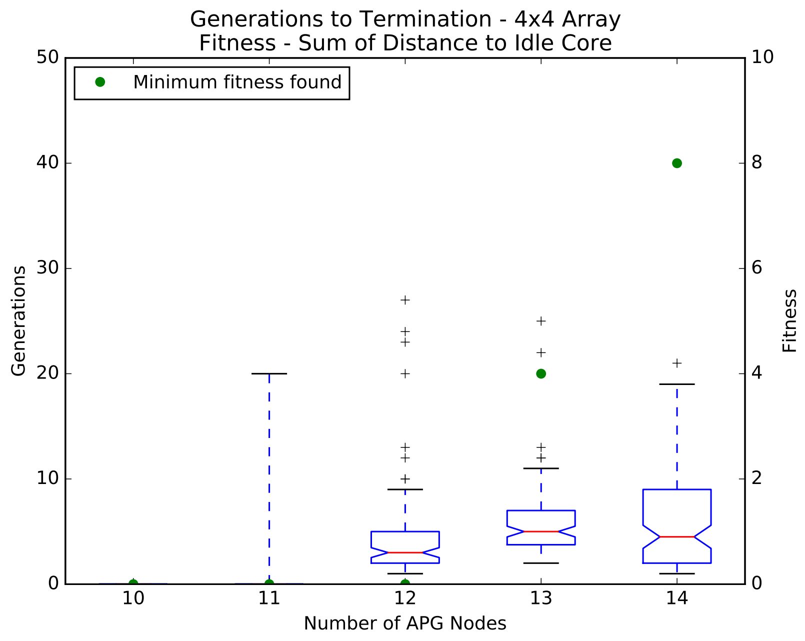
The results of the experiments for array sizes 4×4 to 8×8 are presented as pairs of box plots in figures 4.13 to 4.17, the first plot showing the minimum fitness found by each of 100 evolutions and the second plot showing the number of generations before termination of the same evolutions. Both diagrams also include the minimum discovered fitness of all 100 generations.

For the 4×4 array the minimum fitness flat box plot shows that all evolutions found the same minimum value of fitness indicating that the problem is sufficiently tractable for the evolutionary algorithm to find an optimal solution. This is borne out by the Generations to Termination box plot which shows that all evolutions terminated within 40 generations.

The minimum fitness box plots for array sizes of 5×5 to 8×8 show a clear pattern, that for application process graphs that have solutions with zero fitness, the range of fitness values found is small and in many cases the algorithm found a solution with fitness of zero in the first generation. Examining the box plot Figure 4.15a for the 6×6 array, there appears to be an anomaly for the graph with 22 nodes, for the number of generations taken to find the minimum discovered fitness, compared to graphs with application process graphs with 21, 23 and 24 nodes. This is explained by observing that the initial mapping algorithms work better for some combinations of graph size and array size than for others. In the case of the 6×6 array the initial placement algorithms finds zero fitness mappings for graphs sizes of 21, 23 and 24 but not for a graph size of 22 nodes. This is illustrated by the mappings (a), (b) and (c) in Figure 4.18. This property of the initial mapping algorithms working better for some combinations than other is further illustrated by the mapping of a 39 node graph onto a 7×7 array, which places nodes in exactly the same pattern as seen in the infinite



(a) Fitness - 4x4 Array

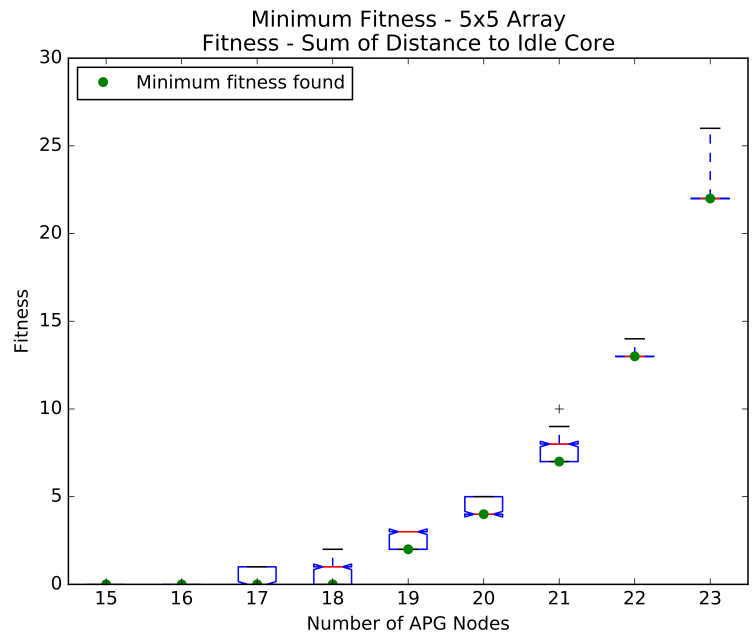


(b) Generations to Termination - 4x4 Array

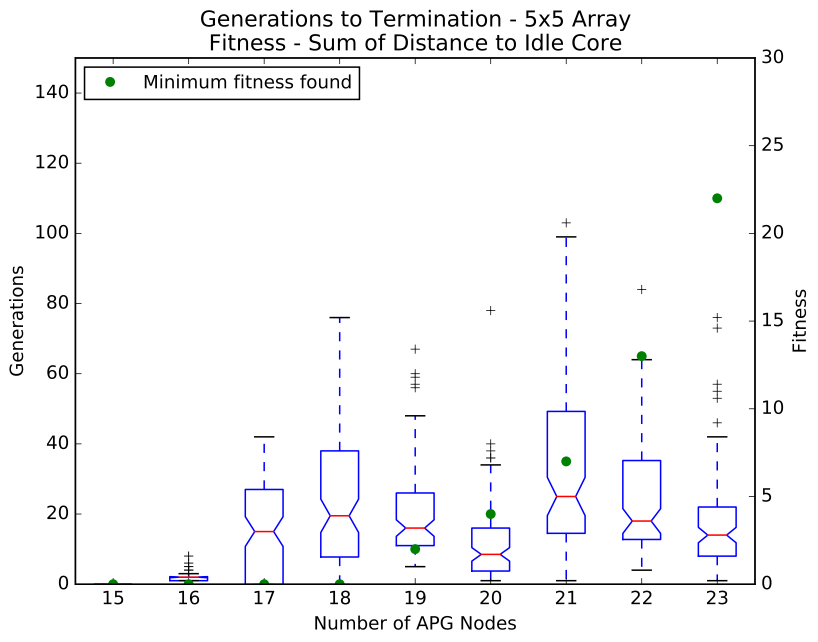
Figure 4.13 – Fault Free Mappings - 4x4 Array

plane as shown in Figure 4.19.

Application process graphs that have three or four fewer nodes than there are cores in the array have flat box plots for minimum fitness indicating that there are a small number of possible fitness levels. Figure 4.18d shows a mapping with the best fitness for a 32 node graph on a 6×6 array, in which the idle nodes are each positioned at the centre of a 3×3 area of the array. The number of generations it takes the evolutionary algorithm to



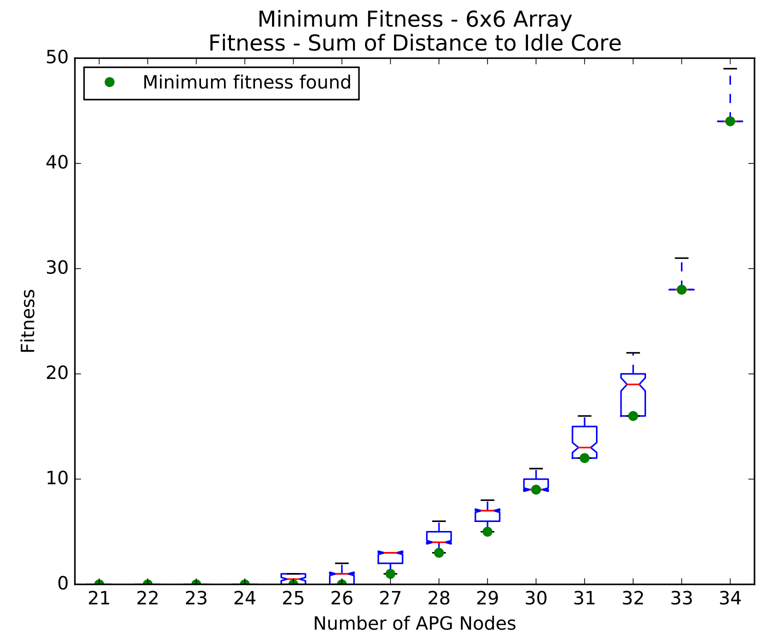
(a) Fitness - 5x5 Array



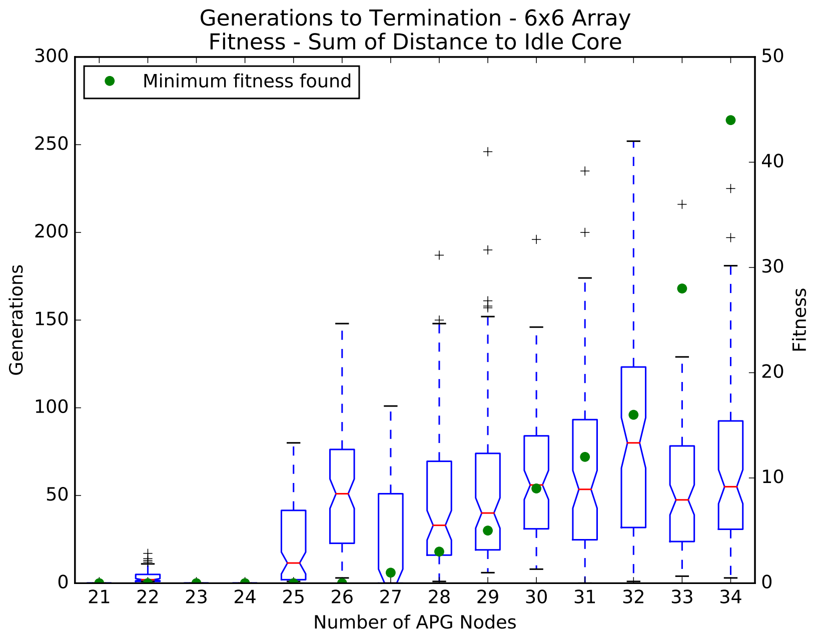
(b) Generations to Termination - 5x5 Array

Figure 4.14 – Fault Free Mappings - 5x5 Array

find the optimum solutions is also relatively low, which reflects the fact that the solutions that can be found are found relatively quickly, which in turn indicates that there are a large number of arrangements of idle cores that have the same minimum fitness. For a given array size the minimum fitness increases smoothly, while the associated number of generations required to find the minimum fitness is erratic. The erratic nature of the number of generations required to find minimum fitness, as the graph size increases, for a particular array size gives an indication of how difficult it is to find minimum fitness solutions, which



(a) Fitness - 6x6 Array



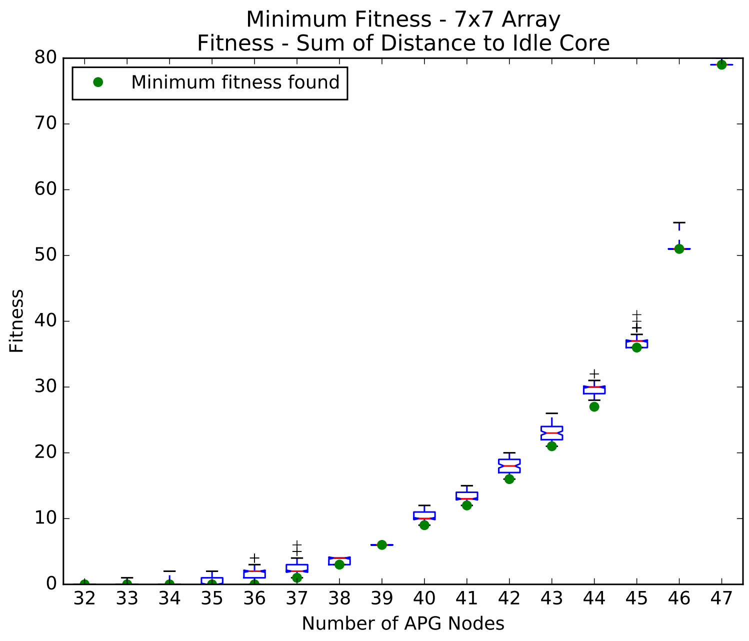
(b) Generations to Termination - 6x6 Array

Figure 4.15 – Fault Free Mappings - 6x6 Array

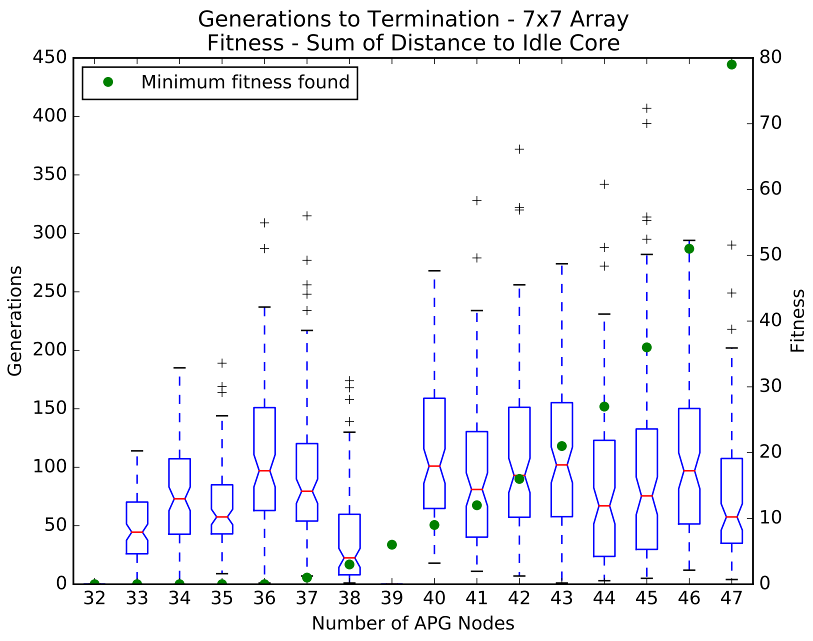
is a reflection of how many minimum fitness solutions there are. Some combinations of graph size and array size have many minimum fitness solutions, while other combinations have few minimum fitness solutions.

Summary

The results of these experiments have allowed us to obtain the minimum discovered fitness for each combination of array size and application process graph size. Although the



(a) Fitness - 7x7 Array

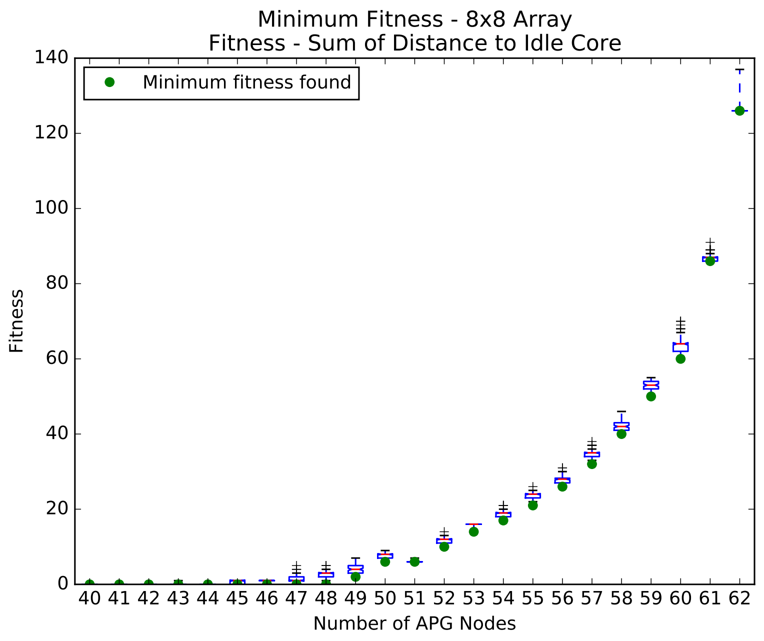


(b) Generations to Termination - 7x7 Array

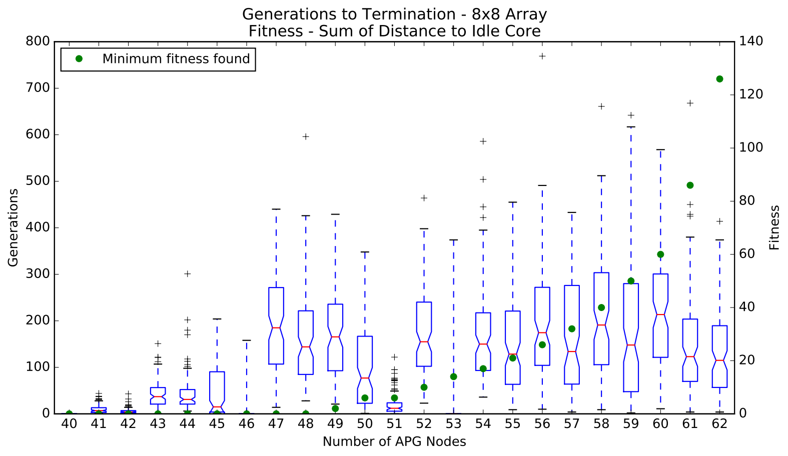
Figure 4.16 – Fault Free Mappings - 7x7 Array

minimum discovered fitness may not be the actual minimum fitness, the box plots showing the distribution of minimum fitness found by each of 100 evolutions show a small range of values, which is strong supporting evidence that the minimum discovered fitness is also the actual minimum.

The number of generations is takes the evolutionary algorithm to terminate is erratic leading to the conclusion that the most significant factor is the properties of the specific combina-



(a) Fitness - 8x8 Array



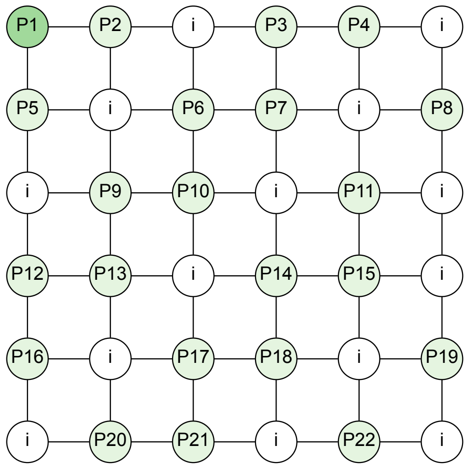
(b) Generations to Termination - 8x8 Array

Figure 4.17 – Fault Free Mappings - 8x8 Array

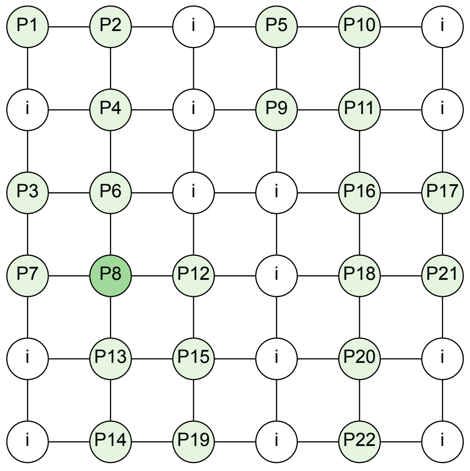
tion of array size and application process graph size, and the number minimum fitness mappings solutions each combination has. The greater the number of minimum fitness mappings, the quicker the evolutionary algorithm will find one.

Mappings for application process graph with a size smaller than the optimum size, are relatively easy for the evolutionary algorithm to find, often taking a few 10s of generations. For this reason these are not considered interesting application process graph sizes for use in later experiments.

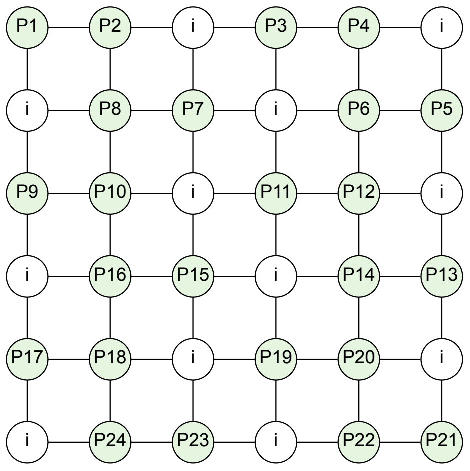
The search for zero fitness mappings for the optimal size of application process graph is generally harder than application process graph sizes one smaller or one larger than the optimal size (the exception is the 8×8 array). Finding zero fitness mappings for the optimal size of application process graph is therefore a challenging problem for later experiments.



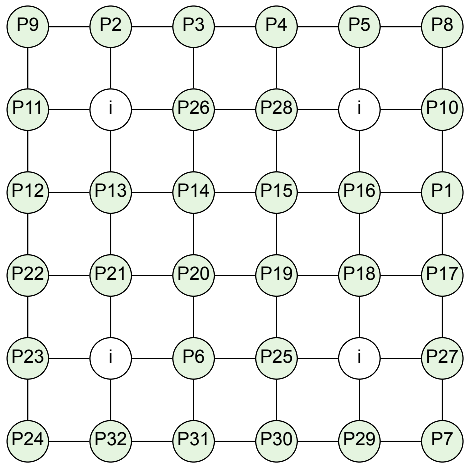
(a) Initial mapping for 22 nodes



(b) Initial mapping for 22 nodes



(c) Initial mapping for 24 nodes



(d) Best Fitness Mapping for 32 Nodes

Figure 4.18 – Selected Fault Free Mappings - 6x6 Array

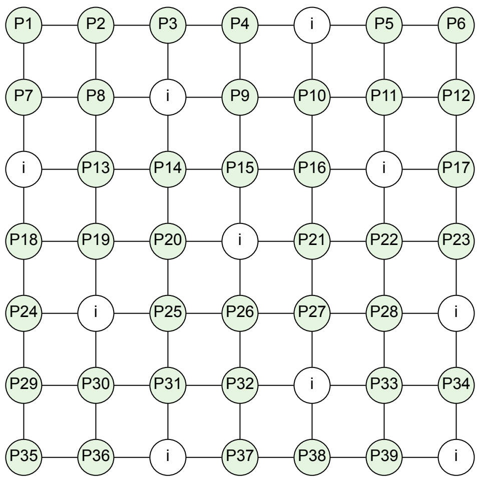


Figure 4.19 – Initial Mapping of 39 Nodes on a 7x7 Array

4.9.4 Optimum Graph Size

The results presented in figures 4.13 to 4.17 show the maximum size of application process graph, for each array size, for which the fault tolerance objective value is zero. These are referred to as the optimum application process graph size for the array size which are listed in Table 4.7 along with the percentage of cores of the array that are idle. Figure 4.21 shows example mappings with zero fitness for the optimal application process graphs.

Table 4.7 – Summary of optimum graph size for zero fitness

Array Size	Graph Size	Percent Idle
4	12	25.0
5	18	28.0
6	26	27.8
7	36	26.5
8	48	25.0
∞	n/a	20.0

As a reference for the results obtained from the experiments, Figure 4.20 illustrates a portion of an infinite plane with an ideal arrangement of idle cores where every processing core is adjacent to exactly one idle core and each idle core protects exactly four processing cores. The arrangement is a tiling of a cross shape with an idle core at its centre and a processing core on each of the four compass points producing a pattern of idle cores which are all separated by a chess knight's move. Portions of this pattern are found in the all of the mappings in Figure 4.21. To protect all processing cores In the infinite plane, only 20% of cores are required to be idle.

The best arrangements discovered by the evolutionary algorithm is 25% of the cores being idle, for array sizes where R and C are multiples of 4. It might naturally be expected that as arrays increase in size the percent of idle cores will tend towards the value of 20% of the infinite plane. This, however, is not the case for the array sizes studied, with the lowest % of idle cores being found in the 4×4 array and the 8×8 array consisting of four copies of the 4×4 array.

The explanation can be found by examining the edges of the arrays. It is possible to select a 4×4 portion of the infinite array where each idle core protects three processing cores and each processing core is protected by exactly one idle core. For portions of the infinite array of sizes 5, 6, 7 and 8, which have been studied, there are always processing cores on the edge of the array that are unprotected. To protect these requires replacement of some of the processing cores with idle cores together with a rearrangement. The proportion of cores on the edges compared to the array as a whole is significant for small array sizes, giving a high cost to the replacement of processing cores with idle cores when compared

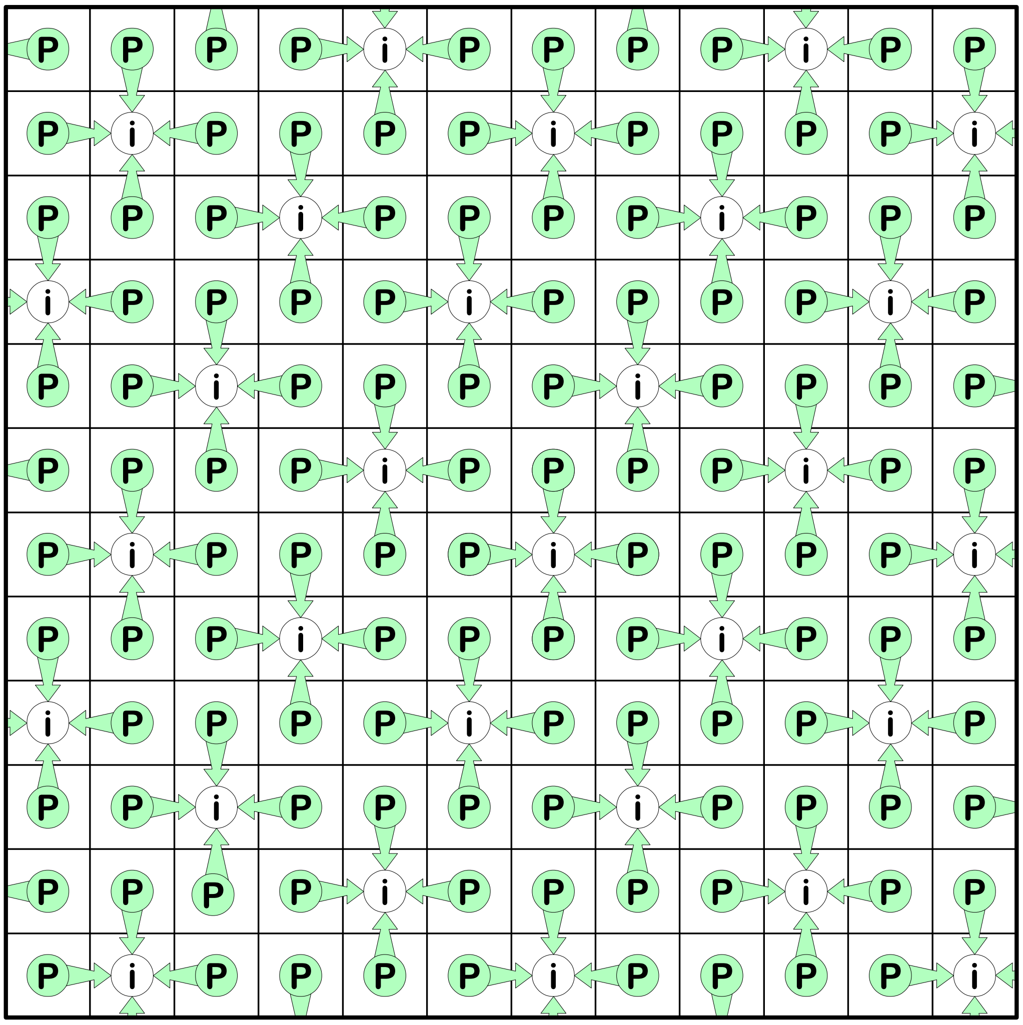


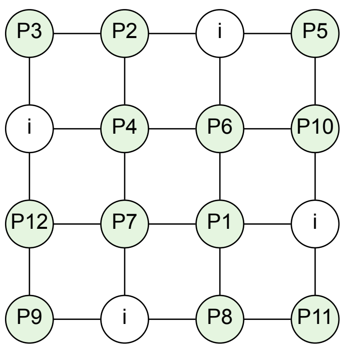
Figure 4.20 – **Minimum Fitness for generalized $n \times n$ array**

to the array size. This proportion reduces as the array size increases indicating that larger arrays will have less than 25% of cores idle. A rough calculation indicates that the crossover may occur with an array size of about 15.

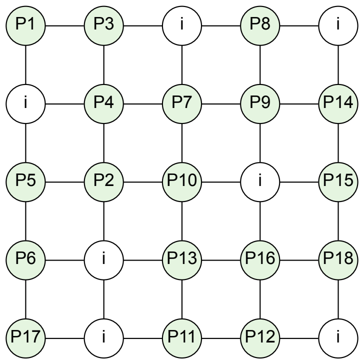
For the 4×4 array there is only one arrangement of idle cores which achieve a fitness value of zero. This arrangement is a subset of the idle core arrangement found in the infinite plane of Figure 4.20. The arrangement of idle cores for the 4×4 array is found in all of the mappings of the larger arrays and can be seen five times in the 8×8 array.

Summary

For an infinite array the percentage of cores that need to remain idle to fully protect the processing cores is 20%. For the array sizes used in these experiments the small percentage of idle cores was 25%, which achieved for the 4×4 array and can therefore be achieved for any rectangular array whose number of rows and columns of cores are multiples of 4.

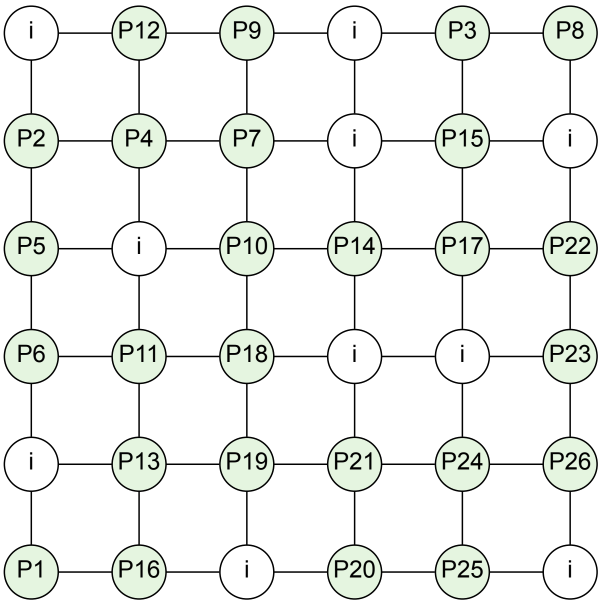


(a) Minimum discovered fitness for a 4×4 array

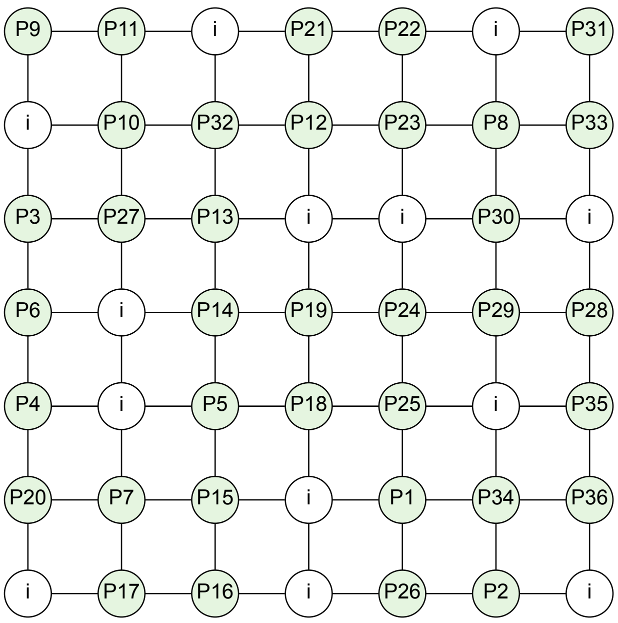


(b) Minimum discovered fitness for a 5×5 array

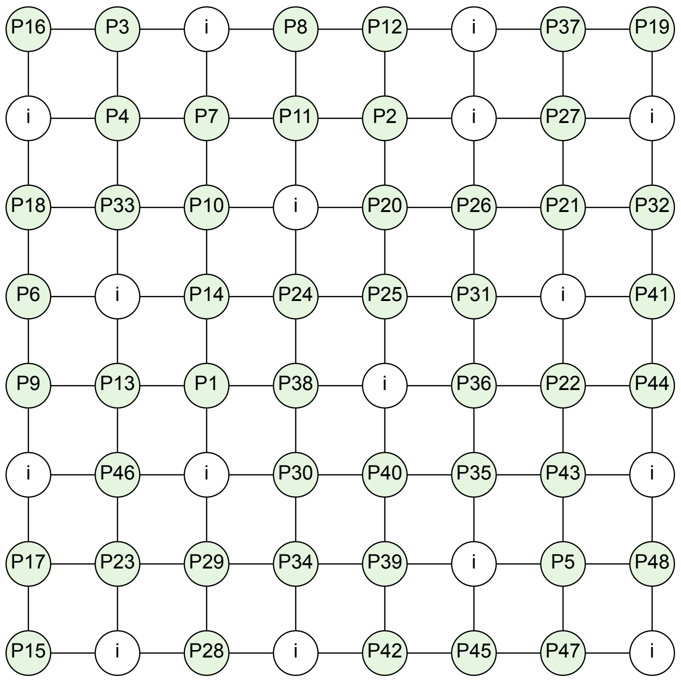
A smaller percentage of idle cores may be sufficient for larger array sizes, although this has not be explored in this work.



(c) Minimum discovered fitness for a 6×6 array



(d) Minimum discovered fitness for a 7×7 array



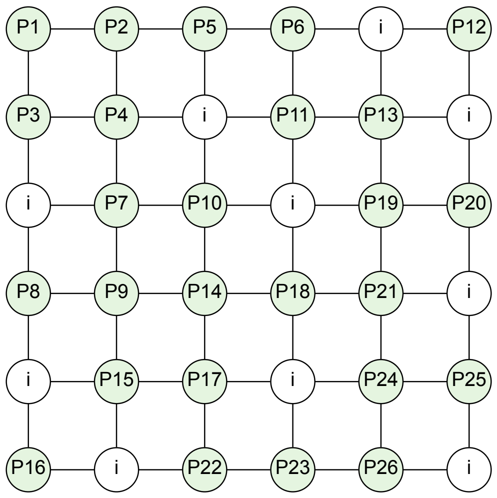
(e) Minimum discovered fitness for na 8×8 array

Figure 4.21 – Minimum discovered fitnesses for array sizes 4×4 to 8×8

4.9.5 Evolution of a Mapping of a 26 Node Graph to a 6x6 Array

To illustrate how the evolutionary algorithm works this section presents an example of an evolution from an engineered mapping through to a mapping with a zero fitness value to illustrate how the evolutionary algorithm works. The mapping of a 26 node application process graph onto a 6×6 many-core array was chosen because it a problem that is large enough to be interesting and the solution was found reasonably quickly. Figure 4.22 illustrates the mappings from the initial engineered mapping to the final mapping with a fitness of zero.

The initial mapping was generated by the initial placement algorithm using a diagonal snake pattern with distribution of idle nodes. This generates a mapping of fitness value of 4 with nodes P1, P2, P9 and P14 all having individual fitness values of 1.

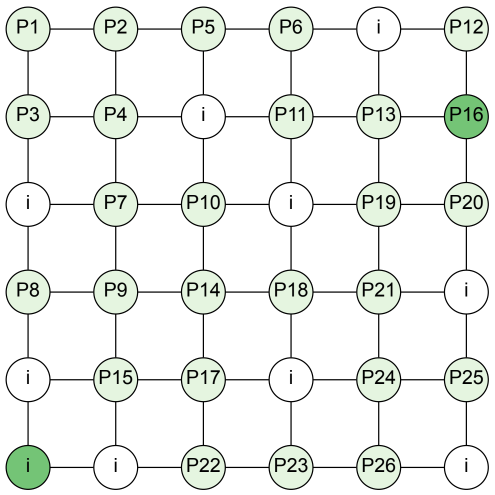


(a) Initial Mapping

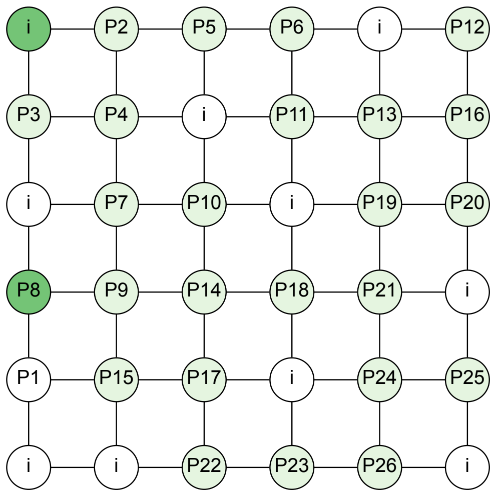
The transformation from the initial mapping to the final mapping went through 6 transformations over 17 generations, while the other 11 generations produced cloned individuals from the previous generations. The mappings show the 6 transformations with the nodes that have been exchanged highlighted. Table 4.8 provides a commentary of the interesting points of the evolutionary chain showing at each point the generation, fitness and rank of the individual that was part of the evolutionary chain.

Summary

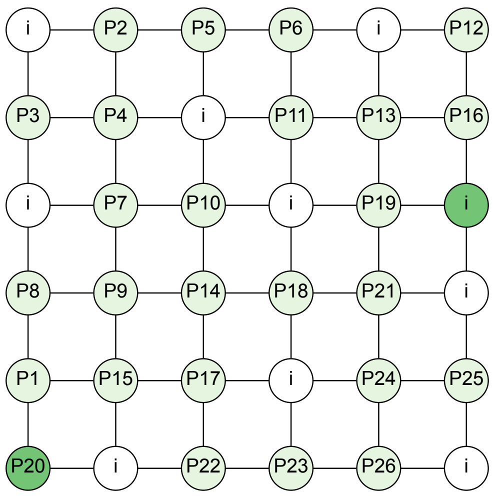
Tracing through the evolution from an engineered mapping to a zero fitness mapping has illustrated how the evolutionary algorithm search functions and the reason that each individual in the chain was selected for the next generation.



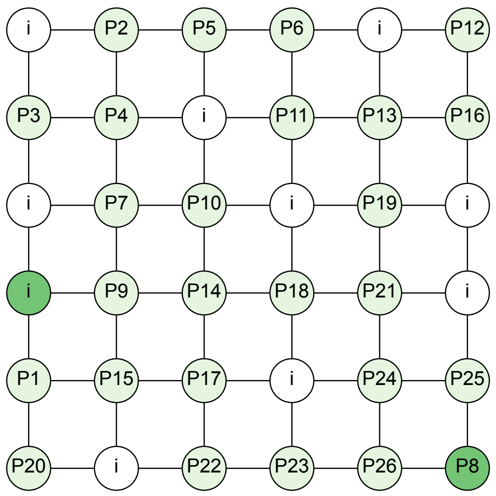
(b) Transformation 1



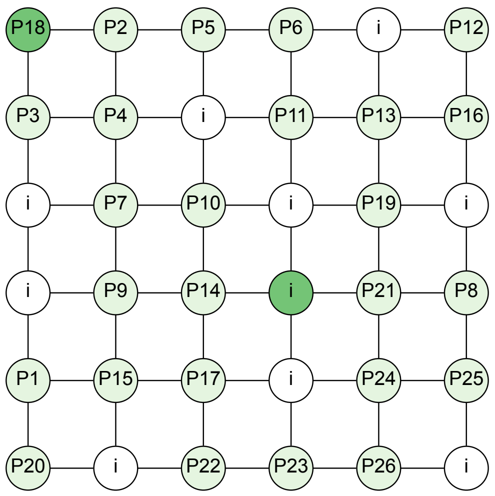
(c) Transformation 2



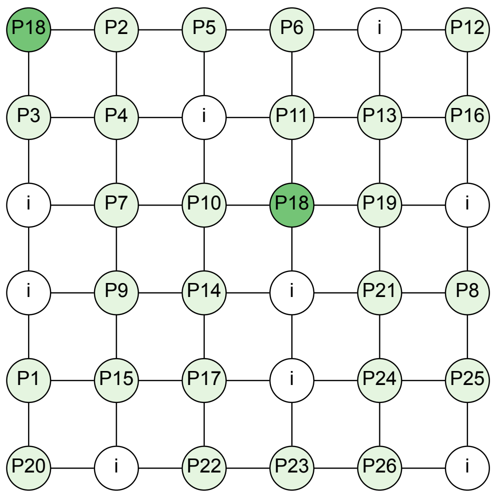
(d) Transformation 3



(e) Transformation 4



(f) Transformation 5



(g) Transformation 6

Figure 4.22 – Evolution of a mapping of a 26 node graph to a 6x6 array

Table 4.8 – **Summary of evolution of mapping of a 26 node graph to a 6×6 array**

Gn	Ft	Rk	Opr	Description
0	4	2	Initial placement	One of two mappings with the best fitness value of 4 and ranked 2^{nd} so that it will be cloned and used as a parent.
1	4	1	Cloned	With a rank of 1 this individual will again be cloned and used as a parent.
2	4	3	Cloned	With a rank of 3 this individual is is not cloned but is used as a parent.
3	5	4	Descendant	A descendant of the previously cloned individual, it has a lower fitness than its parent but is ranked 4^{th} so will be used as a parent.
4	4	4	Descendant	A descendant of the previous individual it is ranked 4^{th} so will be used as a parent.
5	4	4	Cloned	
6	3	1	Descendant	This is the first individual in this evolutionary chain that has a better fitness than the generation zero ancestor.
7-9	3	1-2	Cloned	Cloned for three generations each with a ranking of 1 or 2.
10	3	1	Descendent	Having the same fitness as its parent, this individual is ranked 2^{nd} replacing its parent, ranked 3^{rd}, as one of the two elite individuals of this generation.
11	2	1	Descendent	A further improvement in fitness.
12-16	2	1-2	Cloned	Cloned for five generations each with a ranking of 1 or 2.
17	0	1	Descendent	The final transformation of the genome resulted in a jump in fitness from 2 to 0. Zero being the fitness minimum value attainable the evolution stops here.

4.9.6 Termination Condition

When using a single objective, strategies can be used to define the termination conditions of the evolutionary algorithm to ensure that the computational effort is kept as low as possible while being sufficient to find good solutions for the given objective. Termination of the evolutionary algorithm by one of three conditions listed below has been investigated:

- Maximum computation effort expended
- Zero fitness found
- Termination parameter t, the number of generations the fitness remains static

An informed determination of a strategy for the selection of a suitable termination point in terms of the number of generations t is useful to balance the computational effort to find solutions against the fitness of the solutions.

To understand the effect of termination parameter t, the evolutionary algorithm was run using the parameter values described in Subsection 4.9.2, but without the static generations termination parameter, allowing the algorithm to run until either the fitness is zero or the maximum number of generations is reached. The data collected was then analysed to determine the influence that a variety of termination conditions had upon the computational effort used by the evolutionary algorithm.

The results are given in Tables 4.12 to 4.20. Descriptions of the values in the table columns are given here:

The first three columns of each table are:

Table 4.9 – **Termination Condition Array and Application Process Graph Size**

Name	Description
a	The size of the array in terms of the dimension of one side of the array.
n	The number of nodes in the application process graph being mapped to the array.
f_m	The minimum fitness the evolutionary algorithm discovered (not necessarily the minimum possible fitness).

Each table contains data for three termination conditions. Each termination condition has the following columns:

The computational effort is defined as the total number of individuals the evolutionary algorithm evaluated before termination. Note that for these tables computational effort has been divided by 10 to enhance the presentation and readability.

Each table contains data for two of the six termination conditions which are:

Table 4.10 – Termination Condition Column Descriptions

Name	Description
tc	The termination condition i.e. the number of generations the fitness remains static before the evolutionary algorithm terminates.
$\%f_m$	The percentage of evolutions that found a solution with the minimum fitness.
$\overline{ce_{f_m}}$	The average computational effort of those evolutions that found a mapping with the minimum fitness. This figure ignores any evolutions that did not find a mapping with the minimum fitness. It does not say anything about the evolutions that failed to find a mapping with the minimum fitness. A lower figure is more desirable, however, if no evolutions found a mapping with the minimum fitness this value is set to zero.
$\overline{ce_t}$	The average computational effort across all evolutions. If a computational effort budget is imposed on the evolutionary algorithm, then this figure can be directly compared to the budget to determine if the termination condition will keep the evolutionary algorithm within budget. If all evolutions found a mapping with the minimum fitness this value is set to zero.
$ce_t/\%f_m$	The total computational effort for all evolutions divided by the number of evolutions that found a mapping with the minimum fitness. This figure gives an indication of the cost of finding solutions with minimum fitness, however it does not take into account the fitness of the non-minimum solutions. It does not say anything about the fitness of the evolutions that did not find a mapping with the minimum fitness. If no evolutions found a mapping with the minimum fitness this value is set to zero.
$\bar{f}$	The average fitness of all evolutions.

Each table contains data for three termination conditions. Each termination condition has the following columns:

The different formulae for calculating the termination condition are distinguished by the rate at which they grow as the number of nodes in the graph and the number of cores in the array grow. The constants for the formulae have been chosen so that the values are near equal for a 26 node graph on a 6×6 array. This particular combination of graph and array size has been chosen because the 6×6 array is a suitable sized array of cores to apply the fault tolerant mechanisms of the research and 26 is the largest graph which can be mapped to a 6×6 with a fitness of zero.

Analysis of Results

In general increasing the number of generations of the termination condition increases the number of evolutions that find a mapping with the minimum fitness ($\%f_m$), the cost of finding each minimum value mapping ($\overline{ce_{f_m}}$) and the average computational effort ($\overline{ce_t}$). This is expected as the evolutionary algorithm is generating and evaluating more individuals which increases the computational effort and the likelihood of finding more minimum fitness mappings. The total computation effort for all evolutions divided by the number of evolutions that found a minimum fitness ($ce_t/\%f_m$) and the average fitness ($\bar{f}$) tends to decrease. These last two measures given the best indication that a higher termination condition gives better value for the total computational effort.

A good illustration of the observed affect is the mapping of a 21 node graph onto a 5x5 array, Table 4.13, which shows a 25% increase in the number of minimum value mappings found in return for a 20% increase in total computational effort.

More significant than the absolute value of the initial termination condition, is the effect of the dynamically changing termination condition. Table 4.14 shows that, In most cases, increasing the termination condition each time the fitness improves is a successful strategy for find a greater number of minimum value solutions.

An example where the dynamic termination condition has not worked is where a 12 node graph is being mapped to a 4x4 array, Table 4.14, where both the dynamic and non-dynamic conditions find only 33% of the minimum fitness mappings compared to the control, Table 4.12, that finds the minimum fitness in all 100 evolutions. This is a result of the initial termination condition value of 48 being too low to be effective. The corresponding termination condition of 150 in the $k(n/a^2)$ column, Table 4.13, finds 65% of the minimum fitness mappings. If the starting condition for the dynamic variant was also 150 then we would expect it to perform much better.

The dynamic increase is achieved by taking the number of generations since the last improvement in fitness and multiplying by 10. The value of 10 was chosen by making the observation that each improvement in in fitness took between 5 and 10 times the number of generations than the previous improvement in fitness.

Table 4.11 – **Termination Conditions**

Name	**Description**
No Condition	There is no termination condition dependent on the number of generations the fitness remains static. Each evolution continues until a fitness of zero is found or the maximum number of generations has been reached. This is the control group which shows how the evolutionary algorithm performs when constrained only by an arbitrary limit to the computational effort.
$kn : k = 6$	The number of generations that the fitness remains static before the evolution is terminated is calculated as the number of nodes multiplied by the constant 6.
$ka^2 : k = 4$	The number of generations that the fitness remains static before the evolution is terminated is calculated as the number of cores, that is the square of the array size, multiplied by the constant 4.
$k(n/a^2) : k = 200$	The number of generations that the fitness remains static before the evolution is terminated is calculated as the number of nodes divided by the number of cores, multiplied by the constant 200.
$a.n$	The number of generations that the fitness remains static before the evolution is terminated is calculated as the number of nodes multiplied by the number of cores.
$Dynamic : a.n$	The initial number of generations that the fitness remains static before the evolution is terminated is calculated as the number of nodes multiplied by the number of cores. As an evolution progresses and the fitness improves, it becomes more difficult to find a mapping with an improved fitness. To reflect this the dynamic approach adjusts the termination condition each time the fitness improves by taking the number of generations evaluated since the last improvement, multiplying by 10 and then replacing the current condition if the new value is greater than the existing value.

Table 4.12 – **Analysis of Termination Condition:** No Condition **and** $kn : k = 6$

			No Condition						$kn : k = 6$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
4	11	0	Max	100	37.24	37.24	37.24	0.00	66	87	29.54	34.28	39.40	0.13
4	12	0	Max	100	136.93	136.93	136.93	0.00	72	43	32.21	56.58	131.58	1.14
4	13	4	Max	100	7.20	7.20	7.20	4.00	78	100	7.20	7.20	7.20	4.00
4	14	8	Max	100	5.07	5.07	5.07	8.00	84	100	5.07	5.07	5.07	8.00
5	16	0	Max	100	2.20	2.20	2.20	0.00	96	100	2.20	2.20	2.20	0.00
5	17	0	Max	100	40.60	40.60	40.60	0.00	102	95	35.58	38.90	40.95	0.05
5	18	0	Max	100	104.20	104.20	104.20	0.00	108	81	72.88	85.95	106.11	0.19
5	19	2	Max	100	90.35	90.35	90.35	2.00	114	79	55.62	71.08	89.97	2.21
5	20	4	Max	100	65.40	65.40	65.40	4.00	120	83	35.36	52.02	62.67	4.17
5	21	7	Max	100	263.09	263.09	263.09	7.00	126	69	86.12	121.31	175.81	7.32
5	22	13	Max	100	22.11	22.11	22.11	13.00	132	99	20.53	21.87	22.09	13.01
5	23	22	Max	100	18.74	18.74	18.74	22.00	138	100	18.74	18.74	18.74	22.00
6	22	0	Max	100	3.81	3.81	3.81	0.00	132	100	3.81	3.81	3.81	0.00
6	25	0	Max	100	83.42	83.42	83.42	0.00	150	90	63.42	72.26	80.29	0.10
6	26	0	Max	100	264.60	264.60	264.60	0.00	156	58	116.36	158.33	272.98	0.42
6	27	1	Max	100	667.52	667.52	667.52	1.00	162	22	238.41	293.96	1336.18	1.78
6	28	3	Max	94	1614.21	1525.57	1622.95	3.06	168	34	261.71	758.84	2231.88	3.47
6	29	5	Max	85	1613.78	1410.95	1659.94	5.15	174	36	346.47	1326.87	3685.75	5.09
6	30	9	Max	100	217.03	217.03	217.03	9.00	180	75	92.24	129.01	172.01	9.25
6	31	12	Max	100	625.55	625.55	625.55	12.00	186	51	123.31	257.45	504.80	12.50
6	32	16	Max	100	505.37	505.37	505.37	16.00	192	90	501.27	494.48	549.42	16.30
6	33	28	Max	100	106.30	106.30	106.30	28.00	198	93	81.27	94.09	101.17	28.07
6	34	44	Max	100	66.60	66.60	66.60	44.00	204	100	66.60	66.60	66.60	44.00

Table 4.13 – **Analysis of Termination Condition:** $ka^2 : k = 4$ **and** $k(n/a^2) : k = 200$

			$ka^2 : k = 4$						$k(n/a^2) : k = 200$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
4	11	0	64	84	28.25	33.97	40.44	0.16	137	100	37.24	37.24	37.24	0.00
4	12	0	64	40	29.50	51.95	129.88	1.20	150	65	60.95	93.13	143.28	0.70
4	13	4	64	100	7.20	7.20	7.20	4.00	162	100	7.20	7.20	7.20	4.00
4	14	8	64	100	5.07	5.07	5.07	8.00	175	100	5.07	5.07	5.07	8.00
5	16	0	100	100	2.20	2.20	2.20	0.00	128	100	2.20	2.20	2.20	0.00
5	17	0	100	95	35.58	38.80	40.84	0.05	136	98	38.14	40.10	40.92	0.02
5	18	0	100	80	72.15	84.35	105.44	0.20	144	86	78.29	91.82	106.77	0.14
5	19	2	100	71	48.62	67.51	95.08	2.29	152	88	65.35	77.57	88.15	2.12
5	20	4	100	76	27.22	47.78	62.87	4.24	160	90	44.67	57.49	63.88	4.10
5	21	7	100	60	76.68	111.98	186.63	7.43	168	75	94.15	133.05	177.40	7.25
5	22	13	100	99	20.53	21.55	21.77	13.01	176	100	22.11	22.11	22.11	13.00
5	23	22	100	100	18.74	18.74	18.74	22.00	184	100	18.74	18.74	18.74	22.00
6	22	0	144	100	3.81	3.81	3.81	0.00	122	100	3.81	3.81	3.81	0.00
6	25	0	144	89	62.49	71.65	80.51	0.11	138	88	61.58	70.95	80.63	0.12
6	26	0	144	56	113.13	153.12	273.43	0.44	144	56	113.13	153.12	273.43	0.44
6	27	1	144	20	230.90	279.69	1398.45	1.80	150	20	230.90	284.49	1422.45	1.80
6	28	3	144	31	260.35	645.25	2081.45	3.56	155	31	260.35	750.63	2421.39	3.51
6	29	5	144	31	354.84	1310.07	4226.03	5.14	161	34	347.97	1319.78	3881.71	5.11
6	30	9	144	70	82.70	118.76	169.66	9.31	166	72	86.86	125.19	173.88	9.28
6	31	12	144	49	120.39	236.60	482.86	12.53	172	51	123.31	250.59	491.35	12.50
6	32	16	144	77	522.06	486.26	631.51	16.69	177	86	508.51	492.69	572.90	16.42
6	33	28	144	85	69.87	88.32	103.91	28.16	183	91	77.88	92.87	102.05	28.09
6	34	44	144	99	65.29	66.15	66.82	44.03	188	99	65.29	66.59	67.26	44.03

Table 4.14 – **Analysis of Termination Condition:** $4 \leq a \leq 6$; $a.n$ **and** $a.n : Dynamic$

			$a.n$						$a.n : Dynamic$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
4	11	0	44	67	21.84	29.15	43.51	0.33	44	67	21.84	29.15	43.51	0.33
4	12	0	48	33	22.55	41.57	125.97	1.34	48	33	22.55	42.25	128.03	1.34
4	13	4	52	100	7.20	7.20	7.20	4.00	52	100	7.20	7.20	7.20	4.00
4	14	8	56	100	5.07	5.07	5.07	8.00	56	100	5.07	5.07	5.07	8.00
5	16	0	80	100	2.20	2.20	2.20	0.00	80	100	2.20	2.20	2.20	0.00
5	17	0	85	94	34.94	37.94	40.36	0.06	85	94	34.94	37.94	40.36	0.06
5	18	0	90	77	70.30	82.21	106.77	0.23	90	93	90.08	93.47	100.51	0.07
5	19	2	95	69	46.78	65.98	95.62	2.31	95	74	53.89	71.29	96.34	2.26
5	20	4	100	76	27.22	47.78	62.87	4.24	100	80	34.53	50.95	63.69	4.20
5	21	7	105	62	78.06	113.92	183.74	7.41	105	74	107.57	154.80	209.19	7.28
5	22	13	110	99	20.53	21.65	21.87	13.01	110	100	22.11	22.11	22.11	13.00
5	23	22	115	100	18.74	18.74	18.74	22.00	115	100	18.74	18.74	18.74	22.00
6	22	0	132	100	3.81	3.81	3.81	0.00	132	100	3.81	3.81	3.81	0.00
6	25	0	150	90	63.42	72.26	80.29	0.10	150	90	63.42	72.26	80.29	0.10
6	26	0	156	58	116.36	158.33	272.98	0.42	156	75	170.59	198.51	264.68	0.25
6	27	1	162	22	238.41	293.96	1336.18	1.78	162	50	425.46	445.57	891.14	1.50
6	28	3	168	34	261.71	758.84	2231.88	3.47	168	46	331.46	942.06	2047.96	3.35
6	29	5	174	36	346.47	1326.87	3685.75	5.09	174	49	460.73	1517.95	3097.86	4.96
6	30	9	180	75	92.24	129.01	172.01	9.25	180	86	149.43	170.29	198.01	9.14
6	31	12	186	51	123.31	257.45	504.80	12.50	186	78	401.77	456.68	585.49	12.23
6	32	16	192	90	501.27	494.48	549.42	16.30	192	96	504.49	499.37	520.18	16.12
6	33	28	198	93	81.27	94.09	101.17	28.07	198	96	94.33	103.12	107.42	28.04
6	34	44	204	100	66.60	66.60	66.60	44.00	204	100	66.60	66.60	66.60	44.00

Table 4.15 – **Analysis of Termination Condition:** $a = 7$**;** No Condition **and** $kn : k = 6$

			No Condition						$kn : k = 6$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
7	29	0	Max	100	2.58	2.58	2.58	0.00	174	100	2.58	2.58	2.58	0.00
7	33	0	Max	100	70.27	70.27	70.27	0.00	198	98	66.94	69.56	70.98	0.02
7	34	0	Max	100	115.80	115.80	115.80	0.00	204	96	106.00	111.49	116.14	0.04
7	35	0	Max	100	152.00	152.00	152.00	0.00	210	86	105.17	129.23	150.27	0.14
7	36	0	Max	98	1391.72	1373.26	1401.29	0.02	216	31	724.26	602.22	1942.65	0.71
7	37	0	Max	18	4149.11	1971.82	10954.56	1.41	222	49	9487.18	5218.25	10649.49	1.04
7	38	3	Max	100	636.50	636.50	636.50	3.00	228	45	135.44	193.22	429.38	3.55
7	40	8	Max	17	4660.65	1863.70	10962.94	8.90	240	3	256.00	3287.85	109595.00	7.00
7	41	12	Max	83	1756.07	1558.17	1877.31	12.17	246	30	341.67	1210.04	4033.47	11.77
7	42	16	Max	93	2135.59	2051.26	2205.66	16.07	252	17	844.53	1039.97	6117.47	16.41
7	43	21	Max	69	1202.58	932.16	1350.96	21.31	258	35	469.49	2383.83	6810.94	17.46
7	44	27	Max	35	1966.57	1220.95	3488.43	27.65	264	9	1353.78	2849.63	31662.56	21.98
7	45	36	Max	100	374.86	374.86	374.86	36.00	270	70	169.27	244.60	349.43	36.32
7	46	51	Max	100	152.21	152.21	152.21	51.00	276	95	135.95	147.31	155.06	51.10
7	47	79	Max	100	102.55	102.55	102.55	79.00	282	95	84.27	96.65	101.74	79.05

Table 4.16 – Analysis of Termination Condition: $a = 7$; $ka^2 : k = 4$ **and** $k(n/a^2) : k = 200$

			$ka^2 : k = 4$						$k(n/a^2) : k = 200$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
7	29	0	196	100	2.58	2.58	2.58	0.00	118	100	2.58	2.58	2.58	0.00
7	33	0	196	98	66.94	69.52	70.94	0.02	134	93	62.23	67.25	72.31	0.07
7	34	0	196	96	106.00	111.17	115.80	0.04	138	88	97.05	106.57	121.10	0.12
7	35	0	196	84	101.94	127.22	151.45	0.16	142	75	83.95	115.96	154.61	0.25
7	36	0	196	30	731.33	588.39	1961.30	0.72	146	24	825.00	552.13	2300.54	0.78
7	37	0	196	46	9453.74	4913.07	10680.59	1.14	151	44	9499.45	4811.54	10935.32	1.17
7	38	3	196	39	119.28	175.08	448.92	3.61	155	32	99.91	148.62	464.44	3.68
7	40	8	196	2	241.00	3069.64	153482.00	7.20	163	2	241.00	2851.49	142574.50	7.41
7	41	12	196	25	319.60	1082.80	4331.20	11.95	167	20	227.25	964.22	4821.10	12.15
7	42	16	196	11	1021.64	800.08	7273.45	16.84	171	9	1132.00	777.80	8642.22	16.87
7	43	21	196	30	474.53	2064.84	6882.80	18.21	175	25	409.64	1955.85	7823.40	18.50
7	44	27	196	8	1410.25	2420.79	30259.88	23.17	179	8	1410.25	2313.11	28913.88	23.46
7	45	36	196	60	131.95	217.98	363.30	36.44	183	55	123.58	212.42	386.22	36.53
7	46	51	196	89	123.63	140.61	157.99	51.22	187	88	122.69	139.54	158.57	51.27
7	47	79	196	92	76.99	90.90	98.80	79.08	191	92	76.99	90.50	98.37	79.08

Table 4.17 – Analysis of Termination Condition: $a = 7$; $a.n$ **and** $a.n : Dynamic$

			$a.n$						$a.n : Dynamic$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
7	29	0	203	100	2.58	2.58	2.58	0.00	203	100	2.58	2.58	2.58	0.00
7	33	0	231	99	68.48	70.11	70.82	0.01	231	99	68.48	70.11	70.82	0.01
7	34	0	238	97	107.49	112.60	116.08	0.03	238	100	115.80	115.80	115.80	0.00
7	35	0	245	89	110.85	133.56	150.07	0.11	245	93	126.17	141.23	151.86	0.07
7	36	0	252	35	689.03	626.45	1789.86	0.67	252	67	874.70	973.69	1453.27	0.35
7	37	0	259	50	9497.44	5333.26	10666.52	1.01	259	61	8760.85	5875.34	9631.70	0.86
7	38	3	266	49	144.90	213.10	434.90	3.51	266	51	158.33	219.08	429.57	3.49
7	40	8	280	3	256.00	3794.92	126497.33	6.49	280	5	2099.00	5879.78	117595.60	4.52
7	41	12	287	30	341.67	1428.68	4762.27	11.48	287	43	557.74	1820.68	4234.14	11.06
7	42	16	294	19	809.53	1169.89	6157.32	16.19	294	50	1230.96	1710.84	3421.68	15.69
7	43	21	301	36	467.92	2403.03	6675.08	17.45	301	50	617.30	3142.59	6285.18	15.87
7	44	27	308	15	1322.87	3456.64	23044.27	20.14	308	23	1334.48	4534.61	19715.70	17.16
7	45	36	315	74	182.92	257.32	347.73	36.27	315	89	282.49	317.25	356.46	36.11
7	46	51	322	97	139.81	149.32	153.94	51.05	322	99	149.73	152.15	153.69	51.01
7	47	79	329	98	91.72	97.99	99.99	79.02	329	99	97.05	99.94	100.95	79.01

Table 4.18 – **Analysis of Termination Condition:** $a = 8$**;** No Condition **and d** $kn : k = 6$

			No Condition						$kn : k = 6$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
8	41	0	Max	100	11.84	11.84	11.84	0.00	246	100	11.84	11.84	11.84	0.00
8	42	0	Max	100	5.70	5.70	5.70	0.00	252	100	5.70	5.70	5.70	0.00
8	43	0	Max	100	54.85	54.85	54.85	0.00	258	100	54.85	54.85	54.85	0.00
8	44	0	Max	100	43.41	43.41	43.41	0.00	264	100	43.41	43.41	43.41	0.00
8	45	0	Max	100	285.76	285.76	285.76	0.00	270	61	176.79	216.30	354.59	0.39
8	46	0	Max	100	1183.62	1183.62	1183.62	0.00	276	5	238.00	274.10	5482.00	0.95
8	47	0	Max	88	1823.56	1709.04	1942.09	0.12	282	42	1593.88	1094.91	2606.93	0.67
8	48	0	Max	66	3030.76	2883.67	4369.20	0.40	288	20	4038.10	1806.62	9033.10	1.23
8	49	2	Max	58	2293.97	2189.76	3775.45	2.48	294	22	1326.55	2115.45	9615.68	2.86
8	50	4	Max	61	4514.95	3889.20	6375.74	4.47	300	15	1722.53	2337.12	15580.80	5.25
8	51	6	Max	100	34.34	34.34	34.34	6.00	306	99	19.98	22.84	23.07	6.01
8	52	8	Max	12	4319.67	2452.93	20441.08	9.96	312	3	2236.33	3245.58	108186.00	8.43
8	53	13	Max	18	4956.72	3048.49	16936.06	13.85	318	0	0.00	1499.69	0.00	14.38
8	54	17	Max	53	3004.15	2245.93	4237.60	17.48	324	12	705.58	2465.40	20545.00	15.32
8	55	21	Max	40	3785.95	2917.36	7293.40	21.63	330	4	3313.25	1820.48	45512.00	21.03
8	56	26	Max	35	4720.80	2403.14	6866.11	26.65	336	2	2600.00	3145.67	157283.50	21.58
8	57	32	Max	45	3566.89	2853.07	6340.16	32.55	342	12	1944.50	2322.32	19352.67	30.76
8	58	40	Max	94	1596.10	1570.50	1670.74	40.06	348	37	839.35	1096.67	2963.97	39.91
8	59	49	Max	12	2850.67	2626.53	21887.75	49.91	354	1	1128.00	2446.61	244661.00	44.04
8	60	60	Max	68	829.16	745.45	1096.25	60.92	360	36	460.53	2461.56	6837.67	49.79
			Max											
8	61	86		100	600.32	600.32	600.32	86.00	366	60	285.42	383.05	638.42	86.40
8	62	126	Max	100	176.85	176.85	176.85	126.00	372	94	140.55	161.08	171.36	126.24

Table 4.19 – Analysis of Termination Condition: $a = 8$; $ka^2 : k = 4$ **and** $k(n/a^2) : k = 200$

			$ka^2 : k = 4$						$k(n/a^2) : k = 200$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
8	41	0	256	100	11.84	11.84	11.84	0.00	128	100	11.84	11.84	11.84	0.00
8	42	0	256	100	5.70	5.70	5.70	0.00	131	100	5.70	5.70	5.70	0.00
8	43	0	256	100	54.85	54.85	54.85	0.00	134	95	49.12	53.36	56.17	0.05
8	44	0	256	100	43.41	43.41	43.41	0.00	137	98	41.06	43.21	44.09	0.02
8	45	0	256	57	170.19	210.44	369.19	0.43	140	24	114.25	139.95	583.13	0.76
8	46	0	256	4	232.00	255.04	6376.00	0.96	143	0	0.00	143.00	0.00	1.00
8	47	0	256	38	1206.84	891.76	2346.74	0.73	146	24	1567.25	815.88	3399.50	0.87
8	48	0	256	19	4112.53	1780.72	9372.21	1.24	150	11	3815.27	1406.69	12788.09	1.36
8	49	2	256	19	1348.32	2096.63	11034.89	2.87	153	16	1449.63	1347.49	8421.81	3.24
8	50	4	256	14	1757.93	2109.22	15065.86	5.40	156	12	1646.83	2031.74	16931.17	5.40
8	51	6	256	99	19.98	22.34	22.57	6.01	159	99	19.98	21.37	21.59	6.01
8	52	8	256	3	2236.33	3110.29	103676.33	8.54	162	2	1828.00	2359.28	117964.00	9.43
8	53	13	256	0	0.00	1482.15	0.00	14.34	165	0	0.00	1012.18	0.00	14.96
8	54	17	256	12	705.58	2223.47	18528.92	15.70	168	5	344.40	1673.90	33478.00	16.72
8	55	21	256	3	3218.67	1568.27	52275.67	21.48	171	2	853.50	1395.31	69765.50	21.73
8	56	26	256	1	2774.00	2897.99	289799.00	22.16	175	1	2774.00	2460.67	246067.00	23.27
8	57	32	256	9	1483.33	2060.58	22895.33	31.48	178	4	1717.00	1706.20	42655.00	32.57
8	58	40	256	32	803.59	1039.09	3247.16	39.98	181	21	748.48	789.86	3761.24	41.01
8	59	49	256	0	0.00	1919.50	0.00	46.54	184	0	0.00	1205.59	0.00	50.13
8	60	60	256	26	409.27	2316.61	8910.04	50.80	187	20	365.15	2085.95	10429.75	52.37
8	61	86	256	51	250.49	333.23	653.39	86.51	190	46	243.61	299.21	650.46	86.58
8	62	126	256	91	131.77	152.29	167.35	126.36	193	83	117.23	144.33	173.89	126.68

Table 4.20 – Analysis of Termination Condition: $a = 8$; $a.n$ **and** $a.n : Dynamic$

			$a.n$						$a.n : Dynamic$					
a	n	f_m	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$	tc	$\%f_m$	$\overline{ce_{f_m}}$	$\overline{ce_t}$	$ce_t/\%f_m$	$\bar{f}$
8	41	0	328	100	11.84	11.84	11.84	0.00	328	100	11.84	11.84	11.84	0.00
8	42	0	336	100	5.70	5.70	5.70	0.00	336	100	5.70	5.70	5.70	0.00
8	43	0	344	100	54.85	54.85	54.85	0.00	344	100	54.85	54.85	54.85	0.00
8	44	0	352	100	43.41	43.41	43.41	0.00	352	100	43.41	43.41	43.41	0.00
8	45	0	360	76	206.49	244.67	321.93	0.24	360	76	206.49	244.67	321.93	0.24
8	46	0	368	8	275.13	360.57	4507.13	0.92	368	8	275.13	360.57	4507.13	0.92
8	47	0	376	49	1840.55	1337.98	2730.57	0.57	376	75	2134.41	1965.80	2621.07	0.28
8	48	0	384	23	3662.78	1878.08	8165.57	1.20	384	69	4363.39	3772.02	5466.70	0.54
8	49	2	392	25	1495.00	2552.79	10211.16	2.63	392	44	1738.36	3791.63	8617.34	2.01
8	50	4	400	16	1748.25	2594.68	16216.75	5.10	400	48	3983.29	5112.14	10650.29	3.67
8	51	6	408	99	19.98	23.86	24.10	6.01	408	99	19.98	23.86	24.10	6.01
8	52	8	416	4	3909.00	3779.89	94497.25	7.84	416	11	3876.91	6778.73	61624.82	4.49
8	53	13	424	2	3346.00	1662.07	83103.50	14.23	424	8	4701.13	3346.30	41828.75	12.07
8	54	17	432	16	1100.00	2814.64	17591.50	14.72	432	33	2151.18	4537.04	13748.61	12.07
8	55	21	440	7	2483.00	2244.95	32070.71	20.10	440	25	3024.12	4733.59	18934.36	15.06
8	56	26	448	2	2600.00	3681.11	184055.50	20.18	448	17	4730.65	6305.12	37088.94	13.81
8	57	32	456	16	2137.94	2868.51	17928.19	29.03	456	38	3044.92	5467.78	14388.89	21.00
8	58	40	464	42	885.93	1158.17	2757.55	39.87	464	79	1315.52	1770.45	2241.08	38.20
8	59	49	472	4	1666.75	3181.03	79525.75	40.40	472	11	2474.64	7417.04	67427.64	17.70
8	60	60	480	44	482.75	2884.10	6554.77	46.88	480	56	667.82	3540.50	6322.32	42.56
8	61	86	488	67	322.15	426.75	636.94	86.33	488	82	445.79	481.34	587.00	86.18
8	62	126	496	97	151.77	165.45	170.57	126.12	496	99	165.39	171.72	173.45	126.04

Summary

The analysis of the effects of schemes to determine a suitable termination condition for the evolutionary algorithm have shown that the initial value needs to be sufficiently large to find the first improvement in fitness, after which the best result is to dynamically increase the termination condition when the fitness improves. The simplest calculation for the initial termination condition which gives a sufficiently high value to successfully find the first improvement in fitness is to multiply the array dimension by the number of nodes in the graph. The dynamic increase is achieved by taking the number of generations between the last improvement in fitness and multiplying by 10.

4.10 Conclusions

The experiments presented in this chapter were designed to ensure that the model and evolutionary algorithm defined in this chapter can produce mappings that exhibit good core fault tolerance and determine suitable sizes of array and application process graph for the following experiments.

Initial placement algorithms have been shown to be effective in producing mappings with a good distribution of idle cores across the many-core array and will continue to be used in later experiments to create the zero generation population along with random mappings.

Experiments have demonstrated that the evolutionary algorithm using the optimization objective of fault tolerance was effective in finding mappings with minimum fitness values. The evolutionary algorithm has also been able to find mappings with zero fitness for the optimal combinations of graph sizes and array sizes.

An evolution has been traced from the initial mapping to its final zero fitness mapping demonstrating the ability of the evolutionary algorithm to produce graceful amelioration in terms of improving the fault tolerance of the many-core array.

Experiments have been able to identify which combinations of graph sizes and array sizes are trivial or interesting problems. In general, when there are four or less idle cores there are few unique solutions that are repeated many times so the evolutionary algorithm is able to find a near optimal solution with relatively low computational effort. Also as a general rule, for a given array size, the mapping problem is trivial for graphs with four nodes fewer than the size of the optimal, zero fitness graph.

Experiments in the following chapters will use an array size of 6×6 which finding good fault tolerant mappings is a sufficiently challenging problem for the evolutionary algorithm while being small enough to make run time practical for performing large numbers of tests. An application process graph size of 26 has been found to be optimal for a 6×6, being the largest application process graph that has a fault tolerance fitness of zero.

Chapter 5

Network Power

Having explored the ability of the evolutionary algorithm to find solutions that minimize the objective of core fault tolerance, a second objective of network power is introduced with the aim of finding solutions that simultaneously minimize both the core fault tolerance and the network power objectives.

The fault tolerance objective of the previous chapter has the effect of evenly distributing idle cores amongst the processing cores, which has the effect of pushing processing cores away from each other. The power objective of this chapter will have the effect of bring processing cores closer together to reduce the length of paths between pairs of communicating cores. The effects of the fault tolerance and power objectives are conflicting which gives an interesting solution space for multi-objective searches, compared to objectives which are closely aligned.

Adding a second objective requires the expansion of the evolutionary algorithm to accommodate multiple objectives, which in turn requires a change to the evolutionary cycle. *ComPair* is added to the many-core model, a sorting method for multi-objective solution spaces (see Subsection 5.4.1). Additional genomic representations are also added to the evolutionary algorithm and an exploration of the parameters to control the behaviour of the multi-objective evolutionary algorithm is carried out to establish the best combination of genomic representation and evolutionary algorithm parameters to make the multi-objective evolutionary algorithm as efficient as possible.

The calculation of the power objective requires the introduction of a new concept, the *ComPair*, that is added to the many-core model.

When a search space has two or more objectives the solution space has a *Pareto Front*, at set of solutions that are all considered to be equally good.

Problem Description
Given a fault free many-core array with R rows and C columns and an application process graph with V_a processes where $V_a < RC$, arrange the processing cores and idle cores to minimize the cost of task migration in the event of the failure of a processing core while simultaneously minimising the overall amount of power consumed by network traffic. Network power will be defined explicitly in Section 5.3.

5.1 Communicating Core Pair (ComPair)

A pair of processes from an application process graph connected by an edge will cause data traffic to flow between the cores in the many-core array that the processes are mapped to. The two cores in the many-core array where the processes are located are referred to as a *communicating core pair* or *ComPair*. Since the application process graph is a directed graph each ComPair will have a *source node* from where data traffic originates and a *target node*, where the data is received for use by the target process. For this work, the assumption is made that traffic will use a path of minimal length between the source and the target of the ComPair subnet.

The ComPair is introduced in this chapter to support the calculation of the power metric presented in Section 5.3. ComPairs will also be used extensively in the calculation of traffic metrics presented in Chapter 6.

Definition: Given an edge of an application process graph and the source node and target node of the edge, a ComPair consists of the two cores of the many-core array to which the source node and target node have been mapped. The ComPair has a *subnet* that consists of all the routing nodes and links between and including the node where the source core is located and the node where the target core is located.

Between the source and target nodes of a ComPair there will one or more *minimal length paths*, which are paths where each step in the path gets closer to the target node. In this work metrics are based on minimal length paths.

Once an application process graph has been mapped to a many-core array there will be as many ComPairs in the array as there are edges in the application process graph. Note that a physical link of the many-core array may carry traffic from multiple ComPairs.

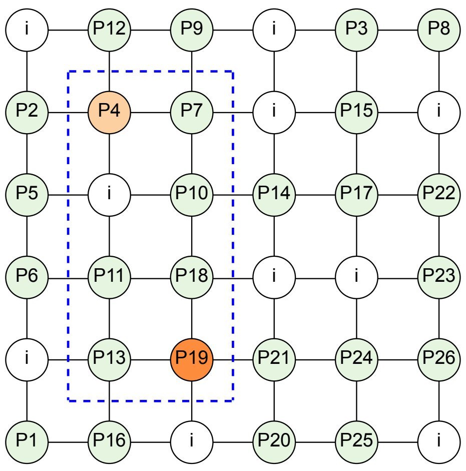
The properties of a ComPair are given in Table 5.1 and those of a ComPair subnet inTable 5.2.

As an example, two ComPairs that are created by mapping the application process graph of Figure 4.2 to a 6×6 many-core array, are illustrated in Figure 5.1.

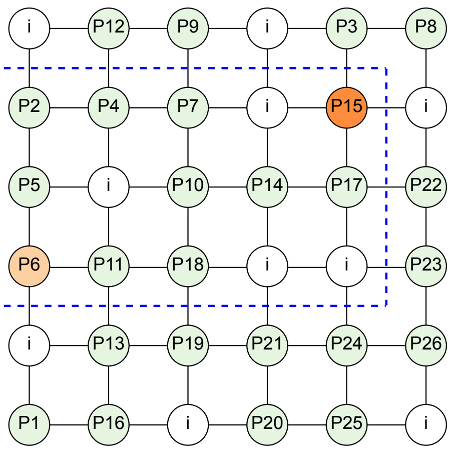
Note that when discussing ComPairs, they will usually be referred to by the symbol Q, taken to mean any unspecified ComPair. When it is necessary to associate a ComPair with an edge in the application process graph the ComPair will be referred to by the symbol Q_e, where e represents the specific edge of the application process graph that generates the ComPair in the many-core array.

The number of rows and columns in a ComPair subnet can be calculated from the source and target locations using equations 5.1 and 5.2.

$$R_q = |t_r - s_r| + 1 \tag{5.1}$$



(a) Subnet for the ComPair with source node P4 and target node P19



(b) Subnet for the ComPair with source node P6 and target node P15

Figure 5.1 – ComPair and subnet examples. Source nodes are coloured light orange, target nodes coloured dark orange and the subnets are bounded by a dashed blue line.

$$C_q = |t_c - s_c| + 1 \tag{5.2}$$

The number of links in a ComPair subnet is given by Equation 5.3.

$$Q_l = R_q(C_q - 1) \ + (R_q - 1)C_q \tag{5.3}$$

The number of shortest length paths from the source node to the target node of a ComPair subnet that is fault free is given by Equation 5.4 as explained in Section 6.7. If there are faulty links in the subnet then the number of links can be calculated using the recursive algorithm described in Section 6.7.

$$P_q = \frac{(R_q - 1)(C_q - 1)!}{(R_q - 1)!(C_q - 1)!} \tag{5.4}$$

The *path length* of a ComPair in terms of the number of links on a shortest length path between the source and target nodes of is given by Equation 5.5.

$$P_q = (R_q - 1) + (C_q - 1) \tag{5.5}$$

Table 5.1 – ComPair Properties

Name	Description
Q_e	The ComPair representing edge e of the application process graph
Q_s	The source node of ComPair Q
Q_{loc_s}	The location of the source node of ComPair Q within the array
s_r	The row of the source location
s_c	The column of the source location
Q_t	The target node of ComPair Q
Q_{loc_t}	The location of the target node of ComPair Q within the array
t_r	The row of the target location
t_c	The column of the target location
R_q	The number of rows of nodes in ComPair Q
C_q	The number of columns of nodes in ComPair Q
Q_l	The path length of the ComPair in terms of the number of links on the shortest path between the source and target nodes
P_q	the number of shortest length paths between the source and target nodes
Q_d	The network load for the ComPair taken from d_n for the edge e_n of the application process graph that the ComPair represents

Table 5.2 – ComPair Path and Link Properties

Name	Description
$\mathcal{P}_q$	The set of unique shortest length paths between the source and target nodes of the ComPair Q
P_q	The number of shortest length paths in the set $\|\mathcal{P}_q\|$
p_n	The n^{th} path of ComPair Q
$\mathcal{L}_{(q)}$	The set of links in the subnet of ComPair Q
L_q	The number of links in set $\|\mathcal{L}_{(q)}\|$ in the subnet of ComPair Q
l_n	The n^{th} link of ComPair Q

5.2 Link Criticality

The many-core array architecture described in Section 4.2 uses a lattice of routers for transmission of traffic through the network. Traffic from the source to the target of a ComPair will be transmitted as a number of packets which will pass through a series of routers, each with a buffer where received packets are temporarily stored before being forwarded to the next router.

Link criticality is a method of categorizing the effect that the failure of a physical link can have on traffic in a ComPair. The classification of a link is specific to a ComPair, so can have different criticalities for each ComPair subnet that it is part of. A link is classified as one of the three following types defined by the effect its failure will have on the transmission of data in the ComPair subnet of which is is part:

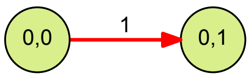
- Critical Links
- Significant links
- Normal links

Each of these three types of links will be explained below with reference the the graphs in Figure 5.2. When developing the metrics it will become apparent that the number of paths that use each link will be important in the calculation of the metric. The edges of the graphs in Figure 5.2 are labelled with the number of paths that use each link.

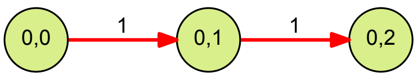
Critical Link
A *critical link* is a link where all paths from the source to the target pass through the link, such that a fault will cause complete failure of communication. Figures 5.2a and 5.2b illustrate subnets with critical links, shown in red, where the failure of a single link causes complete communication failure. Traffic that passes through a critical link is described as *critical traffic*.

If, for ComPair Q, there are P_q paths between the source and target nodes and there are l_{n_p} paths through link l_n, then link l_n is defined as a critical link by Equation 5.6, i.e. when the number of paths through the link is equal to the total number of paths between the source node and target node of the ComPair.

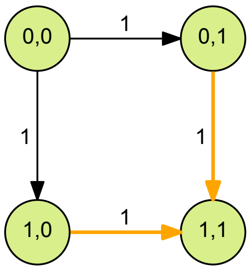
$$l_n \text{ is critical } \Leftrightarrow l_{n_p} = P_q \tag{5.6}$$



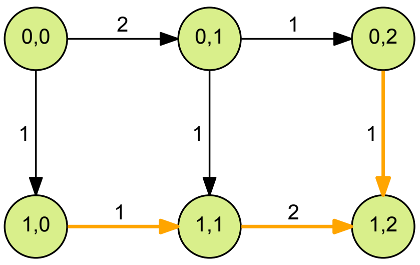
(a) Critical Link of length 1



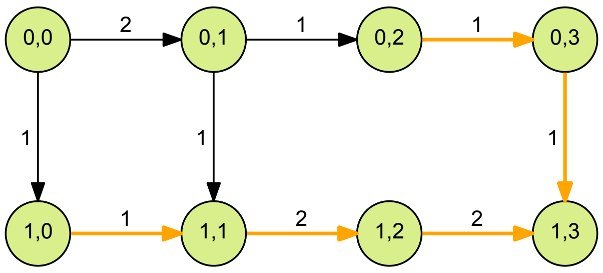
(b) Critical Link of length 2



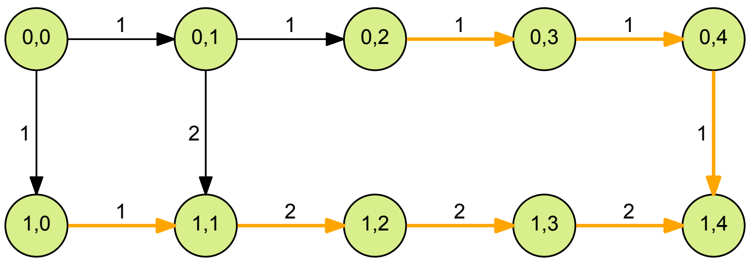
(c) Network with 2 Significant Links



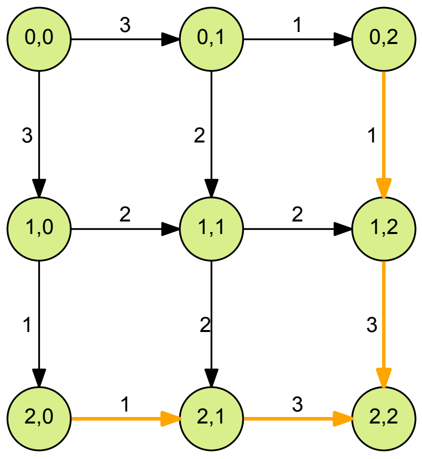
(d) Network with 3 Significant Links



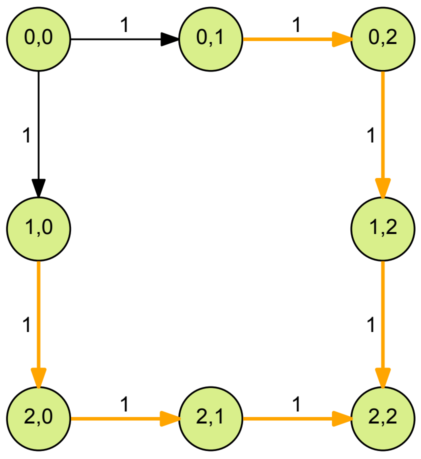
(e) Network with a single Link fault and 5 Significant Links



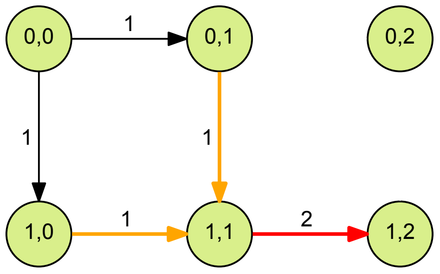
(f) Network with a 2 Link faults and 7 Significant Links



(g) Network with 4 Significant Links



(h) Network with a 4 Link faults and 6 Significant Links



(i) Network with 2 Link faults, 2 Significant Links and 1 Critical Link

Figure 5.2 – Networks illustrating Critical and Significant links

Where:

P_q	=	The total number of paths of the ComPair subnet.
l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
l_{n_p}	=	The number of paths through link l_n of ComPair Q.
Q_l	=	The number of links in the ComPair subnet.

Significant Link

A *significant link* is a link which is the only link from its source node and is a link where only a (proper) subset of paths from the source to the target pass through the link i.e. not all paths use pass through the link, as defined by Equation 5.7:

$$l_n \text{ is significant } \Leftrightarrow (l_{n_p} < P_q) \wedge (l_{s_{out}} = 1) \qquad (5.7)$$

Where:

P_q	=	The total number of paths of the ComPair subnet.
l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
l_{n_p}	=	The number of paths through link l_n of ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
$l_{s_{out}}$	=	The number of outbound edges of source node l_s
l_s	=	The source node of link l

A faulty significant link will sever the paths that use the link while still leaving one or more viable paths that do not use the faulty link. Traffic that passes through a significant link is described as *significant traffic*. A fault in a significant link will potentially cause some packets to be lost where packets are stored in the buffers of routers that have become disconnected from the target node due to the appearance of the fault. Figures 5.2c and 5.2d illustrate subnets where there are significant links shown by orange arrows. Packets will be lost when the routing node attempts to transmit packets through the link after the link has failed.

A significant link exists when it is on the only path from its routing node to the target node of the ComPair but the path it is on is not the only path between the ComPair. Failure of a significant link will create a dead end path of arbitrary length causing the loss of packets that travel down the "dead end". A router only has knowledge of the status of directly connected links so each router will continue to route packets down the dead end until the loss of packets is detected and propagated back up the path until the router at the head of the dead end path is reached. The router at the head of the dead end will eventually establish that packets are not being received through that path so will modify its behaviour to only send packets down the alternative path, thereby maintaining communication. The loss of a significant link will redistribute traffic to other links which may in turn create bottlenecks and reduce the overall performance of the system. In some case, for example in figures 5.2 (c), (d), (e), (f), (h) and (i), bit not (g), the loss of a significant link will have the effect of changing the criticality of other links from significant to critical, making the network vulnerable to further link failure.

Notice that a ComPair subnet will contain as least one critical link or two significant links. This can be confirmed by considering the target node: if there is only one inbound link to the target node then it must be a critical link, if there are two inbound links to the target node then they must both be significant links since failure of either link will cause packet loss.

Normal Link

A *Normal Link* is a link whose failure does not cause the loss of transmission of packets, more specifically it is a link whose routing node has more than one outgoing link to the target of the ComPair. A normal link is defined by Equation 5.8:

$$l_n \text{ is significant } \Leftrightarrow (l_{s_{out}} = 2) \tag{5.8}$$

Where:

l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
$l_{s_{out}}$	=	The number of outbound edges of source node l_s
l_s	=	The source node of link l

Traffic that passes through a normal link is described as *normal traffic*. Figures 5.2c to 5.2i illustrate subnets where normal links are shown by black arrows.

Failure of a normal link has no negative effect on the transmission of packets because the routing node it is connected to has a choice of two links for the transmission of packets. Given that the router has knowledge of the status of its directly connected links it is able to transmit packets along the remaining healthy link when one link fails. Each of the two links are on distinct, alternative paths from the source to the destination.

5.3 Network Power Metric and Objective

Communication traffic is a major contributor to power consumption in a many-core array. The total traffic flow within the many-core array can be used as an approximation for power consumed by communication traffic. This section defines a simple network power metric and objective.

Problem Description
Given a mapping of an application process graph, with E_g edges, to a many core array and the corresponding E_g ComPairs that this mapping creates, define a metric that gives an approximation to the power consumed by the traffic generated by a ComPair and an objective that gives a measure of the total power consumed by the network traffic for whole many-core array.

The power consumed by traffic between a ComPair, for a given router/NoC architecture, is a function of both the traffic volume and the distance in terms of routing nodes and links that the traffic has to traverse between the source and target nodes. Reducing the total network traffic of all ComPairs will reduce the power consumption of the many-core array. The volume of traffic between each ComPair is determined by the application and cannot be influenced by the many-core system whereas the distance the traffic has to travel is dependent on the relative position of the source and target nodes of each ComPair, which is under the control of the many-core system through task migration and remapping.

At this stage in the development of the metrics, a simple calculation based only on the ComPair path length is preferred to a more accurate calculation involving the path length and the traffic volume taken from the application process graph. A more accurate power metric and objective will be developed in Chapter 6.

5.3.1 Distance Between Communicating Core Pairs

As a first approximation for power consumption due to network traffic, the power metric for a simple ComPair is defined as the path length of the ComPair as defined by Equation 5.5 in Section 5.1.

The distance between a ComPair is the rectilinear distance between the source and target nodes and, assuming minimal length paths, represents the number of links the data will travel through from the source node to the target node. Faults on paths between the source and target have no effect on the length of the path that the data will travel along, providing the faults do not sever all paths between the source and target. For a ComPair q with source location $Q_{(s_r,s_c)}$ and target location of $Q_{(t_r,t_c)}$ the *ComPair Path Length* metric (i.e the distance between the source node and the target node) $Mcppl_q$ is given by Equation 5.9.

$$Mcppl_q = |(s_r - t_r)| + |(s_c - t_c)| \tag{5.9}$$

5.3.2 Simple Power Objective

The power objective is to minimize the total power consumption across the whole network. A power consumption of zero is assigned to the power objective when the source and target of a ComPair are adjacent, equivalent to the ComPair path length minus 1. If the source and target nodes of every ComPair are adjacent then the value for the mapping will be zero. The power objective is given in Equation 5.10.

$$Jsp = \sum_{q=0}^{E_g}(Mcppl_q - 1) \tag{5.10}$$

5.4 Many-Core Evolutionary Algorithm

To enable multi-objective search of the solution space, the evolutionary cycle of Chapter 4 has to be extended and additional genomic representations introduced.

The individual elements of the evolutionary algorithm, listed below, will be discussed in the remaining sections of this chapter.

- Pareto Front Sorting
- The Evolutionary Cycle
- Phenome and Genome

5.4.1 Pareto Front Sorting

Pareto front sorting is a method of dividing the individuals of a population into sets using the values of the objectives, such that the individuals in each set are considered to be equivalent and the sets themselves form a hierarchy. The individuals in each set are said to belong to the same Pareto front while the Pareto fronts can be arranged in a ordered list which are numbered, with the best individuals in the *first Pareto front* also referred to as the *Pareto optimal front* (*Pf1*).

5.4.1.1 Definition of Pareto Front

Sorting a population into a series of Pareto front relies on the concept of dominance between individuals in the population. Deb (1999) [175] credits Steuer (1986) [176] with the following definition of domination:

For a problem having more than one objective function (say, $f_j, j = 1, \ldots, M$) and $M > 1$), a solution $x^{(1)}$ is said to dominate the other solution $x^{(2)}$ if both the following conditions are true:

1. The solution $x^{(1)}$ is no worse (say the operator $\prec$ donates worse and $\succ$ denotes better) than $x^{(2)}$ in all objectives, or $f_j(x^{(1)}) \nprec f_j(x^{(2)})$ for all $j = 1, 2, \ldots, M$ objectives.
2. The solution $x^{(1)}$is strictly better than $x^{(2)}$ in at least one objective, or $f_j(x^{(1)}) \succ f_j(x^{(2)})$ for at least one $j \in \{1, 2, \ldots, M\}$.

The concept of dominance is then used to define Pareto fronts. A non-dominated Pareto front is a set of individuals, all of which are not dominated by any other individual. If the non-dominated set is removed from the population of individuals and designated *Pf1*, then the new reduced population will have a non-dominated set of individuals, which can be removed and labelled *Pf2*. This procedure can be repeated until all individuals of the population have been allocated to a Pareto front.

The Pareto fronts so formed have a strict hierarchy where the individuals in *Pf1* are better solutions than those in *Pf1* and so on. Individuals in *Pf1* are considered better than those in *Pf2* because each individual in *Pf2* is dominated by at least one individual in Pf1. Within each Pareto front the individuals are considered to be equivalent, that is one individual cannot be said to be better than another, based on the values of the objectives. The individuals within a Pareto front are not all identical and there may be attributes of the individuals, other than the objective values used in Pareto front sorting, that can be used to differentiate the individuals.

A more formal definition of Multi-Objective problems, Pareto dominance, Pareto optimality, Pareto optimal set and Pareto optimal front can be found in Wang et al. (2014) [177].

5.4.1.2 Significance of Pareto Fronts

Pareto fronts are significant because they allow a set of individuals from a population to be identified as better than the rest of the population. In the case of an evolutionary algorithm obtaining the Pareto front of each generation is an important step in the evolutionary cycle because the Pareto optimal front represents the best individuals from that generation. It is important for the evolutionary algorithm to know which are the best solutions so that they can be used to create the next generation via mutation.

Because the evolutionary algorithm requires the Pareto front to each generation to be identified, Pareto front sorting is carried out once for each generation, making the efficiency of Pareto front sorting an important topic. With a population of 100 and a 1000 generations Pareto front sorting will be carried out at least 1000 times on a population of 100 individuals during the evolutionary process.

Deb et al.'s paper of 2002 [178] presents the NSGAII sorting algorithm that requires in the order of $O(MN^2)$ comparisons, where M is the number of objectives and N is the number of individuals.

Jensen's (2003) [179] algorithm improves upon Deb et al.'s NSGAII run-time with a run-time of $O(N(\log N)^{M-1})$ in the worst case. However, Jensen makes an assumption that no two individuals will have the same value for any of the objectives. However, when two individuals share a value for one of the objectives the Pareto fronts calculated by Jensen's algorithm can be erroneousFortin and Parizeau [180, 181].

Fang et al., 2008 [181], developed a data structure that they refer to as a dominance tree to avoid making unnecessary comparisons between individuals, with an estimated run-time complexity of $O(MN \log N)$ with $O(MN^2)$ in the worst case.

Fortin and Parizeau (2013) [180] correct the problems and generalize Jensen's algorithm for the case where individuals share an objective value. The number of comparisons required for Fortin and Parizeau's algorithm are given as $O(N(\log N))$ in the best case, when every objective for every individual is identical, due to sorting the data as preparation for the Pareto front sorting, $O(N(\log N)^{M-1})$ in the general case and $O(MN^2)$ in the worst case.

Buzdalov et al. (2015) [182] proposed a number of modifications to Fortin and Parizeau's algorithm resulting in a proven number of comparisons of $O(N(\log N)^{M-1})$ in the worst case.

5.4.2 Evolutionary Cycle

In this chapter the many-core evolutionary algorithm has been extended to a multi-objective evolutionary algorithm, the implementation of which, allows for the selection of the objectives to be controlled by parameters. Note that, if only a single objective is selected, as will be the case for early experiments, then the implementation will function as a single objective evolutionary algorithm.

5.4.2.1 The Cycle

The evolutionary cycle is implemented as a series of populations, summarised in Table 5.3.

Table 5.3 – Evolutionary Algorithm Populations

Population	Description
Initial	The initial population which is provided to the evolutionary cycle as a starting point for evolution.
Generation Zero	The first primary population created by evaluating and sorting the initial population and is the starting point for evolution.
Primary	The population that is the product of each evolutionary cycle (and the starting point for the next cycle).
Intermediate	The population created through cloning and genetic manipulation of the primary population of the previous evolutionary cycle, possibly with the addition of a number of new randomly created individuals.
Fitness Evaluated	The intermediate population after the objectives have been evaluated.
Sorted	The fitness evaluated population sorted using a Pareto front sorting algorithm.
Pareto Front Zero (Pf0)	A population that, at the end of each cycle, is guaranteed to contain the Pareto front obtained from all individuals from all generations.

The extended multi-objective evolutionary algorithm is implemented as the *evolutionary cycle* illustrated in Figure 5.3 the additional population .

5.4.2.2 Initial Population

The initial population is composed of a combination of engineered mappings (discussed in Subsection 4.8.3) and random mappings. The initial population is located in the bottom left of the Figure 5.3. Once the individuals of the initial population have been created, their metrics are evaluated and the population is sorted to produce the generation zero population, the first primary population. The size of the initial population is governed by the population size parameter, P_z.

5.4.2.3 Generation Zero Population

The generation zero population is the result of the evaluation and sorting of the initial population and is the first primary population which is used as the starting point of the evolutionary process. The name generation zero is given to this population to distinguish it from all other primary populations that are created through the evolutionary process.

5.4.2.4 Primary Population

The *primary population* is the population produced at the end of each evolutionary cycle and is also the population that is produced on completion of the evolutionary algorithm. The primary population is the population that is used as the starting population for each evolutionary cycle. In the Figure 5.3 the primary population is represented as the green box in the bottom right of the diagram. Outside of this section the primary population will normally be referred to simply as *the population*.

The size of the primary population is determined by the *population* parameter, P_z, which is the minimum number of individuals copied from the sorted population, to create each new primary population. If the population Pf0 (see Subsubsection 5.4.2.8) becomes larger than P_z then the primary population is allowed to grow dynamically so it contains at least the individuals of Pf0.

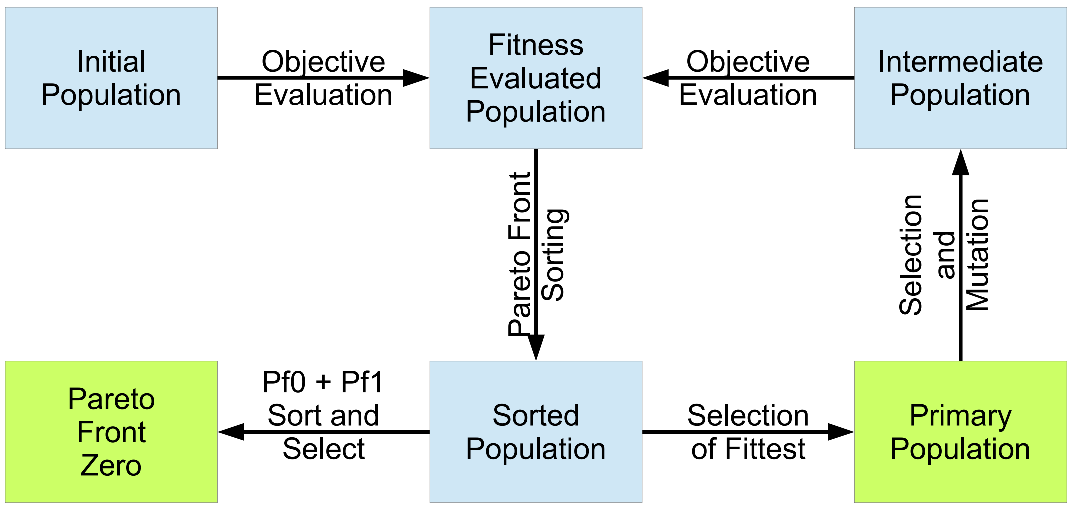


Figure 5.3 – Evolutionary Algorithm Cycle

5.4.2.5 Intermediate Population

The intermediate population is created by selection and mutation of individuals from the primary population. The creation of the individuals in the intermediate population, illustrated in Figure 5.4, is controlled by the following parameters:

Elite Individuals

The elite parameter, P_e , is used to specify the minimum number of fittest individuals of the input population that are copied directly to the new intermediate population. If the number of individuals in *Pf0* is more than the value of the elite parameter, then all the individuals of *Pf0* are added to the intermediate population.

Descendants

The number of individuals specified by the *Parents* parameter, P_p, are selected from the input population. The *Descendants* parameter, P_d, determines how many times each parent is copied and mutated before being added to the intermediate population.

Novel

The novel parameter, P_n, determines how many new randomly generated individuals are added to the intermediate population.

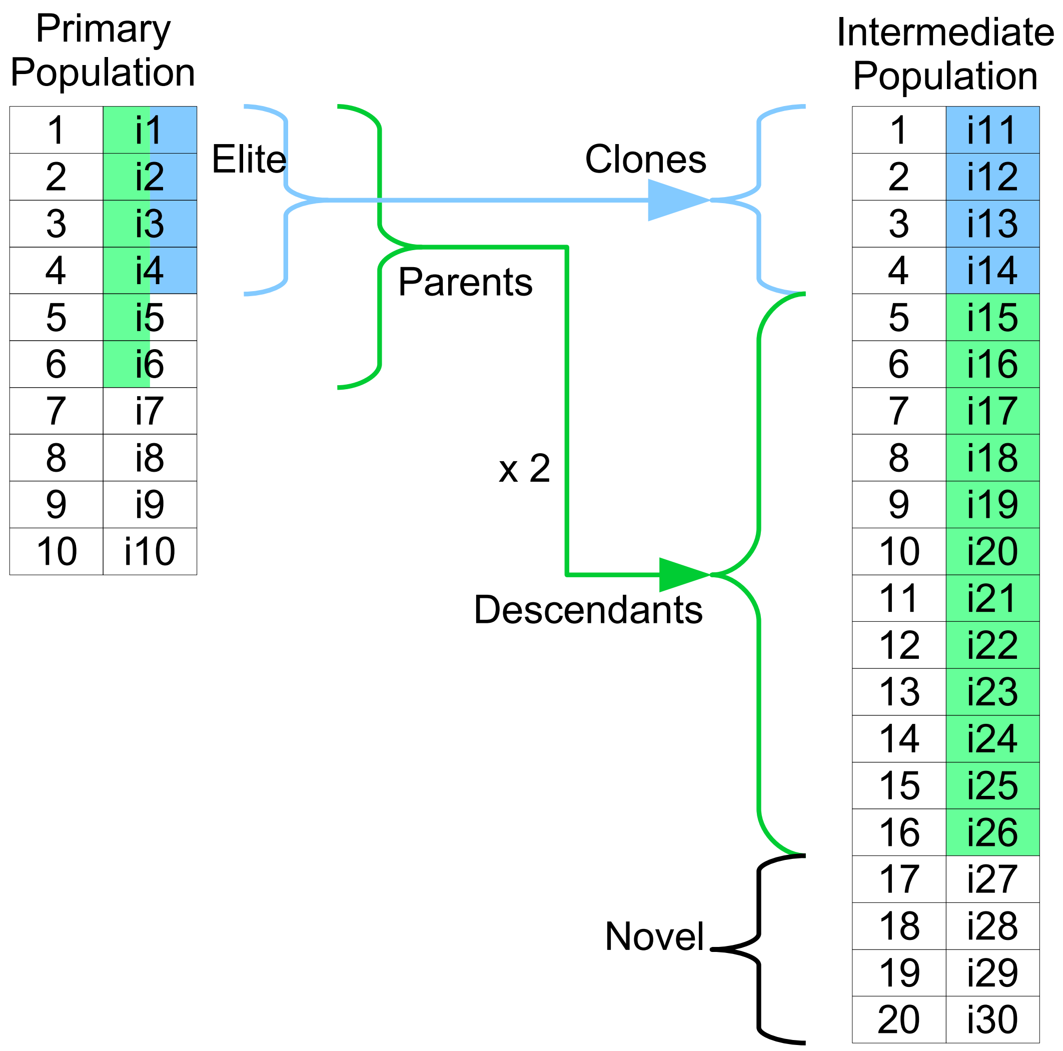


Figure 5.4 – Population Selection and Mutation, using 4 elite individuals, 6 parents mutated twice each and 4 novel individuals, to create an intermediate population of size 20.

The size of the intermediate population is determined by the evolution parameters using Equation 5.11. Since the number of individuals regarded as elite is the greater of the elite parameter or the number of individuals in *Pf0* , the intermediate population is not limited the intermediate population to be the same size as the primary population.

$$Q_z = \begin{cases} P_e + (P_p \times P_d) + P_n, & \text{if } P_e \geq |Pf0| \\ |Pf0| + (P_p \times P_d) + P_n, & \text{otherwise} \end{cases} \tag{5.11}$$

5.4.2.6 Fitness Evaluated Population

Once the intermediate population has been created, the objectives for each individual need to be evaluated in preparation for sorting the population. The *fitness evaluated population* contains the same individuals as the intermediate population with the evaluated objectives added to each individual.

Some of the objectives, for example excess traffic, require detailed analysis of the volume of traffic that flows through each link in each ComPair making evaluation a lengthy process.

5.4.2.7 Sorted Population

Pareto front sorting is a method of dividing the individuals of a population into sets using the values of the objectives, such that the individuals in each set are considered to be equivalent and the sets themselves form a hierarchy. The individuals in each set are said to belong to the same Pareto front while the Pareto fronts can be arranged in an ordered, numbered list with the best individuals in the first Pareto front denoted by *Pf1*.

The Pareto front sorting algorithm takes as input the fitness evaluated population and produces as output the sorted population.

The sorted population is used as input into the process that selects the fittest individuals to produce the primary population. The process of selecting and copying individuals to the primary population must ensure that when the Pareto front is complete the individual remain sorted. This ensures that the selection and mutation process that creates the intermediate population does not need to sort the population of individuals.

5.4.2.8 Pareto Front Zero (Pf0)

In the evolutionary cycle it is possible that points on *Pf1* of one generation may be omitted from the population of the next generation even though they would qualify for inclusion of Pareto optimal front of the new generation. This happens when the number of points on the Pareto optimal front is greater than the number of elite individuals, P_e, then some of the points on the the Pareto optimal front will be lost when the elite individuals are copied to the intermediate population. It is also possible that after a large number of generations the number of points on the Pareto optimal front exceeds the primary population size.

As it is undesirable to lose any individuals from the Pareto optimal front without them first being dominated by a new individual a separate population contains all the points from the Pareto optimal front regardless of how many points are on the front. This additional population has been given the name *Pareto Front Zero (Pf0)* to distinguish it from Pf1 of each individual generation.

Pf0 is maintained as an additional population to the primary population. Once the intermediate population has been sorted and the individuals on the Pareto optimal front are copied into Pf0. Pf0 is then subjected to a Pareto sort to determine which individuals are still in Pf1. Only the individuals on the Pareto optimal front of Pf0 are retained. All other individuals are removed from Pf0.

5.4.2.9 Population Evolution Parameters

The size of the primary population and the intermediate population are determined by the properties and evolution parameters given in Table 5.4.

Table 5.4 – **Population Properties and Parameters**

Parameter	Description
P	The primary population
Q	The intermediate population
P_z	Primary population size
Q_z	Intermediate population size
P_e	Number of elite Individuals
P_p	Number of parents
P_d	Number of descendants
P_n	Number of novel Individuals
P_v	Number of non-viable individuals created before abandonment of a process that creates new individuals

5.4.3 Phenome and Genome

In addition to the genome using a direct phenomic representation described in Subsubsection 4.8.2.2, this section describes an additional two genome representations are added here which are then compared to original representation.

5.4.3.1 Genome - String Representing Absolute Position

Genomic Representation
This representation is based on the typical genomic representation that uses a string of bits to represent the genes. For programming simplicity, integers are used in preference to a string of bits to represent a gene, so that the genome is represented by an ordered list of integers. The length of the list, the number of genes in the genome, is the number

of processes in the application process graph, V_p, so that each gene in the genome represents a specific process from the application process graph. The processing nodes of the application process graph are numbered sequentially, so each gene in the list will represent the corresponding process in the list of processing nodes, ensuring that each process is represented exactly once in the genome. The alleles of the genetic alphabet represent the position of the gene in the phenome so consist of the set of numbers $\{1, 2 \ldots, \mathcal{P}\}$, where $\mathcal{P}$ is given by Equation 5.12.

$$\mathcal{P} = |P| \ = P_{rows} \cdot P_{cols} \tag{5.12}$$

Phenomic Representation

The value of each gene is an allele from the genetic alphabet which, as defined above, are integers that represent each position in the array. The alleles, through expression of the genome, translate to a two dimensional Cartesian coordinate within the phenome. Given a phenome P composed of the set of phenes $\{p_1 \ldots p_{\mathcal{P}}\}$, the Cartesian coordinates of the phene in the phenome can be calculated from the gene value using the formulae given in equations 5.13 and 5.14.

$$c = g_n \pmod{P_{rows}} \tag{5.13}$$

$$r = \frac{g_n - c}{P_{cols}} \tag{5.14}$$

An example of mapping 12 process to a 4×4 array is illustrated in Figure 5.5.

Gene	Allele	Node
1	1	0,0
2	3	0,2
3	4	0,3
4	5	1,0
5	6	1,1
6	7	1,2
7	10	2,1
8	11	2,2
9	12	2,3
10	13	3,0
11	14	3,1
12	16	3,3

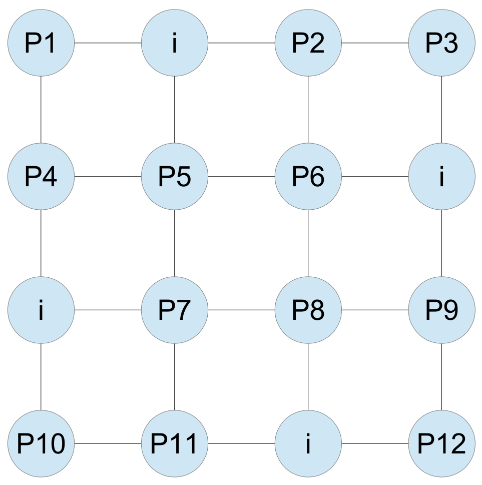


Figure 5.5 – Genome Absolute Position Mapping

Other than for the special case where the number of processes in the application process graph is the same as the number of nodes in the many-core array, the length of the gene will always be less than the number alleles in the genetic alphabet. An allele can appear in a single gene, in multiple genes or in none of the genes. The value of the each gene is interpreted as the *preferred target* phene of the phenome. If the target phene is unavailable because it either has a failed status or already has a process mapped to it then a search is made for the nearest idle core which ensures that expression of the genome results in a valid phenome.

Permutation
Permutation, in this representation, does not affect the alleles represented in the genome, it can only reorder those alleles that are in the original genome. The location of idle cores and processing cores remain unchanged, while the processes are rearranged among the processing cores. This will limit the genetic diversity reducing the ability of this genetic operator to produce only a fraction of possible mappings.

If all alleles are unique then, selecting and swapping two genes will cause corresponding two phenes in the phenome to be swapped, giving good correlation.

When an allele appears more than once in the genome the first allele will map to the preferred target phene. Each time the same allele appears in the genome the target phene will not be available, having already been used. The allele will therefore be mapped to the nearest available phene which will cause a larger change in the phenome.

If all genes have the same allele, then swapping a pair of genes will have no effect giving poor correlation.

If only one allele is represented in the genome then all but one phene will be relocated away from the preferred target which is likely to produce very poor correlation between the genome and the phenome.

Permutation has poor correlation and poor genetic diversity, making this an unsuitable genetic operator.

Mutation
Mutation is implemented by randomly selecting a gene and then randomly selecting a new allele for the selected gene. If the new allele was not present in the original genome, then the expression of the mutated genome will produce a phenome where a process has been swapped with an idle core, giving good correlation. If the new allele already exists in the genome then the expression of the mutated genome is likely to produce a phenome exhibiting multiple changes, giving poor correlation.

Figure 5.6 shows how a single mutation, where the gene for process 8 has been changed from 11 to 14, results in process 8 being located at $(3, 1)$ which in turn causes process 11 to be relocated to position $(3, 2)$, resulting in three phenes being altered.

Crossover
Crossover can be implemented by selecting two parents and randomly selecting sections of each genome to combine. The difficulty with this genetic operator is that there will be little evidence of either parent in the resulting phenome. The first block of alleles will produce a recognisable correspondence with the phenome for the parent, however each subsequent block of genes will become increasingly "scrambled" in the phenome as duplicate alleles

Gene	Allele	Node
1	1	0,0
2	3	0,2
3	4	0,3
4	5	1,0
5	6	1,1
6	7	1,2
7	10	2,1
8	14	2,2
9	12	2,3
10	13	3,0
11	14	3,1
12	16	3,3

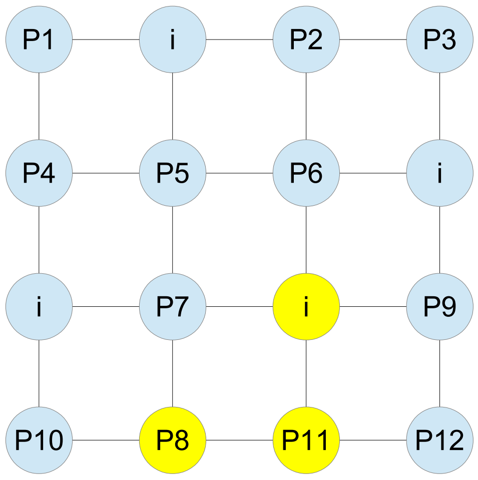


Figure 5.6 – Genome Absolute Position Mutation

are discovered and relocated giving particularly poor correlation. For this reason crossover is not considered to be a reasonable genetic operator to use.

Correlation
As noted above the correlation between the genome and phenome is not fixed, but is dependent on the diversity of alleles in the genome, the position of the gene that is changed and the genetic operator used to produce new individuals. Consideration of the above genetic operators leads to the conclusion that mutation will maintain the best diversity of alleles and the best correlation between the genome and phenome.

Even when mutation is used, many mutations will have poor correlation between the genome and phenome.

5.4.3.2 Genome - String Representing Relative Position

Genomic Representation
The string representation with relative positioning has the same length of the genome and uses the same alphabet of alleles that are used by the absolute positioning genome. The difference between the two representations is in the interpretation of the values of alleles. In the relative positioning representation, the alleles represent the number of available positions from the previously placed process to the placement of the current process.

Phenomic Expression
For the first gene in the genome, the phenome is scanned, starting from position $(0, 0)$ of the phenome, for phenes containing the idle phene which can be replaced by a process phene. Counting the first idle phene as 1, idle phenes are counted until the count reaches the number corresponding to the value of the allele in the gene. The phenome is scanned using a left to right raster scan and when the last phene has been scanned the scan wraps around to continue at $(0, 0)$. The phene, thus found, has the idle phene replaced by the

Gene	Allele	Node
1	1	0,0
2	2	0,2
3	1	0,3
4	1	1,0
5	1	1,1
6	1	1,2
7	3	2,1
8	1	2,2
9	1	2,3
10	1	3,0
11	1	3,1
12	2	3,3

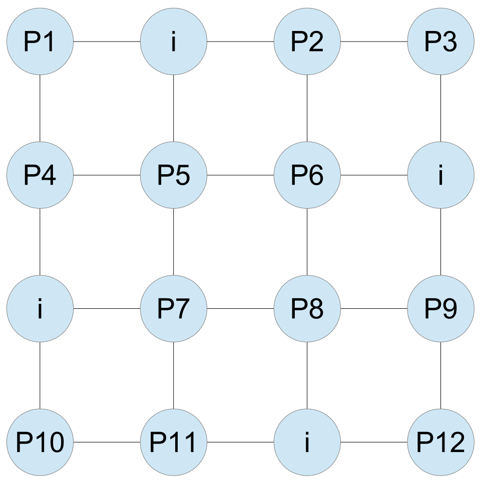


Figure 5.7 – **Genome Relative Position Mapping**

process phene represented by the position of the gene in the genome.

After the first gene is expressed, the next gene is similarly expressed but with the start position in the phenome being the phene immediately after the phene that has just been replaced with a process.

A change of allele of a gene will change the place where the process is placed within the phenome. It will also affect the placement of all remaining processes which results in all genetic operators producing poor correlation. Figure 5.7 illustrates a relative position genome that produces the same phenome as the absolute position genome in Figure 5.5.

Gene	Allele	Node
1	1	0,0
2	2	0,2
3	1	0,3
4	1	1,0
5	1	1,1
6	1	1,2
7	3	2,1
8	4	2,2
9	1	2,3
10	1	3,0
11	1	3,1
12	2	3,3

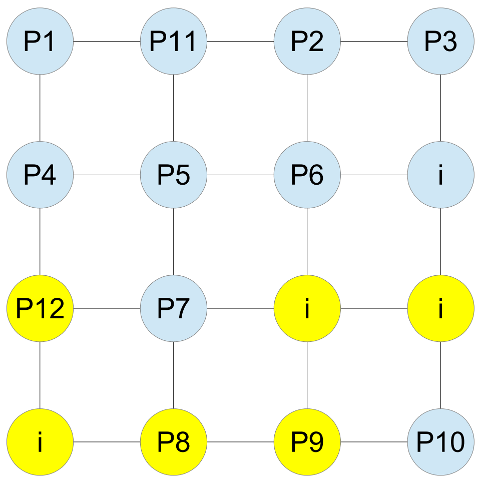


Figure 5.8 – **Genome Relative Position Mutation**

Genetic Operators

As explained above, all genetic operators produce poor correlation due to the fact than any change to a gene in the genome will affect the expression of all remaining genes in the genome. Consequently there is little to choose between the genetic operators. To allow a comparison with the absolute position genome the mutation operator will be used and will function identically to the mutation operator of the absolute position genome.

Figure 5.8 shows the effect of a mutation for process 8 from 1 step to 4 steps. The relocation of process 8 causes all remain processes (9-12) to be relocated resulting in six phenes being changed. This example illustrates how a single mutation in the genome can have a significant effect on the phenome giving poor correlation.

5.5 Exploring Evolutionary Algorithm Parameters

This set of experiments are designed to explore the parameters that control the evolutionary algorithm to establish a suitable set of parameters for later experiments.

It is important to select a combination of genomic representation and associated parameters that maximise the efficiency of finding high quality solutions. The selection of genomic representation and the values of parameters used by the evolutionary algorithm can significantly affect the EA's ability of finding good solutions within an acceptable time frame. Initial experiments are designed to guide selection of the genomic representation and establish appropriate parameters for the evolutionary algorithm.

The number of combinations of evolutionary parameters is so large that it is impossible to exhaustively test all the combinations of parameters that are of interest. The order in which the parameters are explored is, to some extent, arbitrary. The approach taken is to explore combinations of a small number of related parameters while keeping the other parameters fixed, use the results to select a set of values that appear to work well together, then fix these parameters and move on to another set of related parameters to explore.

The evolutionary algorithm will be used in single objective mode for both core fault tolerance and network power to explore combinations of population size, number of generations and genome representation to determine parameters that are appropriate for both objectives.

5.5.1 Core Fault Tolerance Objective

For each combination of graph size and array size there is a minimum discovered fitness for the core fault tolerance objective that the evolutionary runs can find sufficiently often to make this a good indicator of the performance of the evolutionary algorithm parameters. This allows the results of these experiments to be expressed in terms of the number of evolutions that found the minimum discovered fitness. The number of evolutions that find the minimum discovered fitness is represented by f_m.

The result tables for the core fault tolerance experiments will use the column headings which are explained in Table 5.5.

The application process graph used in this set of experiments is illustrated in Figure 5.9.

Table 5.5 – Explanation of Table Columns

Column	**Description**
Evols:	The number of evolutions.
Gens:	The number of generations.
Pop:	The population size.
EA:	The genomic representation rd=Random; ph=phenome, sa=string; absolute position; sr=string, relative position.
mr:	The mutation rate m=fixed rate; g=Gaussian distribution; with the following number r representing the maximum mutation rate.
a:	The size of the array in terms of the dimension of one side of the array.
n:	The number of nodes in the application process graph being mapped to the array.
f_m:	The number of evolutions that found the minimum possible fitness.
$\%f_m$:	The percentage of evolutions that found a solution with the minimum fitness.
$\overline{ce_{f_m}}$:	The average computation effort of those evolutions that found a mapping with the minimum fitness. This calculation ignores any evolutions that did not find a mapping with the minimum fitness. It does not say anything about the evolutions that failed to find a mapping with the minimum fitness. A lower figure is more desirable.
amd:	The absolute mean deviation of the computational effort.
sd:	The standard mean deviation of the computational effort.

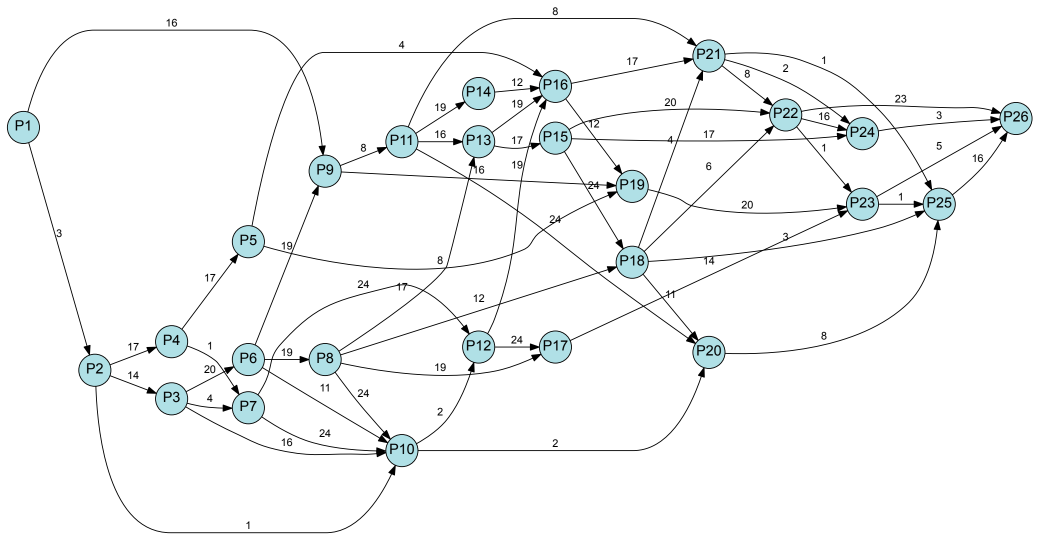


Figure 5.9 – Graph 'dag0026' - A 26 Node Application Process Graph

5.5.1.1 Effect of Population Size with Core Fault Tolerance Objective

The first exploration is to determine the effect of variations in the number generations and population size, while keeping the total number of evolutions constant. A mutation rate of m1 (a single mutation) is used in theses experiments.

Seven combinations of generations and population size are used, while keeping the num-

Table 5.6 – **EA Parameter Exploration Test Parameters**

Parameter	Value
Environment	6x6
Array Size	6x6
Graph size	26
Graph	dag0026
Evolutions	100
Individuals	100,000
Elite	The greater of 10% of the population or all individuals in pf1
Parents	20%
Descendants	4
Novel	10%
Mutation	m1

ber of evaluated individuals fixed at 100,000. Fixing the total number of evaluated individuals imposes a computational budget on the evolutionary process, which will be necessary in an embedded system - the target platform for this work. A fixed computational budget also ensures that the experiments are comparing like for like so that direct comparison of the results is valid. The number of evolutions is set to 100 for the collection of statistical data. The combinations of generations and population are tested against each of the three genomic representations using a fixed single mutation to produce new individuals from a parent.

A random generation of 100,000 individuals is included as a benchmark to ensure that the evolutionary algorithm performs better than random generation.

The evolutionary algorithm parameters used are given in Table 5.6 and the results are given in table Table 5.7.

The random generation of 100,000 individuals failed to find the minimum value for core fault tolerance, while all of the evolutionary algorithm experiments did find the minimum value. One evolutionary algorithm experiment (relative positioning phenome with population size 10) found the minimum value in 40% of evolutions, which appears to be an anomaly compared to all the other experiments that found the minimum value in at least 86% of evolutions. This confirms that the evolutionary algorithm is significantly better than random generation and that there is sufficient randomness in the evolutionary algorithm search of the solution space to find good solutions.

The statistical measures of *absolute mean deviation* (AMD) and *standard deviation* (SD) generally follow the average computational effort so, to compare experiments, only the percentage of experiments that found the minimum value and the average computation effort that it took to find the value need to be considered. The difference between AMD and SD is only in the magnitude of the values; they are identical for the purposes of comparing

Table 5.7 – Effect Population Size with Core Fault Tolerance Objective

Evols	Gens	Pop	EA	mr	a	n	f_m	$\%f_m$	$\overline{ce_{f_m}}$	amd	sd
100	1000	100	rd	m1	6	26	0	0	0	0	0
100	10000	10	ph	m1	6	26	93	93	3357	4251	9876
100	5000	20	ph	m1	6	26	100	100	6425	5813	9504
100	2000	50	ph	m1	6	26	99	99	8776	8182	12336
100	1000	100	ph	m1	6	26	96	96	11411	10133	15370
100	800	125	ph	m1	6	26	95	95	15746	16100	21613
100	500	200	ph	m1	6	26	86	86	15207	15256	19696
100	400	250	ph	m1	6	26	88	88	20474	19266	23977
100	10000	10	sa	m1	6	26	90	90	6640	7099	11282
100	5000	20	sa	m1	6	26	86	86	2679	2482	3811
100	2000	50	sa	m1	6	26	95	95	2395	1614	2432
100	1000	100	sa	m1	6	26	99	99	5628	5326	11033
100	800	125	sa	m1	6	26	99	99	5284	4303	9687
100	500	200	sa	m1	6	26	98	98	6259	4507	9261
100	400	250	sa	m1	6	26	99	99	4619	1648	2758
100	10000	10	sr	m1	6	26	40	40	23866	21564	26501
100	5000	20	sr	m1	6	26	95	95	19841	18341	23530
100	2000	50	sr	m1	6	26	100	100	11714	9312	14218
100	1000	100	sr	m1	6	26	97	97	10676	8104	12511
100	800	125	sr	m1	6	26	100	100	10465	7225	12249
100	500	200	sr	m1	6	26	99	99	10103	5822	9561
100	400	250	sr	m1	6	26	100	100	10520	6073	8935

Objective: Core Fault Tolerance.
Number of evaluated individuals fixed at 100,000.
Population sizes 10 - 250.

experiments which confirms that AMD can be used in preference to SD without loss of information and since the computation of AMD is significantly simpler than for SD, AMD is the preferred statistical method for this work.

The results are grouped by genomic representation each of which have a different results profile.

Analysis

The value of $\%f_m$ obtained by the *ph* genomic representation peaks for a population size of 20 with a value of 100% although the difference for populations between 20 and 125 is not significant, indicating that any of the population sizes in this range are good choices. This pattern can be explained by larger populations allowing for more genetic variation in each generation, while the decrease in the number of generations brings the evolutionary process to a premature halt. These two processes, acting in opposite directions result in poor performance for small and large populations while producing a higher performance region for population sizes between the two extremes. The average computational effort generally increases as the population size increases, except for the small dip for the population of 200, suggesting that, as the population size increase, the number of generations it takes to find the minimum solution increases.

The value of $\%f_m$ obtained by the *sa* genomic representation is almost 100% for populations between 100 and 250. Changes to the *sa* genome will often cause multiple changes to the phenome, which will cause greater variation of descendants. The results suggest that the larger phenomic variation has a more significant effect with populations larger than 50 resulting in more evolutions finding the best fitness. The average computational effort is also lower than for the *ph* genomic representation, which indicates that the minimum solutions are being found faster due to the larger changes to the phenome.

The value of $\%f_m$ obtained by the *sr* genomic representation has consistently high values of $\%f_m$ for all population sizes except the population size of 10 and are similar to the values for *sr*. Since the phenomic variation resulting from changes to the *sr* genome is also greater than the *ph* genome, these experiments confirm that the greater phenomic variation is responsible for the improved results.

The two string based genomic representations seem to work equally well with the middle and large population sizes. The phenomic representation works best will the middle populations, suggesting that the middle populations will perform more consistently in a wider range of situations.

Summary
For the mutation rate of m1 which give a single point mutation per generation, the *sa* and *sr* genomes give high performance across the whole range of population size while the *ph* representation performs best in the middle population sizes. The lower correlation of the *sa* and *sr* genomes with the phenome results in larger changes to the phenome for the m1 mutation rate compared to the highly correlated *ph* genome.

5.5.1.2 Effect of Mutation Rate with Core Fault Tolerance Objective

The mutation rate controls the number of changes made to the genome to create a new individual. If the changes are too small then the solutions may become confined to a localised area, while changes that are too large will be little better than randomly generated individuals.

This group of experiments will compare a single point mutation with Gaussian distributions with a range of upper limits.

For these experiments a population size of 100 has been chosen based on the results of the experiments in the previous section. The number of evolutions has been increased to 200 to provide a larger number of data points for the collection of statistics. The evolutionary algorithm parameters used for these experiments are in Table 5.8 and the results are in Table 5.9

Table 5.8 – EA Parameter Exploration Test Parameters

Parameter	Value
Environment	6x6
Array Size	6x6
Graph size	26
Graph	dag0026
Evolutions	200
Generations	1000
Population	100
Elite	The greater of 10% of the population or all individuals in pf1
Parents	20%
Descendants	4
Novel	10%

Analysis

The value of $\%f_m$ obtained by the *ph* genomic representation is 100% for mutation rates of g10 and above and only sightly below this level for m1 and g5. The average computational effort decreases with the increase in mutation rate. Taking these two measures together, implies that a higher mutation rate is beneficial for performance in terms of the ability to find an optimal solution and the number of generations required to find it.

The results for the *sa* and *sr* genomic representations both display better results for the lower mutation rates than the higher mutation rates. In both cases the results for the mutation rate of m1 are better that the results for all the other mutation rates.

Since the phenomic variation due to genomic changes is lower for *ph* than for *sa* which is lower than that for *sr*, the *ph* genome requires a higher mutation rate than the other representations to achieve the same phenomic variations. When the number of mutations becomes too large, the resultant changes to the phenome become random in nature and performance drops. This effect can be seen in the results of these experiments for the *sa* than for *sr* genomes.

As the performance of *sa* and *sr* reduces, the average computational effort increases which suggests that the larger changes to the phenomes is making solutions with good core fault tolerance harder to find.

The higher mutation rate for *ph* improved for performance, while for *sa* and *sr* performance was adversely affected. The choice of mutation rate is, therefore, dependent upon which genomic representation is used; lower values are better with string representations and higher values work better with the phenomic representation.

Summary

These results show that the *ph* genome outperforms the *sa* and *sr* genomes in terms of how many evolutions found an optimal mapping and how quickly the mapping was found.

Table 5.9 – Effect of Mutation Rate with Core Fault Tolerance Objective

Evols	Gens	Pop	EA	mr	a	n	f_m	$\%f_m$	$\overline{ce_{f_m}}$	amd	sd
200	1000	100	ph	m1	6	26	193	96.5	13242	13485	20463
200	1000	100	ph	g5	6	26	197	98.5	10574	9755	14020
200	1000	100	ph	g10	6	26	200	100.0	8930	9187	14590
200	1000	100	ph	g12	6	26	200	100.0	7857	7594	13158
200	1000	100	ph	g15	6	26	200	100.0	6303	5931	9366
200	1000	100	ph	g20	6	26	200	100.0	4661	3472	5296
200	1000	100	sa	m1	6	26	197	98.5	5286	4459	8576
200	1000	100	sa	g5	6	26	191	95.5	6927	6174	11703
200	1000	100	sa	g10	6	26	193	96.5	11957	9576	14264
200	1000	100	sa	g12	6	26	184	92.0	12145	10004	15310
200	1000	100	sa	g15	6	26	188	94.0	14319	10818	15162
200	1000	100	sa	g20	6	26	184	92.0	16516	13031	18449
200	1000	100	sr	m1	6	26	199	99.5	10471	7190	11548
200	1000	100	sr	g5	6	26	199	99.5	14762	9604	15105
200	1000	100	sr	g10	6	26	196	98.0	22859	13307	18313
200	1000	100	sr	g12	6	26	190	95.0	26508	16638	21046
200	1000	100	sr	g15	6	26	193	96.5	31611	18887	23595
200	1000	100	sr	g20	6	26	167	83.5	37620	19450	23438

Objective: Core Fault Tolerance.
Number of evaluated individuals fixed at 100,000.
Generations: 1000, Population: 100.

This is due to the correlation of the genome and phenome making the same number of changes to the phenome as a made to the genome.

The *ph* genome looks like a good choice for latter experiments with a mutation rate of at least g10.

5.5.1.3 Effect of Population Size with Core Fault Tolerance Objective Revisited

We now return to the comparison of population size, but this time using a mutation rate of g15. The *sr* genome is omitted because, while the experiments results were reasonable, they are judged to be more erratic than the other two representations and the average computation effort required is, overall, greater than for the other two genomic representations.

For these experiments a mutation rate of g15 has been chosen based on the results of the experiments in the previous section. The evolutionary algorithm parameters used for these experiments are in Table 5.10 and the results are in Table 5.11

The number of evolutions has been increased to 200 to increase the accuracy of the results of the experiments.

Analysis

Table 5.10 – **EA Parameter Exploration Test Parameters**

Parameter	Value
Environment	6x6
Array Size	6x6
Graph size	26
Graph	dag0026
Evolutions	200
Individuals	100,000
Elite	The greater of 10% of the population or all individuals in pf1
Parents	20%
Descendants	4
Novel	10%
Mutation	m1

The value of $\%f_m$ obtained by the *ph* genomic representation is 100% for all populations except for the population of 10 which is the smallest population. Compared to the results using the m1 mutation rate the results obtain with the g15 mutation rate are significantly better except for the population of 10. As observed in the Subsubsection 5.5.1.1 experiments, the average computational effort increases with population size.

The results of these experiments shows a marked difference between the *ph* and *sa* with the phenomic genome outperforming the string genome in both $\%f_m$ and average computational effort.

Summary
The conclusion from this set of experiments is that the combination of phenomic genome, population of 100 and mutation rate of g15, gives a high level of performance and is a good compromise compared to other high performing combinations.

5.5.2 Network Power Objective

In contrast to the core fault tolerance objective value, whose minimum value can be easily established, the network power objective value is much more varied with a lower bound that cannot be established with a high level of certainty. The number evolutions that find the same minimum value of the network power objective is so low that statistics based on the number of evolutions finding the minimum value are not useful. Instead these experiments will use statistics based on the actual minimum fitness found and the computational effort required to find the value.

Because there is not an established minimum value to find, each evolution will evaluate all individuals in all generations specified by the evolutionary algorithm parameters. The evolutionary algorithm has been designed to keep track of the generation in which the

Table 5.11 – **Effect of Population Size with Core Objective**

Evols	Gens	Pop	EA	mr	a	n	f_m	$\%f_m$	$\overline{ce_{f_m}}$	amd	sd
200	10000	10	ph	g15	6	26	175	87.5	8128	8654	13665
200	5000	20	ph	g15	6	26	200	100.0	3062	2598	4258
200	2000	50	ph	g15	6	26	200	100.0	4845	4216	7214
200	1000	100	ph	g15	6	26	200	100.0	5568	4825	7477
200	800	125	ph	g15	6	26	200	100.0	7555	6944	12435
200	500	200	ph	g15	6	26	200	100.0	8633	8104	12460
200	400	250	ph	g15	6	26	200	100.0	7368	5827	9276
200	10000	10	sa	g15	6	26	154	77.0	18632	16571	21526
200	5000	20	sa	g15	6	26	141	70.5	13470	12962	18868
200	2000	50	sa	g15	6	26	171	85.5	10812	8891	14442
200	1000	100	sa	g15	6	26	181	90.5	11970	9370	13498
200	800	125	sa	g15	6	26	186	93.0	13515	10643	16075
200	500	200	sa	g15	6	26	193	96.5	17196	11211	16314
200	400	250	sa	g15	6	26	190	95.0	15559	9760	14757

Objective: Core Fault Tolerance.
Number of evaluated individuals fixed at 100,000.
Population sizes 10 - 250.

minimum value of the whole evolution was found, which is then used as the computational effort for finding the value.

The result tables for the network power experiments will use the column headings which are explained in Table 5.12.

Table 5.12 – **Explanation of Table Columns**

Column	**Description**
Evols:	The number of evolutions.
Gens:	The number of generations.
Pop:	The population size.
EA:	The genomic representation rd=Random; ph=phenome, sa=string; absolute position; sr=string, relative position.
mr:	The mutation rate m=fixed rate; g=Gaussian distribution; with the following number representing the maximum mutation rate.
a:	The size of the array in terms of the dimension of one side of the array.
n:	The number of nodes in the application process graph being mapped to the array.
$\overline{f}$:	The average of the minimum fitness found by each evolution.
ce:	The computation effort required to find the minimum fitness.
$\overline{ce_{f_m}}$:	The average computation effort of each evolution.
amd:	The absolute mean deviation of the computational effort.
sd:	The standard mean deviation of the computational effort.

5.5.2.1 Effect of Population Size with Power Metric

Guided by the results of experiments in Subsection 5.5.1, the mutation rate is fixed at g15, and the genomic representations are restricted to *ph* and *sa* while the population size is varied from 10 to 250. The evolutionary algorithm parameters used for these experiments are in Table 5.13 and the results are in Table 5.14.

Table 5.13 – EA Parameter Exploration Test Parameters

Parameter	Value
Environment	6x6
Array Size	6x6
Graph size	26
Graph	dag0026
Evolutions	100
Individuals	100,000
Elite	The greater of 10% of the population or all individuals in pf1
Parents	20%
Descendants	4
Novel	10%
Mutation	g15

Analysis

For the *ph* genome there is no clear correlation between the population size and the average minimum fitness found. The worst results were obtained from the smallest population size of 10 and the best results from the population of size 20 with the population size of 100 having the median value. The average computational effort increases and the statistical deviation reduces as the population size increases.

The population sizes of 200 and 250 have the 2nd and 3rd highest average minimum fitness and highest computational effort. The poor average minimum fitness and high computational effort implies that the lowest fitness was found by the evolution towards the end of the evolution and so a greater number of generations may improve performance.

For the population size of 10, which has the worst average minimum fitness and the lowest computational effort. This suggest that the best fitness solution was found early in the run and the small population size prevented the evolutionary algorithm finding better mappings.

For the *sa* genome the average minimum fitness found tended to decrease, the computational effort increased, and the statistical deviation decreased with increasing population size.

For population size 10 - 125 the *ph* genome outperformed the *sa* genome, while for population sizes 200 and 200 the *sa* genome outperformed the *ph* genome.

These results have not provided a clear picture, so an additional set of experiments with a

Table 5.14 – Effect of Population Size with Power Objective 100K Individuals

Evols	Gens	Pop	EA	mr	a	n	$\overline{f}$	$\overline{ce}$	amd	sd
100	1000	100	rd	m1	6	26	96.72	31099	31155	34206
100	10000	10	ph	g15	6	26	48.43	37689	21311	25464
100	5000	20	ph	g15	6	26	43.84	59495	20039	23527
100	2000	50	ph	g15	6	26	43.90	66658	17292	20508
100	1000	100	ph	g15	6	26	44.50	76144	13758	16632
100	800	125	ph	g15	6	26	43.92	77588	13771	16297
100	500	200	ph	g15	6	26	45.12	83240	10026	13020
100	400	250	ph	g15	6	26	45.06	85838	9098	11676
100	10000	10	sa	g15	6	26	50.99	48314	24730	28496
100	5000	20	sa	g15	6	26	47.05	52762	19135	23462
100	2000	50	sa	g15	6	26	45.88	60038	22754	26418
100	1000	100	sa	g15	6	26	44.93	65280	20120	23457
100	800	125	sa	g15	6	26	44.34	67118	19373	22798
100	500	200	sa	g15	6	26	44.80	69352	13872	17756
100	400	250	sa	g15	6	26	44.58	76145	14568	17096

Objective: Network.
Number of evaluated individuals fixed at 100,000.
Population sizes 10 - 250.

four fold increase in the computational budget to 400,000 evaluated individuals were run. The results obtained from these experiments are in presented in Table 5.15.

The extended experiments show an overall improvement in performance with the average minimum fitness reduced compared to the experiments with a smaller number of individuals. The improvement is small compared to the additional computation effort, which is four times larger.

For both the *ph* genome and *sa* genome, the best average minimum fitness were obtained with the largest population of 250 which also had the highest computational effort.

In terms of average minimum fitness, there is little difference in the results of *ph* genome and *sa* genome, however the combination of good average minimum fitness and low computational effort is best for the *ph* genome with population sizes 20,50 and 100.

Summary

Overall, the results suggest that finding an optimal solution for the network power objective is much more difficult than for core fault tolerance. This reflects the comments at the beginning of this section that the power objective values are much more varied and a minimum value more difficult to identify.

There is not one "best" combination of genome and population size for finding a low average minimum fitness and low computational effort, although the results indicate the the choice of population size of 100 with a 1000 generations for an evolution remain reasonable

Table 5.15 – Effect of Population Size with Power Objective 400K Individuals

Evols	Gens	Pop	EA	mr	a	n	$\overline{f}$	$\overline{ce}$	amd	sd
100	40000	10	ph	g15	6	26	47.55	131698	91788	107801
100	20000	20	ph	g15	6	26	41.72	144930	86280	105357
100	8000	50	ph	g15	6	26	42.22	148142	82009	101741
100	4000	100	ph	g15	6	26	41.24	197944	78968	90116
100	3200	125	ph	g15	6	26	41.33	207313	83235	95525
100	2000	200	ph	g15	6	26	41.84	234520	83188	96835
100	1600	250	ph	g15	6	26	41.02	255648	72538	83791
100	40000	10	sa	g15	6	26	47.33	178985	92941	110180
100	20000	20	sa	g15	6	26	44.23	202843	103548	120952
100	8000	50	sa	g15	6	26	42.75	219694	88250	104183
100	4000	100	sa	g15	6	26	41.79	204852	87388	101112
100	3200	125	sa	g15	6	26	41.93	211638	87319	101888
100	2000	200	sa	g15	6	26	41.39	210178	85526	98039
100	1600	250	sa	g15	6	26	40.97	241428	89078	101662

Objective: Network Power.
Number of evaluated individuals fixed at 400,000.
Population sizes 10 - 250.

choices.

5.5.2.2 Network Power and Mutation Rate

To explore the effect of mutation rate with the network power objective the population size is fixed at 100 and the number of generations fixed at 2000, resulting in the evaluation of 200,000 individuals. The evolutionary algorithm parameters used for these experiments are in Table 5.16 and the results are in Table 5.17.

Analysis
For both the *ph* genome and *sa* genome, the average minimum fitness and computational effort increase with the mutation rate. The difference between the best performance for the mutation rate m1, and the worst performance with mutation rate g20 is less than 10%. The average minimum fitness is very similar for both genomes, while the computational effort is generally lower for the *ph* genome.

Taking these results alongside those of Table 5.15, where better solutions are found with larger population sizes, leads to the conclusion that, to find the best solution for the network power objective, the best strategy is to use a large population, with single point mutations and a large number of generations. This, however, is not generally an efficient strategy as it involves the evaluation of a large number of similar individuals, and so is inappropriate for more general multi-objective problems.

Summary

Table 5.16 – **EA Parameter Exploration Test Parameters**

Parameter	Value
Environment	6x6
Array Size	6x6
Graph size	26
Graph	dag0026
Evolutions	100
Generations	2000
Population	100
Elite	The greater of 10% of the population or all individuals in pf1
Parents	20%
Descendants	4
Novel	10%

Finding minimum fitness solutions for the network power objective has proved to be significantly more difficult than find good fault tolerance mappings. The results imply that, by a relatively small margin, the best strategy is to a mutation rate of m1, with a large population size and since the benefit is margin for the power objective and is not compatible with the results for the fault tolerance objective, the preferred choice of population size remains at 100 and the mutation rate remains at g15.

5.6 Core FT with Network Power Results

These experiments explore the effect of varying the graph size, for a given array size, for a multi-objective problem.

The array size chosen is a 6×6 with application process graph sizes ranging from 24 to 34 processing nodes. The full set of evolutionary parameters can be found in Table 5.18.

The objectives chosen for these experiments are core fault tolerance and network power.

5.6.1 Data Presentation

In each experiment 100,000 individuals are evaluated and plotted using graded colours representing the generation when the individual first appeared. The colour bar relating the generation of creation to a colour is shown in Figure 5.10. The points of the last generation are plotted first, and the first generation plotted last so that points in the early generations are not masked by the points of later generations.

Plots of Individuals

The purpose of these experiments is to compare how the application process graph size affects the fitness values of individuals. To facilitate a like for like comparison between plots

Table 5.17 – Network Power Objective - Mutation Rate

Evols	Gens	Pop	EA	mr	a	n	$\overline{f}$	$\overline{ce}$	amd	sd
100	2000	100	ph	m1	6	26	41.07	69363	41532	49607
100	2000	100	ph	g5	6	26	41.38	81354	40581	48896
100	2000	100	ph	g10	6	26	41.99	115823	43168	48898
100	2000	100	ph	g12	6	26	42.00	122319	38853	45760
100	2000	100	ph	g15	6	26	42.37	118742	34812	41633
100	2000	100	ph	g20	6	26	43.07	141894	32964	38786
100	2000	100	sa	m1	6	26	40.99	86817	44553	51588
100	2000	100	sa	g5	6	26	41.78	99179	44531	53342
100	2000	100	sa	g10	6	26	42.72	111966	45836	51826
100	2000	100	sa	g12	6	26	42.80	112224	48522	54660
100	2000	100	sa	g15	6	26	43.20	116688	45066	51731
100	2000	100	sa	g20	6	26	44.65	122021	40422	48344

Number of evaluated individuals fixed at 200,000.
Generations: 2000, Population size: 100.

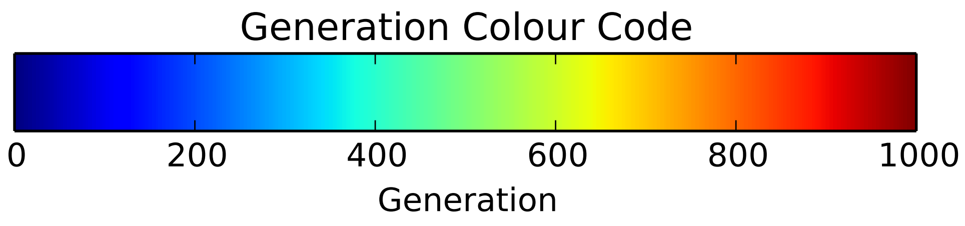


Figure 5.10 – Generations Colour Bar

the scale of the axes is identical for all plots. The scale and range have been chosen so that all individuals for all graph sizes can be shown.

Two plots are shown for each graph size: figure (a) showing the whole dataset, with figure (b) magnifying the bottom left hand corner of figure (a) to emphasise the points on the Pareto front.

5.6.2 Results Analysis

Core Fault Tolerance Objective
We know from the single objective experiments in Chapter 4 that, for core fault tolerance the minimum objective value is 0 for graph sizes up to 26, and then the minimum fitness increases as the graph size continues to increase. Chapter 4 also gave an indication that the range of fitness values increased as the graph size increased

Both of these patterns are also evident with multi-objective plots, where we see the points moving away from 'Core FT = 0' axis and expanding at the same time. This confirms that the ability to find mappings with optimum core fault tolerance fitness has not been

Table 5.18 – **Power with Core Fault Test Parameters**

Parameter	Value
Hardware Map Size	6 x 6
Many-Core Array Size	6 x 6
Many-Core Array Offset	0, 0
Number of Generations	1000
Population Size	100
Elite	10 fittest individuals cloned
Parents	20 fittest individuals used as parents
Descendants	4 from each parent via permutation
Novel	10 randomly generated individuals
Mutation Pattern	g15

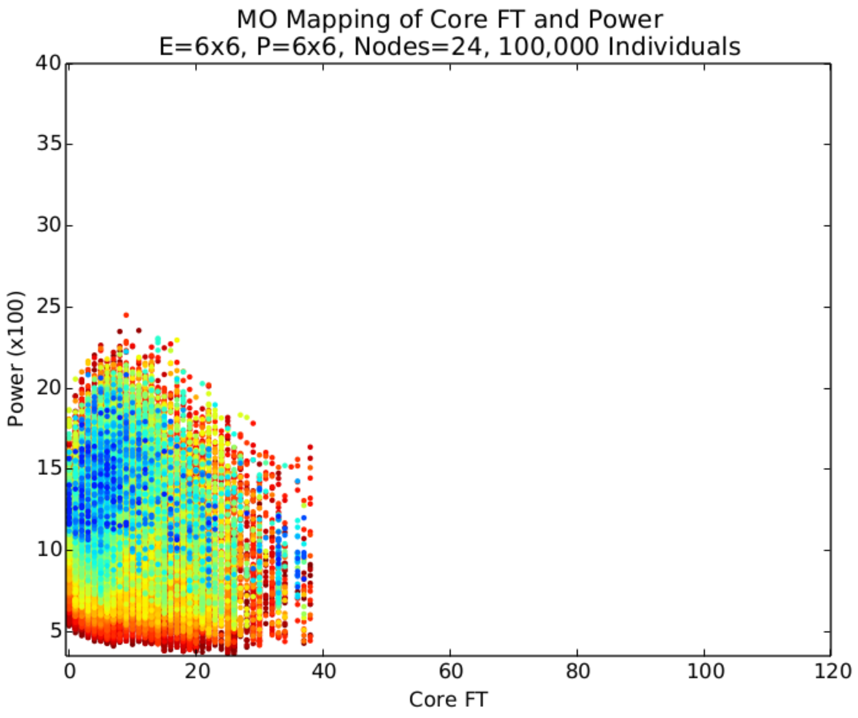
compromised by the move from single to multi-objective giving confidence that the multi-objective evolutionary algorithm is performing well.

An additional feature that was not evident in the single objective experiments is that the number of unique fitness values for core fault tolerance for the large graph of 34 nodes in Figure 5.21a suddenly reduces compared to the graph with 33 nodes. The explanation is that for a graph size of 34 there are only two idle cores. The number of unique placements of two cores in a 6×6 array is considerably smaller than for three idle cores.

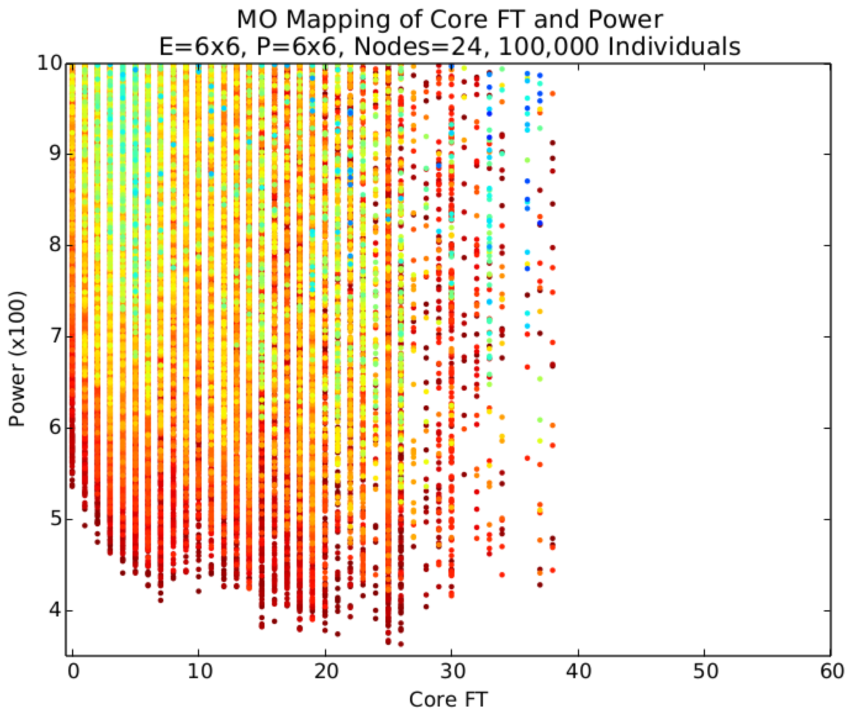
Network Power Objective

The minimum values of the network power objective are lowest for the smallest graph sizes. This is expected as the smaller graph sizes allows more options for placing ComPairs close together, which reduces the network power objective values, while being able to maintain good core fault tolerance properties.

The minimum values of the network power objective increase from graph size 24 to 25 and 26, followed by lower minimum values for graph sizes 27 and 28 before increasing again. From Figure 4.14b in Chapter 4 we know that the solutions for some combinations of array size and graph size are found quicker than for other combinations. In Figure 4.14b, solutions the graph with 27 nodes were found particularly quickly, which is an indication that this combination results in a large number of mappings with the optimal arrangement. Although similar single object experiments were not carried out for the network power objective, we can speculate that the same may be true for objectives other than core fault tolerance and that this is the effect we are observing for the power objective for graphs of size 27 and 28.

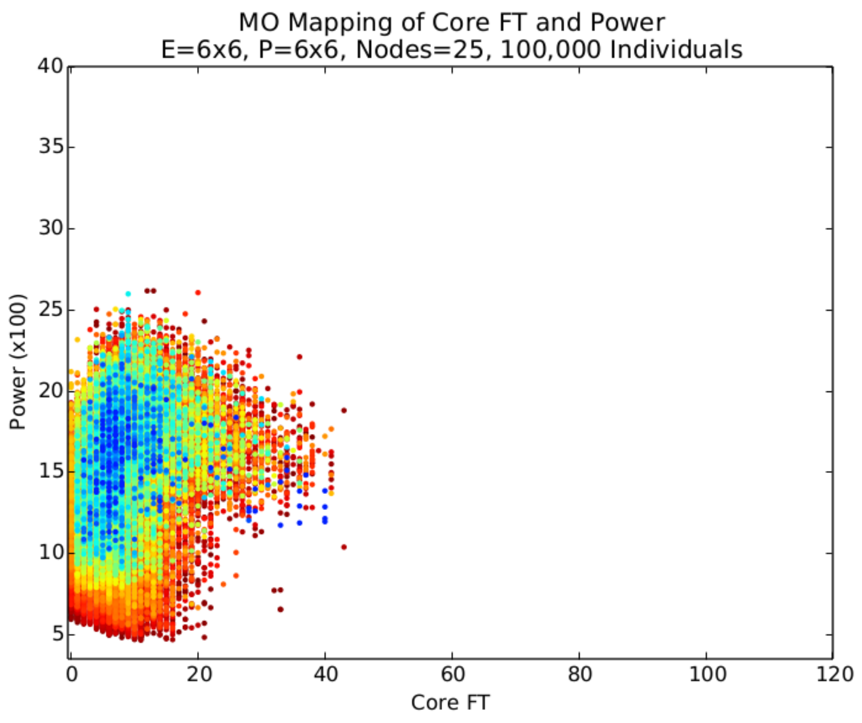


(a) All individuals

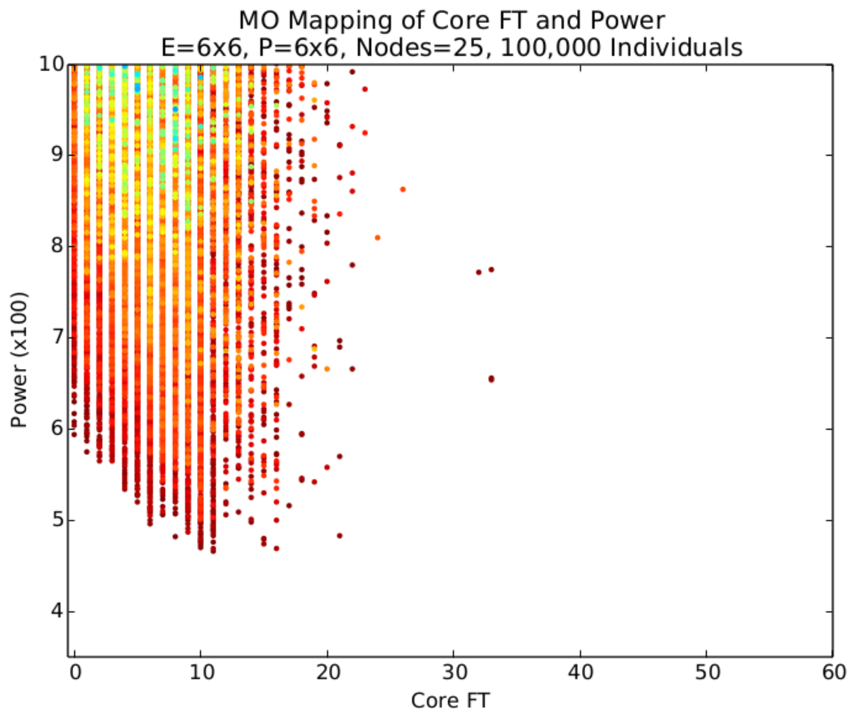


(b) Pareto front Individuals

Figure 5.11 – Core FT with Performance, 24 Node Graph

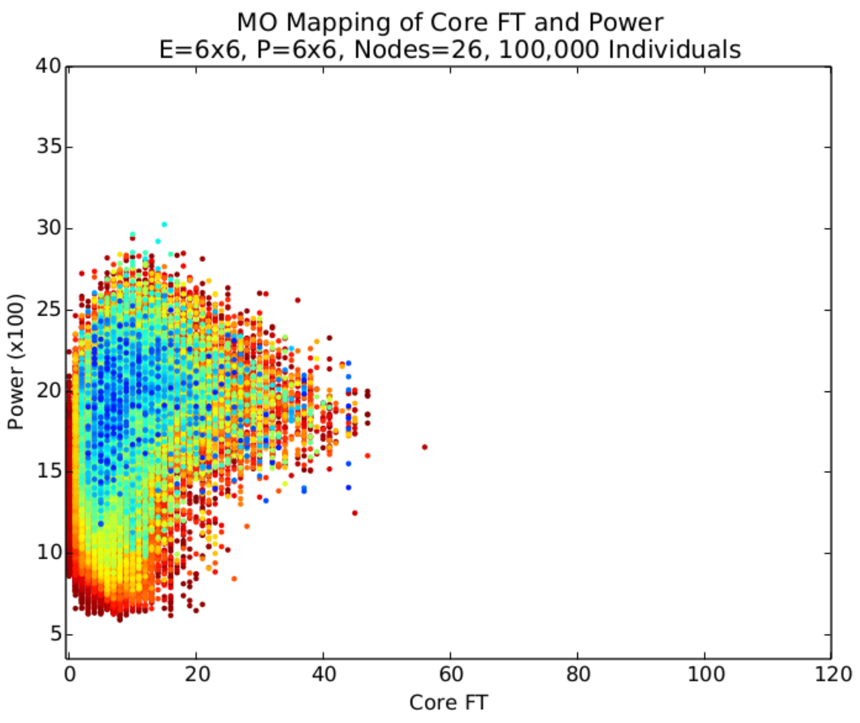


(a) All individuals

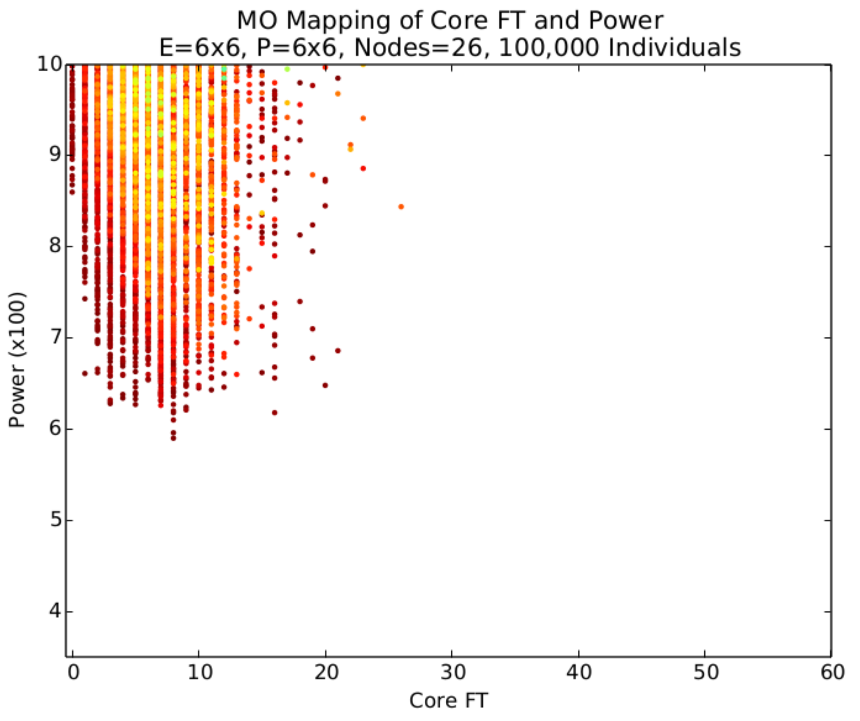


(b) Pareto front Individuals

Figure 5.12 – Core FT with Performance, 25 Node Graph

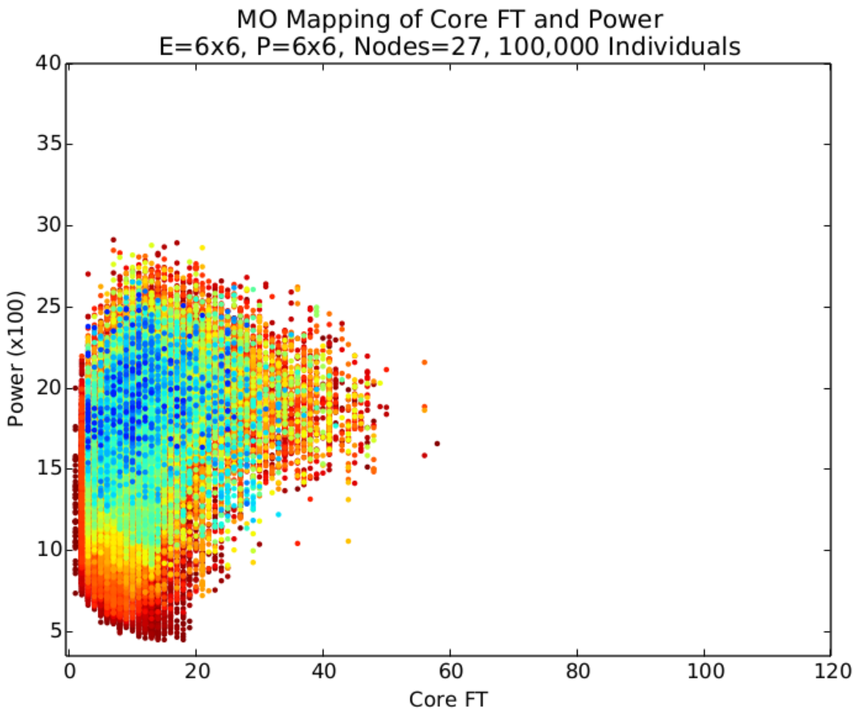


(a) All individuals

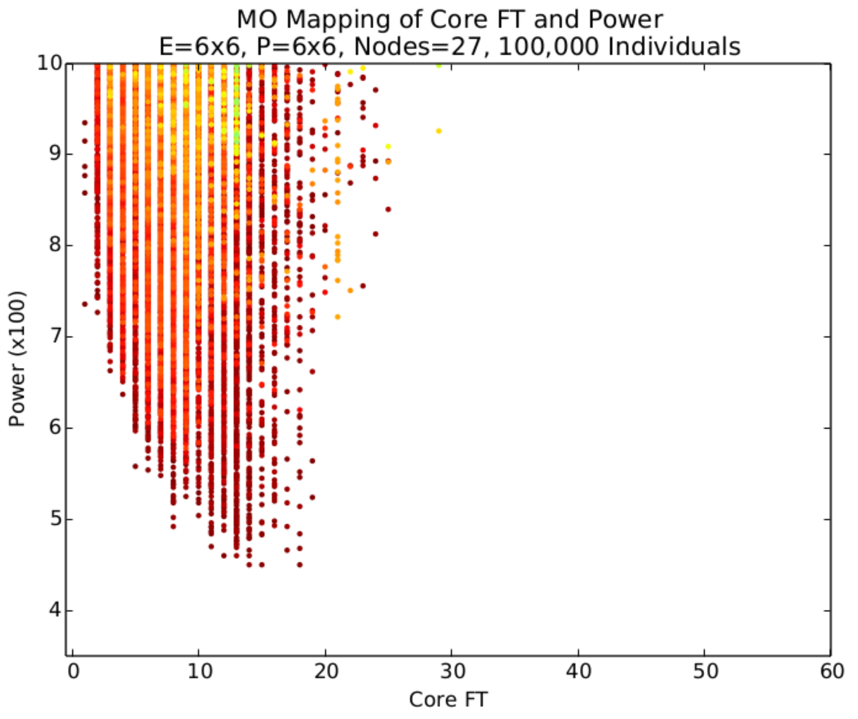


(b) Pareto front Individuals

Figure 5.13 – Core FT with Performance, 26 Node Graph

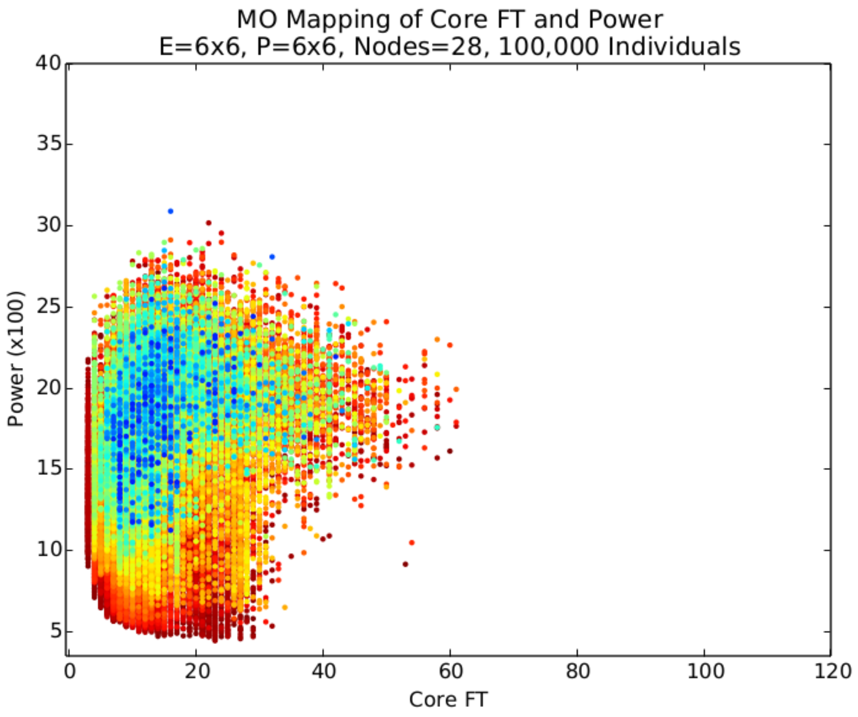


(a) All individuals

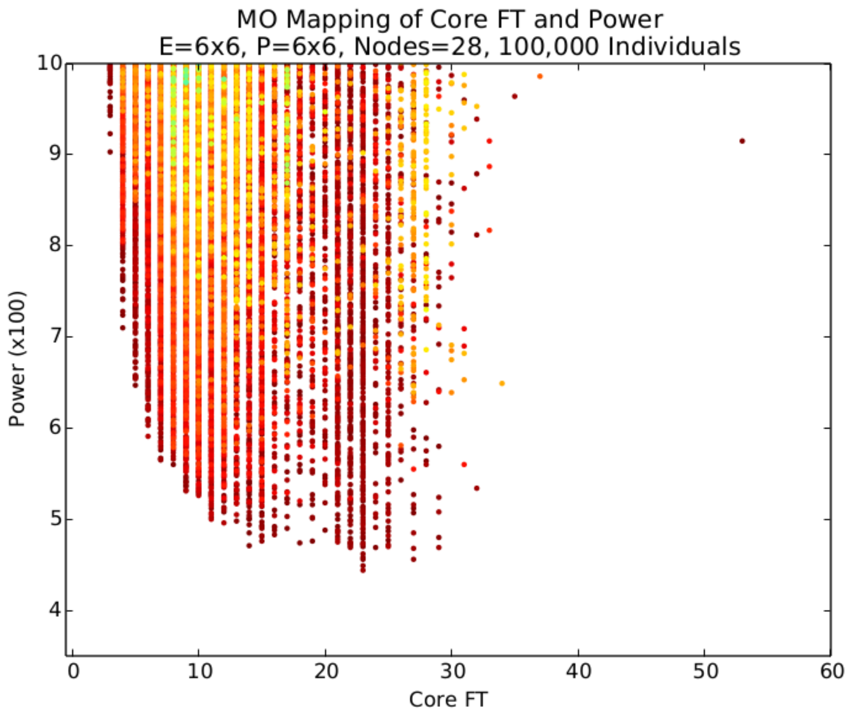


(b) Pareto front Individuals

Figure 5.14 – Core FT with Performance, 27 Node Graph

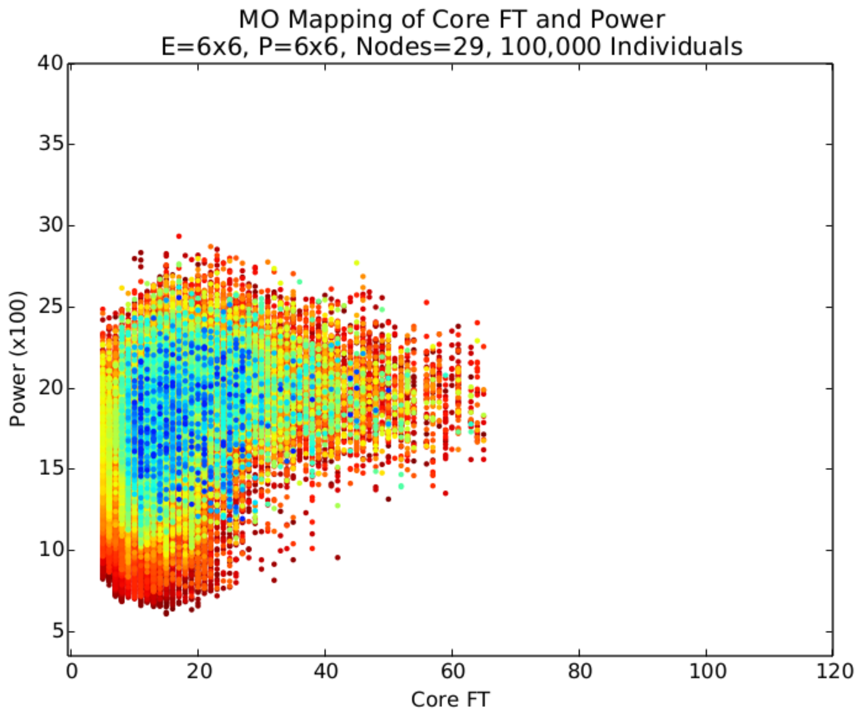


(a) All individuals

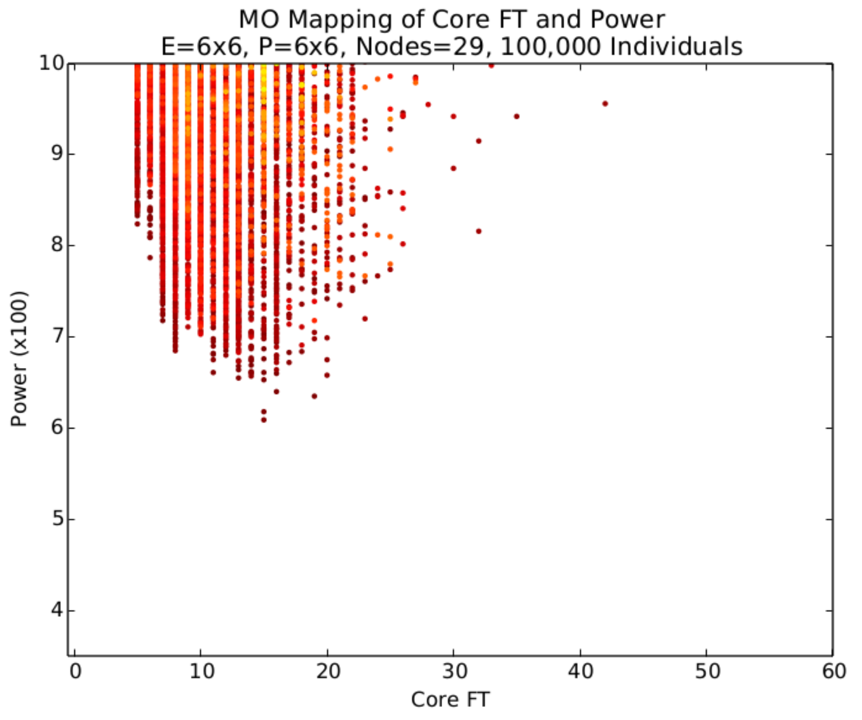


(b) Pareto front Individuals

Figure 5.15 – Core FT with Performance, 28 Node Graph

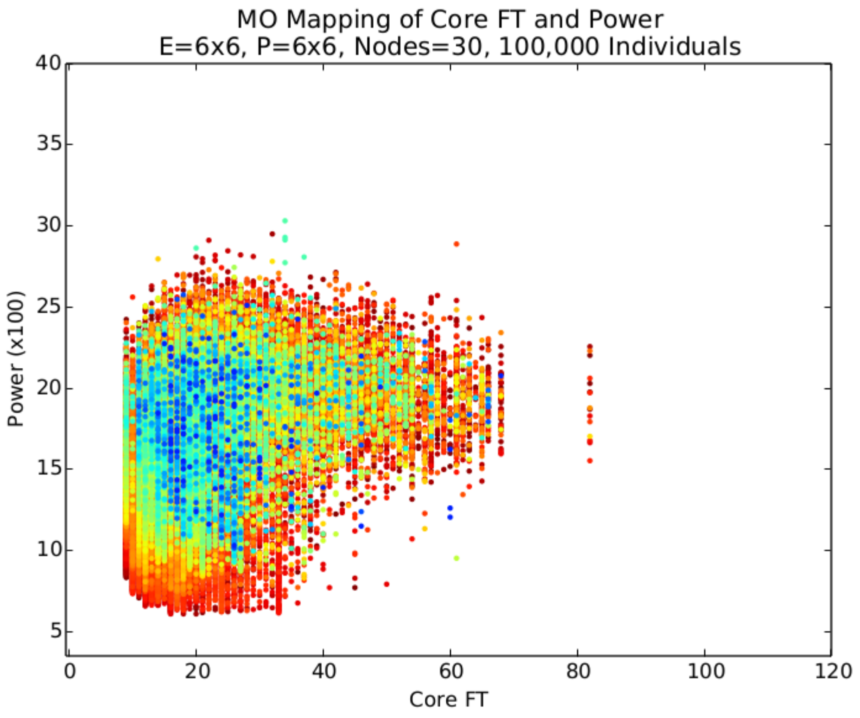


(a) All individuals

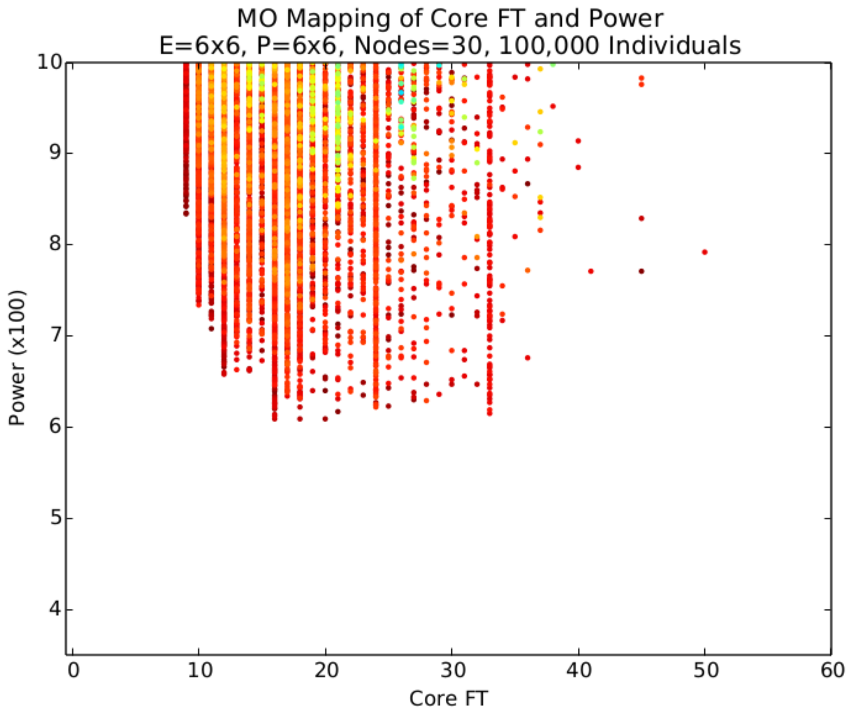


(b) Pareto front Individuals

Figure 5.16 – Core FT with Performance, 29 Node Graph

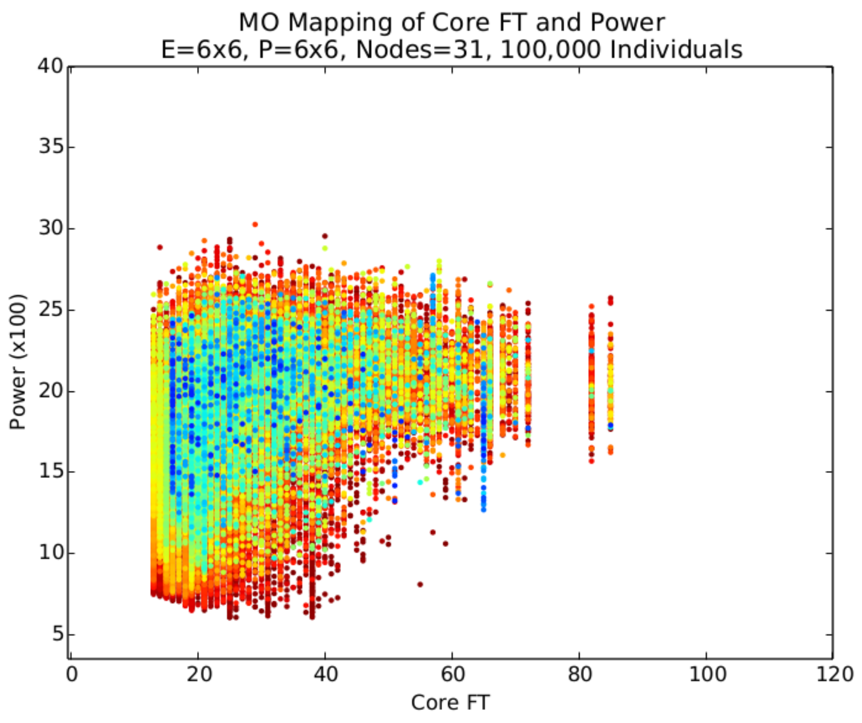


(a) All individuals

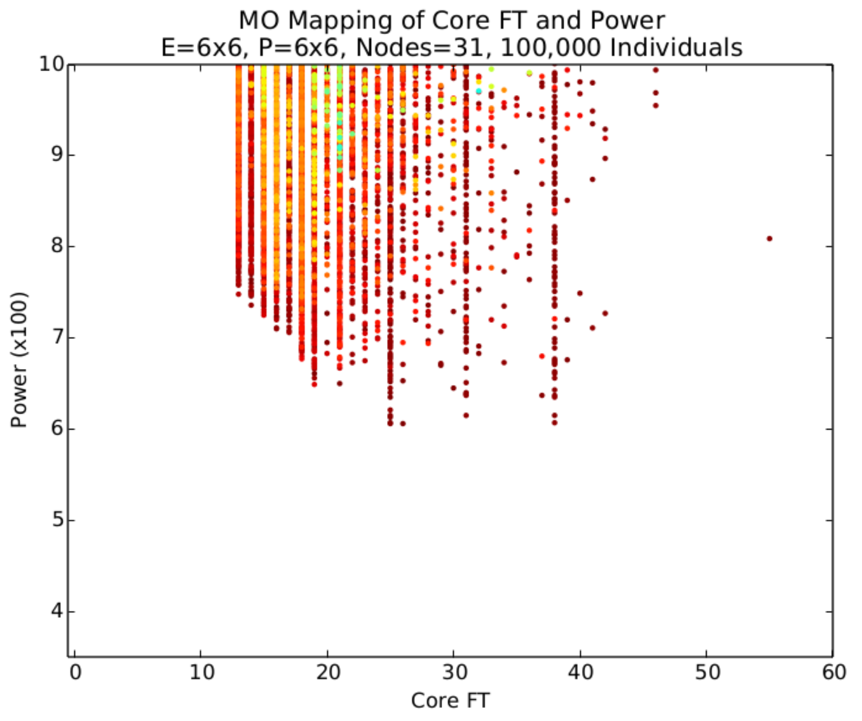


(b) Pareto front Individuals

Figure 5.17 – Core FT with Performance, 30 Node Graph

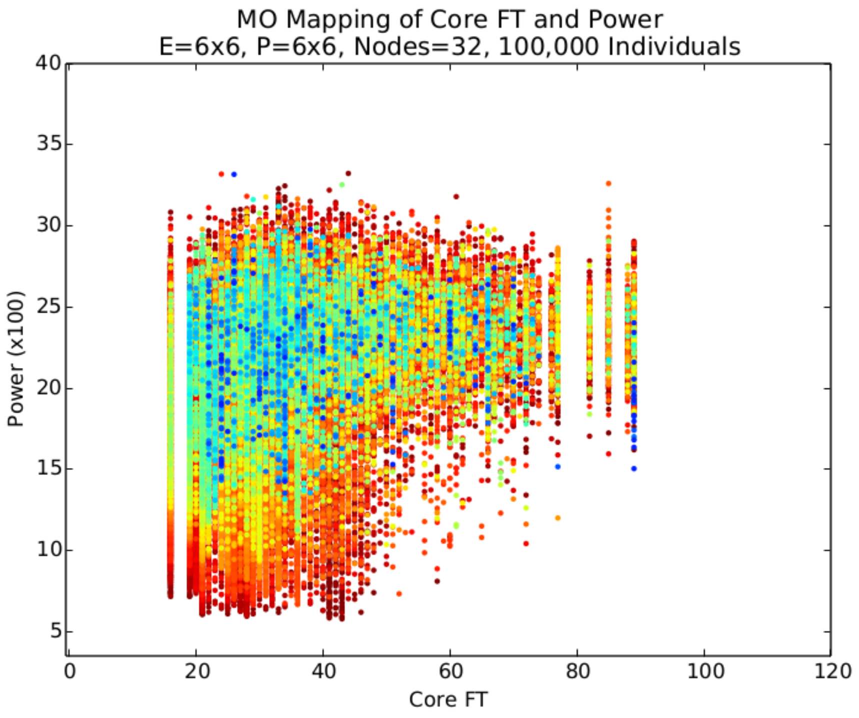


(a) All individuals

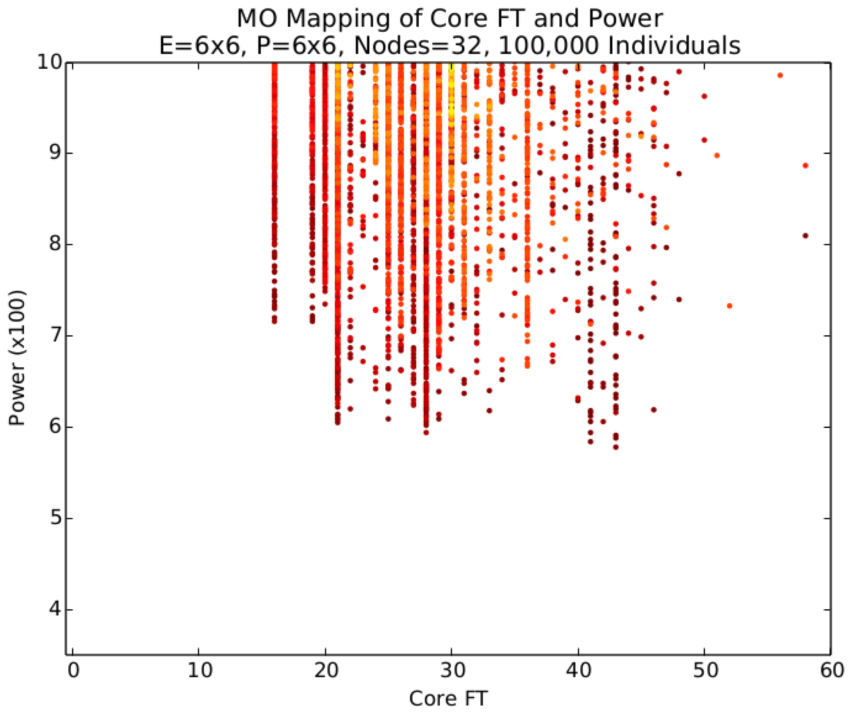


(b) Pareto front Individuals

Figure 5.18 – Core FT with Performance, 31 Node Graph

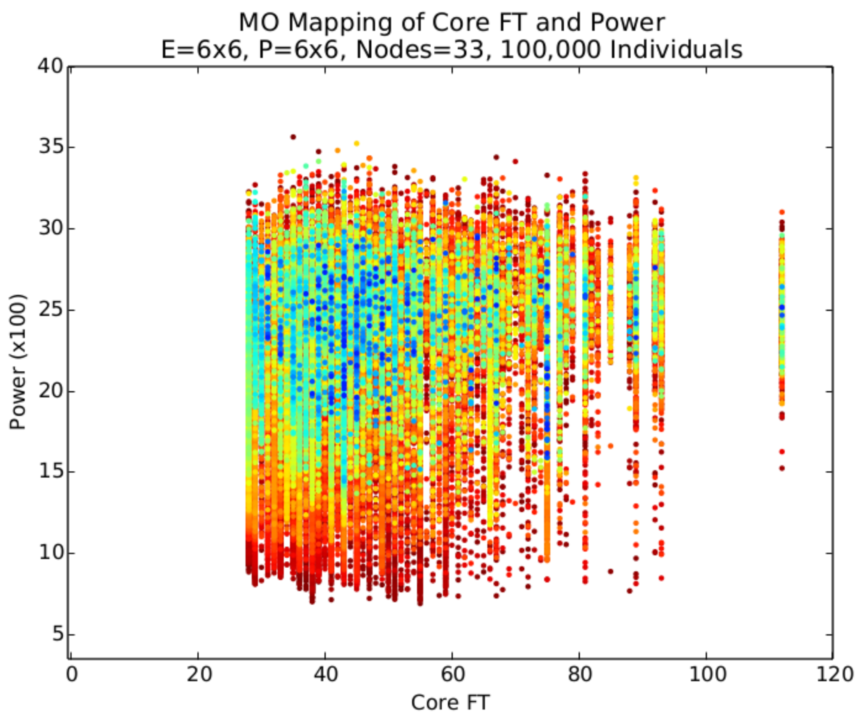


(a) All individuals

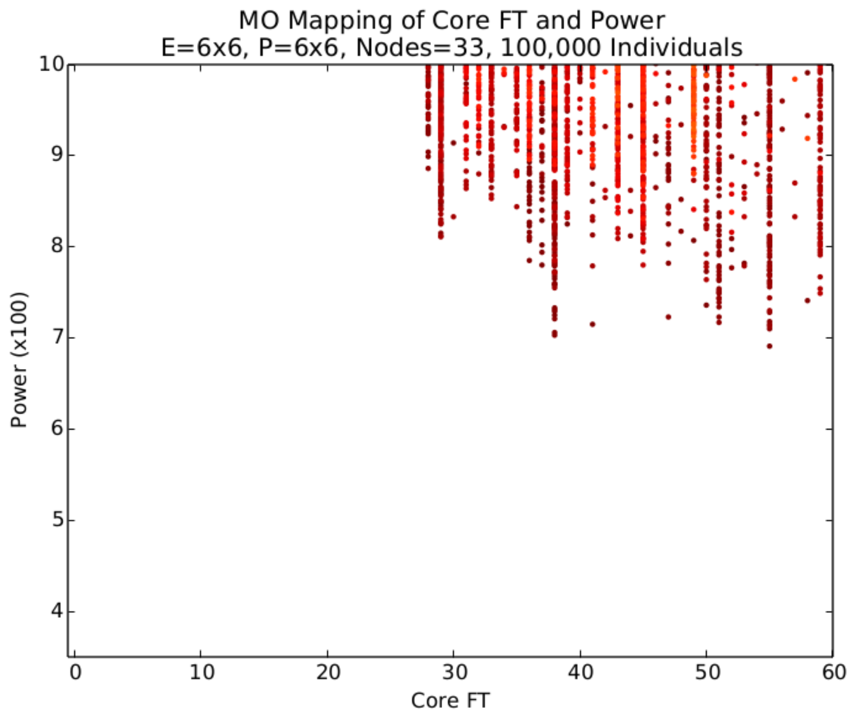


(b) Pareto front Individuals

Figure 5.19 – Core FT with Performance, 32 Node Graph

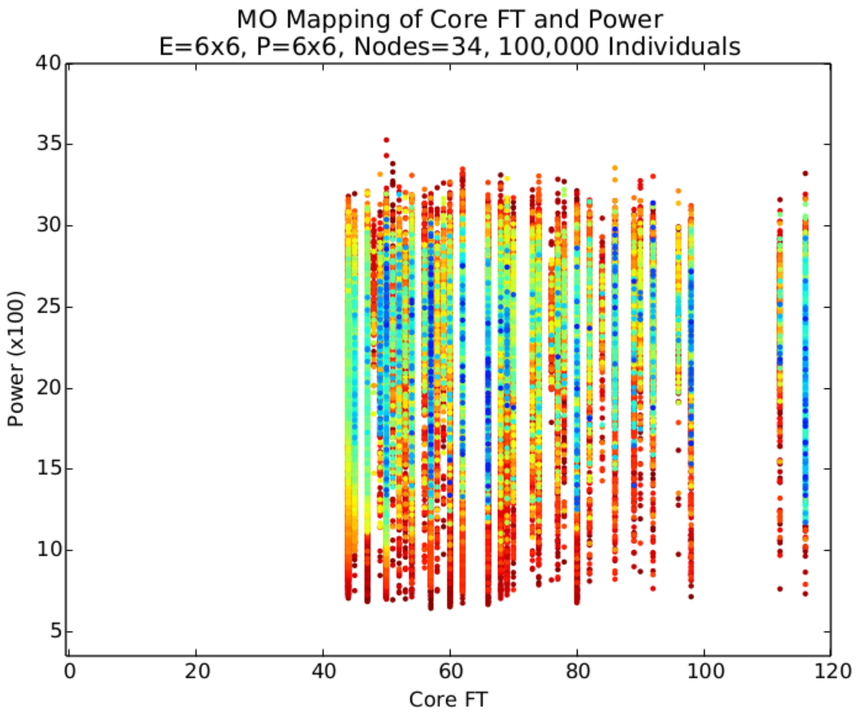


(a) All individuals

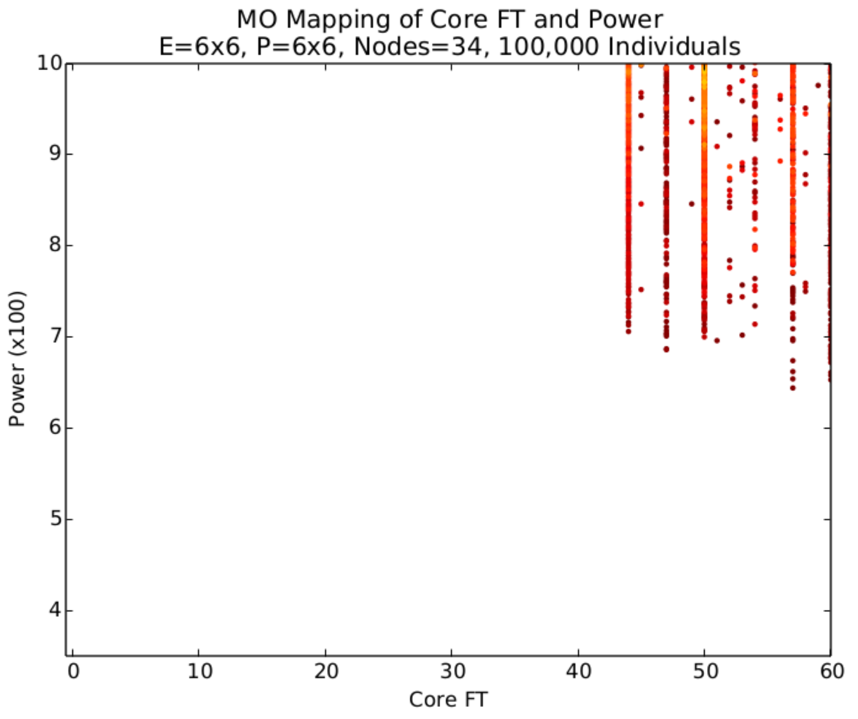


(b) Pareto front Individuals

Figure 5.20 – Core FT with Performance, 33 Node Graph



(a) All individuals



(b) Pareto front Individuals

Figure 5.21 – Core FT with Performance, 34 Node Graph

5.7 Conclusion

These results build on the results of the single objective experiments exploring the effect of the size of an application process graph for a fixed many-core array size.

The result of the single objective and multi-objective experiments support each other, giving confidence that the evolutionary algorithm is functioning well. Both sets of results also show that the objective values do not follow a smooth path of change as the graph sizes increase. For certain combinations of array size and graph size, it is easier to find optimal solutions indicating that there are more optimal solutions in the solution set. This effect was first observed in the single objective experiments for core fault tolerance, while the multi-objective experiments indicate this is also true for the network power objective, and by implication for other objectives. To confirm this, further work with single objective experiments are required.

The ability of the evolutionary algorithm to find good quality mappings has been shown to be superior to randomly generated mappings.

The evolutionary algorithm parameters have been explored in an attempt to find a set of parameters that perform well in a range of situations. Different sets of parameters performed better in different situations, so any single set of parameters is a compromise. The set of parameters chosen for future experiments are:

- Population size: 100
- Generations: 1000
- Total individuals 100,000
- Mutation rate Gaussian with maximum of 15 mutations
- Genome type: Phenomic

Chapter 6

Link Fault Tolerance and Network Traffic

This chapter extends the fault tolerance properties of mappings by adding fault tolerance of links to the core fault tolerance defined in Chapter 4, for which a new textitLink Fault Tolerance objective is added. The hardware links also feature in a new *Network Traffic* objective, design to spread traffic evenly across the network.

A series of multi-objective experiments are run, using with pairs of objectives, to explore the efficacy of the new objectives and also to establish the orthogonality between each pair of objectives. The results of the experiments are used to select suitable objectives for the final chapter.

The *Link Fault Tolerance* objective is designed to direct the search algorithm to find mappings that are tolerant to a faulty link and the *Network Traffic* objective, for which a variety of metrics are proposed, is designed to direct the search algorithm to find mappings where the traffic is evenly distributed across the many-core array. The simple power objective of Chapter 5 is replaced with a more sophisticated and accurate *Network Power* objective.

To support the calculation of the new and revised metrics and to more accurately model a many-core system, the model is also revised and extended. The application process graph of Chapter 4 is revised and a *hardware map* and an *environment map* are added to model multiple sources of data received by the application and multiple sinks for data produced by the application, where the sources and sinks equate to interfaces between the many-core array and external data sources, data sinks, sensors, actuators and neighbouring many-core regions. The hardware map also maintains a real-time inventory of faulty cores and faulty links.

6.1 Application Process Graph (APG)

This section extends the application process graph model given in Section 4.3 by adding source and sink nodes which model interfaces to resources external to the circuitry of the many-core system. Examples of external resources are memory systems, sensors, controllers or neighbouring many-core array regions. As before, each process or interface to an external resource, is represented as a node with data transfers between nodes represented as edges. The nodes of the revised application process graph can be one of the following types:

- a *source node*, an interface to an external source of data received by an APG process, is a node that has no inbound edges
- a *process node*, has both inbound and outbound edges
- a *sink node*, an interface that is the recipient of data from an APG process, is a node that has no outbound edges

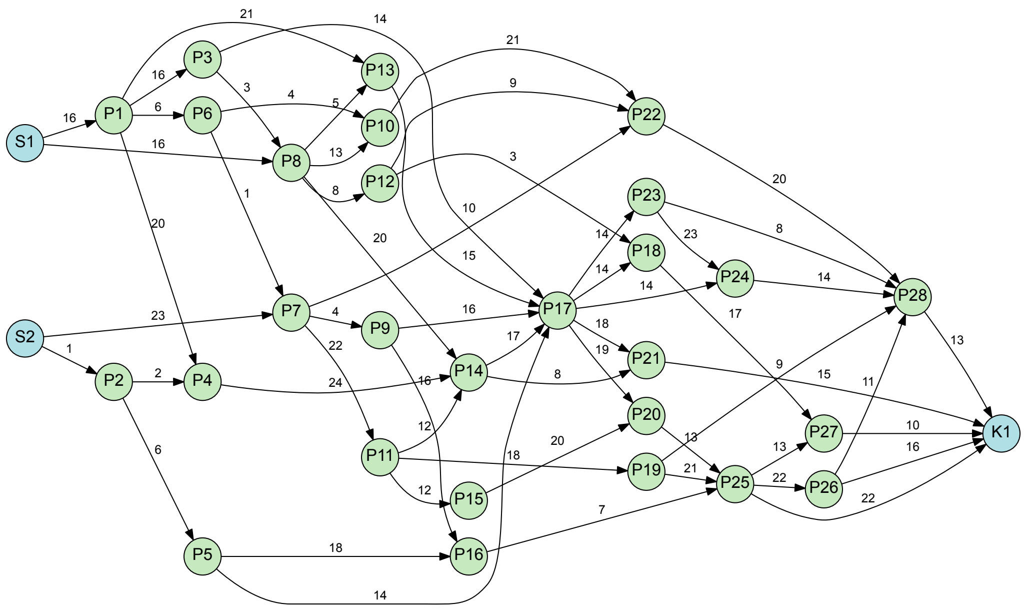


Figure 6.1 – Sparsely connected graph with 2 source nodes, 28 processing nodes and 1 sink node.

As in Section 4.3 the graph $\mathcal{G}$ is represented as a tuple of the set of nodes, $\mathcal{V}_g$, and the set of edges, $\mathcal{E}_g$:

$$\mathcal{G} := (\mathcal{V}_g, \mathcal{E}_g) \tag{6.1}$$

The set of nodes $\mathcal{V}_g$ is now the union of three distinct sets of nodes within the application process graph: the set of source nodes $\mathcal{S}_g$, the set of sink nodes $\mathcal{K}_g$, the set of process nodes $\mathcal{P}_g$:

Each node in the APG is identified by a label: process nodes by P$n \mid 1 \leq n \leq P_g$ where P_g is the number of process nodes in the graph, sources nodes by S$n \mid 1 \leq n \leq S_g$ where S_g is the number of source nodes in the graph and sink nodes by K$n \mid 1 \leq n \leq K_g$ where K_g is the number of sink nodes in the graph.

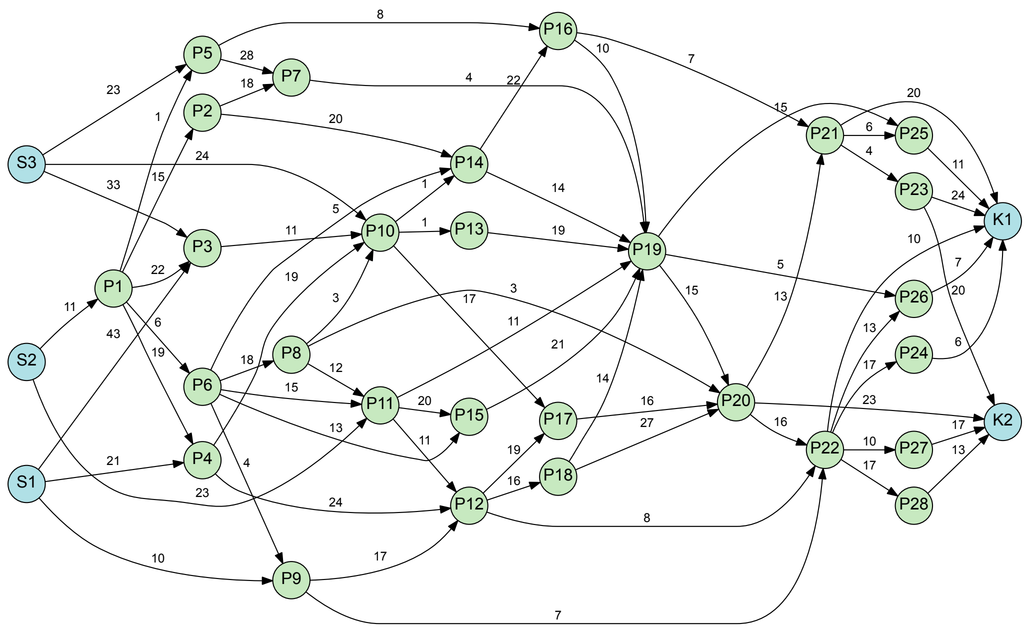


Figure 6.2 – Moderately connected graph with 3 source nodes ,28 processing nodes and 2 sink nodes.

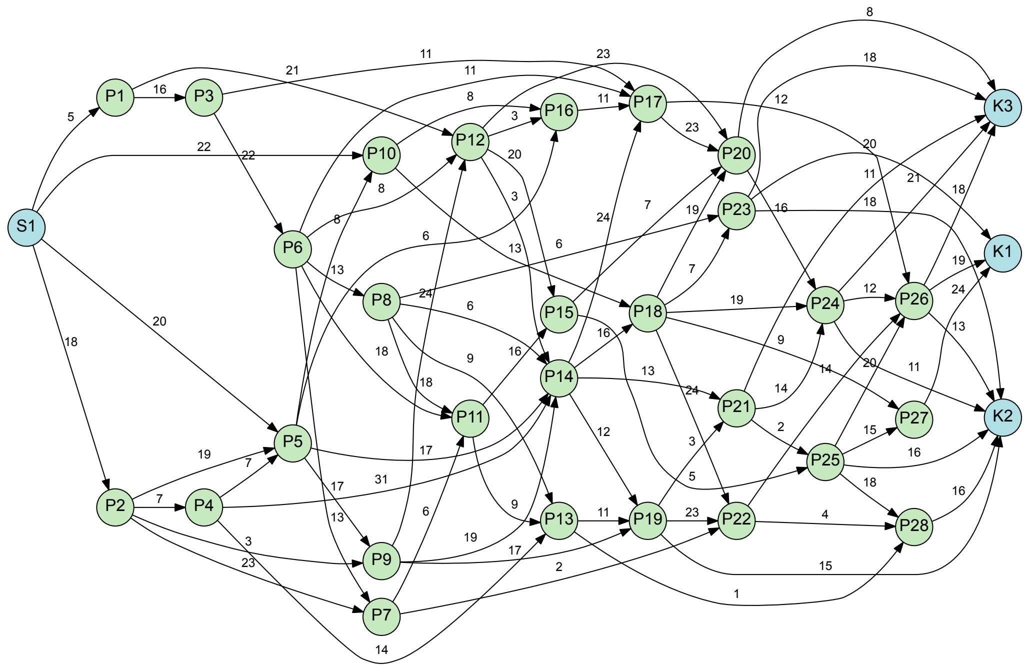


Figure 6.3 – Densely connected graph with 1 source node, 28 processing nodes and 3 sink nodes.

The total number of nodes is denoted by V_g, where:

$$V = |\mathcal{V}_g| \ = S + K + P \tag{6.2}$$

The number of edges is:

$$E = |\mathcal{E}_g| \tag{6.3}$$

The node sets are defined as:

$$\mathcal{S}_g = \{s_1, \ldots, s_n \mid 1 \leq n \leq S\} \tag{6.4}$$

$$\mathcal{K}_g = \{k_1, \ldots, k_n \mid 1 \leq n \leq K\} \tag{6.5}$$

$$\mathcal{P}_g = \{p_1, \ldots, p_n \mid 1 \leq n \leq P\} \tag{6.6}$$

$$\mathcal{V}_g = \mathcal{S}_g \cup \mathcal{K}_g \cup \mathcal{P}_g \tag{6.7}$$

$$\mathcal{E}_g = \{e_1, \ldots, e_n \mid 1 \leq n \leq E_g\} \tag{6.8}$$

An APG can be characterised by the three numbers representing the number of process nodes, source nodes and sinks nodes; for example the graph in fig 6.3 can be characterised by the three numbers $s = 1, p = 28, k = 3$ or, more compactly, by $V(1 : 28 : 3)$.

More densely connected graphs will, by definition, have more edges, which implies a greater total volume of traffic, which will affect metrics based on traffic volumes. To quantify the effect of graph density on the metrics, we categorized graphs as *sparsely connected*, *moderately connected* and *densely connected* depending on the density of edges in the graph. A sparsely connected graph is defined as a graph where there are less than two edges per node, a moderately connected graph is a graph where the number of edges is more than or equal to 2 edges per node and less than 2.5 edges per node, and a densely connected graph is a graph where the number of edges is more than or equal to 2.5 edges per node. These are arbitrary values which were found to be useful in this work.

The connection density of graphs is defined by Equation 6.9

$$C_g \begin{cases} sparse, & \text{if } \mathcal{E} < 2 \cdot \mathcal{V} \\ moderate, & \text{if } (\mathcal{E} \geq 2 \cdot \mathcal{V}) \wedge (\mathcal{E} < 2.5 \cdot \mathcal{V}) \\ dense, & \text{otherwise} \end{cases} \tag{6.9}$$

Where:

C_g	=	The connectivity of graph G.

The graph density also has an impact on the number of mappings that are link fault tolerant. Each edge creates a ComPair so with more edges creates more ComPairs, and the greater number of ComPairs makes it more difficult to arrange the processes in cores of the many-core array such that, the source node and target node of each ComPair are not on the same row and column i.e.there are no critical links.

6.2 Hardware Map

A *hardware map* or *network topology map* models, for a single region, the cores and links of the many-core array and the interfaces that connect the many-core array to external resources or neighbouring regions and the status of each element of the map.

The purpose of the hardware map is to model the essential elements of the many-core array and interfaces to external resources in sufficient detail to enable valid mappings to be generated and the metrics of the mappings to be calculated.

6.2.1 Nodes and Links of a Hardware Map

The definition of the hardware map is similar to the many-core array defined in Section 4.2 with the addition of information defining the type of each node and the status of each core and link.

The hardware map must maintain information to:

- Specify the type of each node
- Specify the status of each node and link
- Uniquely identify each node and link

Nodes
Nodes represent the cores of a many-core array or interfaces to resources external to the many-core array or cores of neighbouring regions. Interfaces to external systems can be classified as capable of acting as both sources or sinks, sources only or sinks only. The node types are summarised in Table 6.1.

Table 6.1 – Hardware Map Node Types

Type		Description
c	Core:	A node that can be used by an APG process node.
s	Source:	A node that can be used only by an APG source nodes.
k	Sink:	A node that can be used only by an APG sink nodes.
b	Both:	A node that can be used by an APG source or sink node.
r	Region:	A node that is either shared with or belongs to a neighbouring region.

Links
All links are unidirectional and are identical whether they are part of the internal circuitry of the many-core array or a link between the many-core array and external interfaces or regions.

Node Identity
Each node is given a location (c, r) starting with coordinate $(0, 0)$ at the top left most corner as illustrated in Figure 4.1.

Link Identity
Each link is directional so is identified by the location of the source node followed by the location of the target node $(loc(s), loc(t))$.

6.2.2 Modelling Faults

Node and Link Status
Each node and link in the hardware map has a status which indicates if the element is in good working condition or faulty as summarised in Table 6.2

Table 6.2 – **Core and Link Status Values**

Status		Description
g	Good:	The element is fully functioning.
f	Faulty:	The element is faulty and cannot be used.

Using the status of nodes and links the hardware map can model:

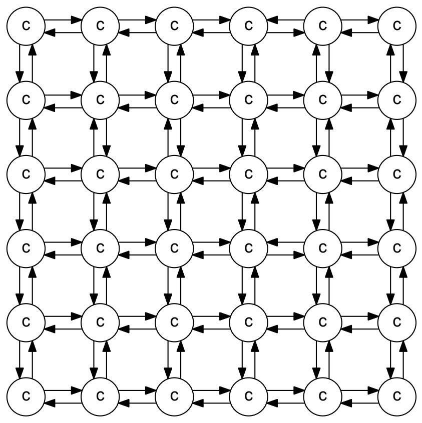
- Processing Core Faults
- Link Faults
- Routing Node Faults

A fault free hardware map and three types of hardware faults are illustrated in the hardware maps of Figure 6.4 with the faulty nodes and links being marked ***f*** and highlighted in red.

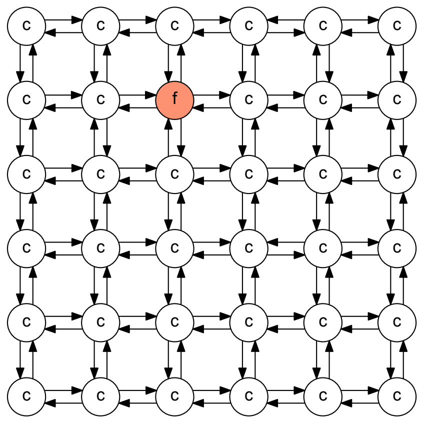
Processing Core Faults
Figure 6.4b illustrates a hardware map with a single faulty processing core, while the routing node and all links are functioning normally. Traffic passing though the routing node is unaffected.

Link Faults
Figure 6.4c illustrates a hardware map with two faulty links that are unable to carry any traffic.

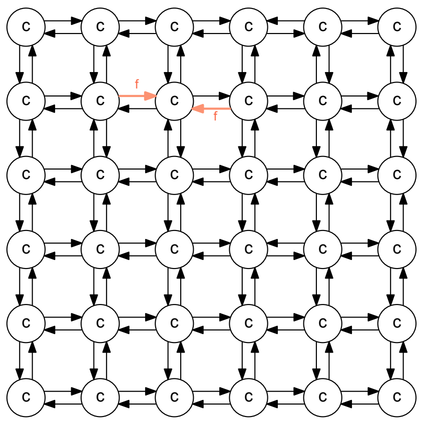
Routing Node Faults
Figure 6.4d illustrates a hardware map with a single faulty routing node, which is modelled by marking the node, and all 8 adjacent links, as failed. If the routing node fails then



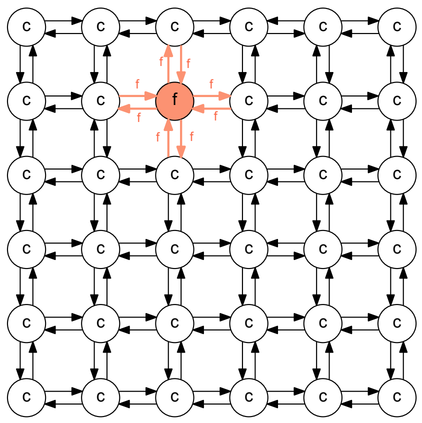
(a) Fault free



(b) A single core fault



(c) Two link faults



(d) A single routing node fault

Figure 6.4 – Hardware Maps representing a 6×6 many-core array with interfaces to external resource on each of the four edges of the many-core array.

the processing node becomes unavailable because it cannot receive or transmit any data; this allows the routing node to be modelled as a combination of a processing node failure together with the failure of all links attached to the routing node which is a considerable simplification compared to modelling the processing node and routing node separately.

6.2.3 Hardware Map Definition

The hardware map $\mathcal{H}$ is represented as a tuple of the set $\mathcal{V}_h$ of nodes and the set $\mathcal{E}_h$ of links:

$$\mathcal{H} := (\mathcal{V}_h, \mathcal{E}_h) \tag{6.10}$$

Given that there are R_h rows and C_h columns, the number of nodes and links are defined as V_h and E_h:

$$\begin{aligned} |\mathcal{V}_h| &= V_h \\ &= R_h C_h \end{aligned} \tag{6.11}$$

$$\begin{aligned} |\mathcal{E}_h| &= E_h \\ &= R_h(C_h - 1) + (R_h - 1)C_h \\ &= 2R_hC_h - R_h - C_h \end{aligned} \tag{6.12}$$

The set of nodes is defined as:

$$\mathcal{V}_h = \{v_1, \dots, v_m \mid qm = V_h\} \tag{6.13}$$

with a node defined as an ordered tuple consisting of the row and column coordinates of the node within the many-core array, which is also its location:

$$v_n = (loc, t, s) \tag{6.14}$$

Where:

$$loc = (r, c) \tag{6.15}$$

and:

r	is the row coordinate of node v_n.
c	is the column coordinate of node v_n.
t	is the type of the node $v_n t \mid t \in \{c, b, s, k, r\}$ as defined in Table 6.1.
s	is the status of the node $v_n \mid s \in \{g, f\}$ as defined in Table 6.2.

Row and column coordinates are arbitrarily defined as beginning at 0 and location $(0, 0)$ referring to the top-left hand core of the many-core array with location $(R_h - 1, C_h - 1)$ referring to the bottom-right hand core of the many-core array.

The set of links is defined as:

$$\mathcal{E}_h = \{e_1, \dots, e_m \mid m = E_h\} \tag{6.16}$$

A link is defined as a set of ordered tuples consisting of a source node location, a target node location and a status:

$$e_n = (loc_s, loc_t, s) \tag{6.17}$$

Where:

e_n	is a link from the set $\mathcal{E}_h$.
loc_s	is the location of the source node of link e_n.
loc_t	is the location of the target node of link e_n.
s	is the status of the link $e_n \mid s \in \{g, f\}$ as defined in Table 6.2.

6.2.4 Hardware Map Configurations

The hardware map is designed to be flexible so that it can model a range of configurations, a selection of which are illustrated in Figure 6.5. Configurations are defined through the use of the set of parameters listed in Table 6.3.

Table 6.3 – **Hardware Map Definition Parameters**

Parameter	Description
R_h	The number of rows of nodes in the hardware map
C_h	The number of columns of nodes in the hardware map
R_a	The number of rows of nodes in the many-core array
C_a	The number of columns of nodes in the many-core array
R_o	The position of the first of the row of the many-core array relative to the first row of the hardware map
C_o	The position of the first of the column of the many-core array relative to the first row of the hardware map
B_n	The border type of the north edge, see Table 6.4
B_w	The border type of the west edge, see Table 6.4
B_e	The border type of the east edge, see Table 6.4
B_s	The border type of the south edge, see Table 6.4
H_{df}	A dataflow machine is modelled when this boolean switch is set to ***true***
$L_{(h)}$	The set of links in the hardware map

Table 6.4 – **Border Types**

Type		Description
s	Source:	A node that can be used only by an APG source nodes.
k	Sink:	A node that can be used only by an APG sink nodes.
b	Both:	A node that can be used by an APG source or sink node.
r	Region:	A node that is either shared with or belongs to a neighbouring region.
n	None:	There is no border.

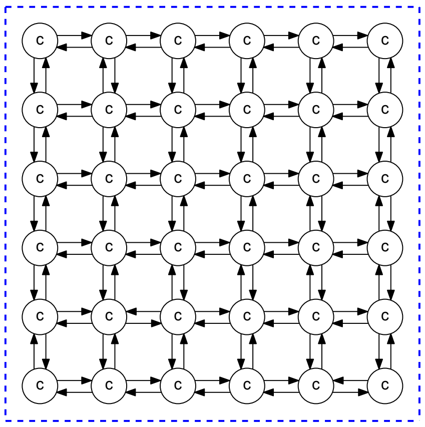
Border types are used to define how the nodes on the rows and columns, that are on the edges of the hardware map are configured. The parameters allow for borders to have of multiple rows or columns, which allow for a variety of multi-region arrangements. The types of borders are listed in Table 6.4.

A hardware map has four borders which are described using the four compass points of north, east, south and west. The four borders can be represented as a list of border types and width pairs, starting with the north border and working clockwise. For example, a hardware map that has a single row/column on each side that can be used for both sources and sinks can be described as $B((b,1),(b,1),(b,1),(b,1))$. If there is no border on one of the edges then the border is described using the border type and width pair of $(n,0)$.

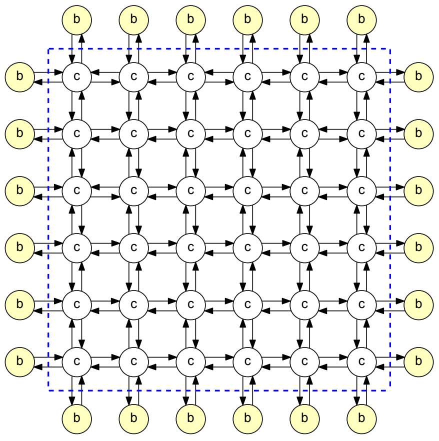
A selection of possible hardware map configurations are presented in Figure 6.5, each showing a 6×6 array of processing cores with different combinations of border types. The four example configurations are described below.

To fully specify a configuration, the following information is required:

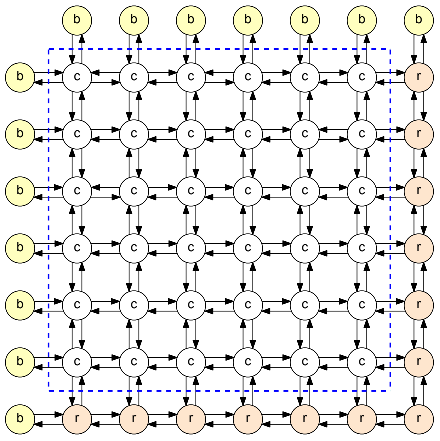
- The number of rows and columns of the hardware map.
- The number of rows and columns of the many-core array.
- The offset row and column of the many-core array with respect to the top-left corner of the hardware map.
- The type and width of the border of each edge of the hardware map.



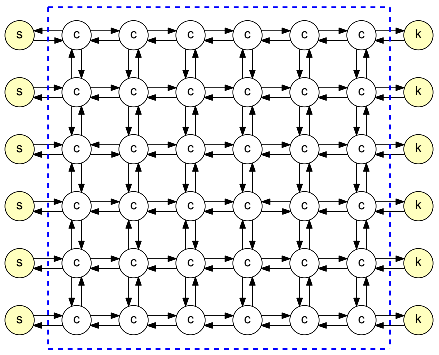
(a) A detached 6x6 Hardware Map with no source or sinks: $B((n,0),(n,0),(n,0),(n,0))$.



(b) A connected 8x8 Hardware Map were all borders can be either sources or sinks: $B((b,1),(b,1),(b,1),(b,1))$.



(c) A multi-region 8x8 Hardware Map for a with north and west borders that can be either sources or sinks and east and south borders that are neighbouring regions: $B((b,1),(r,1),(r,1),(b,1))$.



(d) A 6x8 Hardware Map for a Dataflow Machine with no borders on the north and south edges, sources on the west border and sinks on the east border: $B((n,0),(k,1),(n,0),(s,1))$.

Figure 6.5 – Hardware Maps for a Variety of Configurations. The nodes inside the blue dotted lines represent the cores of the many-core array while the nodes outside of the blue dotted box represent interfaces to resource external to the many-core array and cores of adjacent regions.

Configuration for Detached Many-Core Array
The many-core arrays being modelled in Figure 6.5a is defined as a *detached* many-core array with a 6×6 array of nodes with no external data sources or sinks, or neighbouring regions. Although this is an unrealistic configuration for practical systems, it was useful for the initial experiments in chapters 4 and 5. Since the configuration is modelling only cores and links of the many-core array, the hardware map and the many-core array have the same dimensions and the application process graph must consist only of processing cores. This hardware map has a border definition of $B((n,0),(n,0),(n,0),(n,0))$ and is the model used is most literature on the subject of fault tolerance in many-core arrays.

Configuration for Connected Many-Core Array
The hardware map configuration in Figure 6.5b is modelling a 6×6 *connected* many-core array which has sources and sinks to external resources but does not have any neighbouring many-core arrays or regions. The many-core array is located in the inner 6×6 nodes of the hardware map with source and sink communication ports to external resources modelled by the outside edges of the 8×8 hardware map. This hardware map has a border definition of $B((b,1),(b,1),(b,1),(b,1))$.

The connected configuration will be used in the majority of the experiments of this chapter.

Configuration for Multi-Region Many-Core Array
The hardware map configuration shown in Figure 6.5c models a many-core array that is part of a *multi-regio*n arrangement many-core arrays. The north and west edges have a border type of b indicating the the nodes can be either sources or sinks while the east and south edges have a border type of r to indicate that the nodes are either under shared control with a neighbouring region or under the exclusive control of a neighbouring region. This hardware map has a border definition of $B((b,1),(r,1),(r,1),(b,1))$.

Configuration for Dataflow Many-Core Array
The hardware map in Figure 6.5d is an arrangement that forces all source nodes to be located on the left edge of the array and all sink nodes to be located on the right edge of the array. This arrangement will ensure that all data arrives via the west edge of the array and leaves via the east edge, producing a flow of data from west to east, hence the name *dataflow machine*. This hardware map has a border definition of $B((n,0),(k,1),(n,0),(s,1))$.

6.3 The Environment

The application process graph is a model of the application processes and its sources and sinks. The hardware map is a model of the many-core array and the connections between the many-core array and external resources. A process map is the mapping of process nodes from the application process graph to cores in the many-core array.

Since the process map only includes process nodes from the application process graph, an additional mapping is required to map the application process graph source and sinks nodes and nodes belonging to adjacent regions; this is the role of the environment.

This section describes the environment, and the relationship between the application process graph, the hardware map, the environment and the process map.

The Environment Mapping

The environment maps the resources that are external to the many-core array to the borders defined in the hardware map. Application process graph source nodes can be mapped to nodes in borders of type b or s while sink nodes can be mapped to nodes of borders of type b or k and nodes of neighbouring regions are mapped to nodes of borders of type r.

The environment mapping is established at the beginning of an evolution and remains fixed for the duration of the evolution. The calculation of the metrics of process maps that are generated during the evolution are affected by the position of resources in the environment map and are therefore calculated with respect to the environment mapping.

From the hardware map in Figure 6.6a, that is a connected many-core array with a border of $B((b,1),(b,1),(b,1),(b,1))$ and the application process graph of Figure 6.2, the environment map of Figure 6.6b and process map of Figure 6.6c are created, the environment map containing the source and sink nodes from the APG and the process map contain for the processing nodes of the APG. The illustrations of the hardware map, the environment map and the process map all include a dotted blue line, the *many-core array boundary*. Inside the boundary are the nodes relating to the many-core array, while the nodes outside of the boundary relate to the environment.

The environment map consists only of the nodes outside of the many-core array boundary while the process map consists only of the nodes inside the boundary.

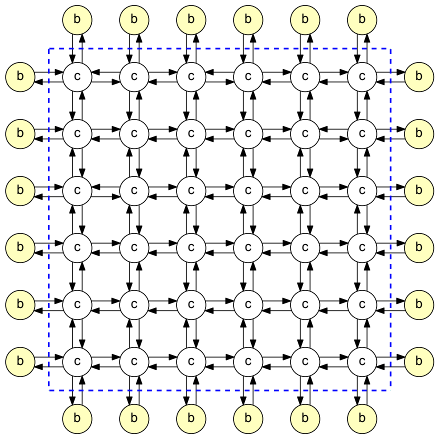
The environment and process maps only tell us which cores in the hardware map each node of the APG is mapped to but do not have any information regarding the status of links between the nodes. In the illustrations of environment and process maps, the lines linking the nodes are included to illustrate that the nodes are connected by links. The status of links is maintained in the hardware map.

The Aggregate Map

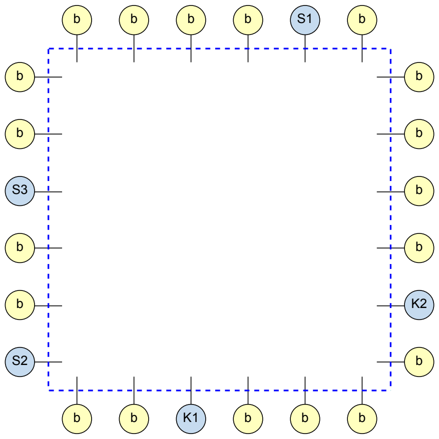
The calculation of metrics for a process map can only be made within the context of the environment. The metrics are therefore calculated using an *aggregate map* which is a combination of the environment map and process map and illustrated in Figure 6.6d.

6.4 Link Fault Tolerance Metric and Objective

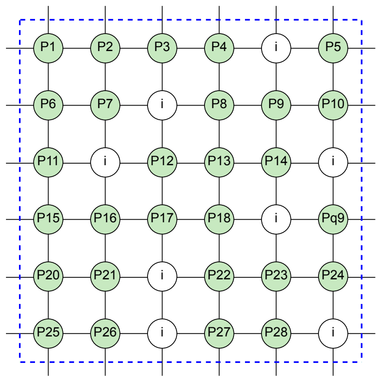
Failure of a link can result in disruption to communication in the array by causing packets to take longer routes, lose packets or sever communications completely. The exact effect a failed link has on communications of the array depends of how critical the link is for



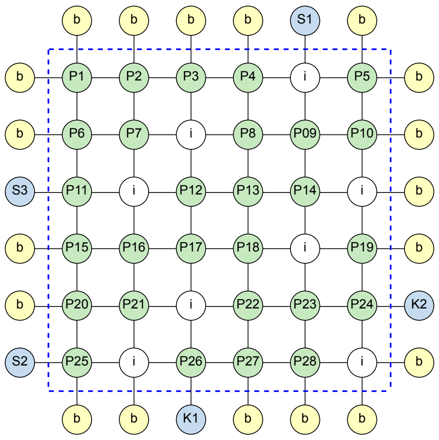
(a) An 8x8 Hardware Map where all borders can be either sources or sinks.



(b) An environment map of the source and sink nodes.



(c) A process map of the processing nodes.



(d) An aggregate map which is the combination of environment map (b) and process map (c).

Figure 6.6 – From the information in the APG in Figure 6.2 and the hardware map (a) the environment map (b) and process map (c) are produced. The environment map and process maps are then combined to make the aggregate map (d) used to calculate the metrics of the process map.

each ComPair that uses the link. In this section link failures will be explored and a metric developed that is designed to measure the detrimental effect the failure of a link would have on the network.

The metrics developed in this section relate to a single ComPair, a ComPair. The objective combines the metrics for all ComPairs into a single value representative of the vulnerability of the whole mapping to link failure.

6.4.1 Problem Description

Develop a metric that measures the effect that failure of a link will have on network traffic and an objective that can be used to minimize the negative effects of possible link failures.

The definition of the metrics in this section refers to discussion of link criticality and the accompanying ComPair subnets in Section 5.2 *Link Criticality*. For convenience the ComPair subnets are reproduced here as Figure 6.7.

6.4.2 Vulnerability to Critical Link Failure Metric

A failed critical link causes communication failure because it is used by all paths between the source and target nodes of the ComPair . Comparing figures 6.7a and 6.7b, it can be observed that the graph of Figure 6.7a has a single path consisting of a single critical link and the graph of Figure 6.7b has a single path consisting of two critical links. In this respect the graph of Figure 6.7b can be described as being twice as vulnerable to critical link failure when compared to Figure 6.7a. It is also possible for a critical link to have multiple paths using it as shown in Figure 6.7i. Failure of such a link will sever all the paths that pass through it so the metric must reflect the number of paths affected by the link.

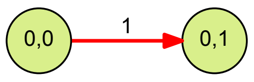
The number of paths of a ComPair subnet that include a link l_n is given by l_{n_p} and by Equation 5.6 a link is critical when $l_{n_p} = Q_p$. The metric $Mvclf_c$ is a measure of the *Vulnerability to Critical Link Failure* of a ComPair, is defined as the sum of the number of paths that pass through each of the critical links in the ComPair subnet. Metric $Mvclf_c$ is given by Equation 6.18 which include the equivalence that makes use of the fact that all critical links, by definition, must be included in all paths of the ComPair.

$$Mvclf_c = \sum_{l=1}^{Q_l} \begin{cases} l_{n_p}, & \text{if link } l_{n_p} = Q_p \text{ i.e. is a critical link} \\ 0, & \text{otherwise} \end{cases} \equiv Q_p \cdot l_c \tag{6.18}$$

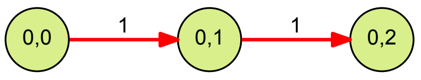
Where:

$Mvclf_c$	=	The vulnerability to critical failure metric for the ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
Q_p	=	The total number of paths of the ComPair subnet.
l_c	=	The number of critical links in a ComPair

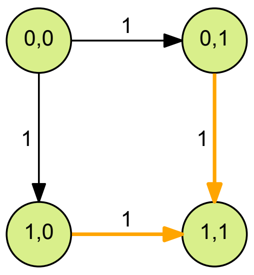
Equation 6.18 will give a value of 1 for the ComPair illustrated in Figure 6.7a and give a metric value of 2 for the ComPair illustrated Figure 6.7b which is consistent with the notion that 6.7b is twice as vulnerable as 6.7a.



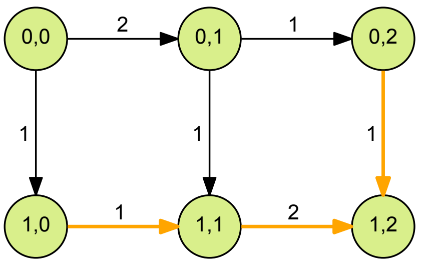
(a) Critical Link of length 1



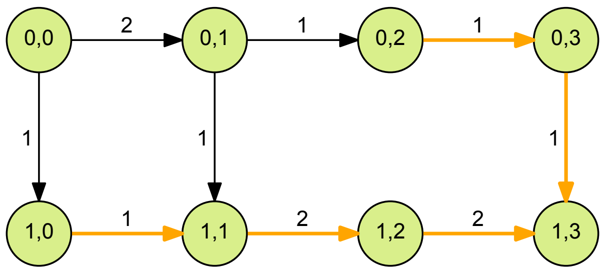
(b) Critical Link of length 2



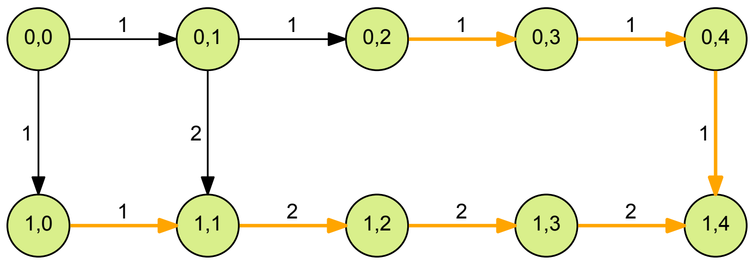
(c) Network with 2 Significant Links



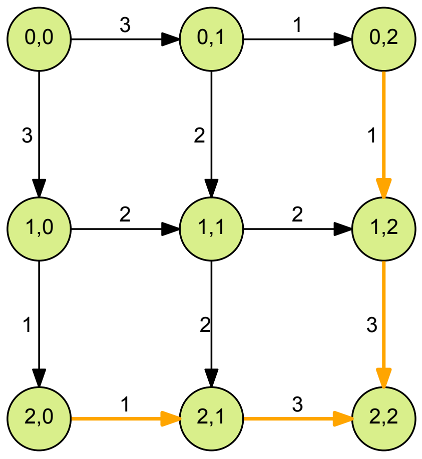
(d) Network with 3 Significant Links



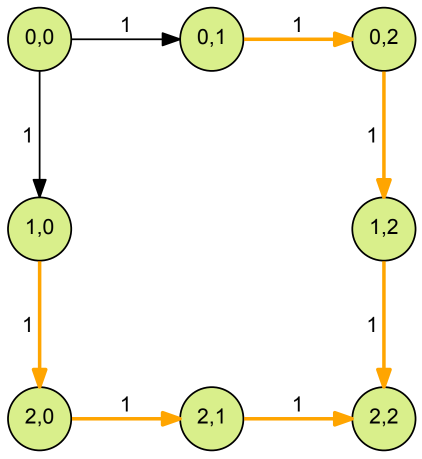
(e) Network with a single Link fault and 5 Significant Links



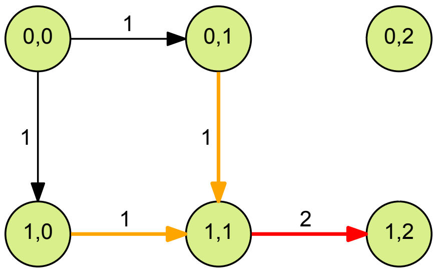
(f) Network with a 2 Link faults and 7 Significant Links



(g) Network with 4 Significant Links



(h) Network with a 4 Link faults and 6 Significant Links



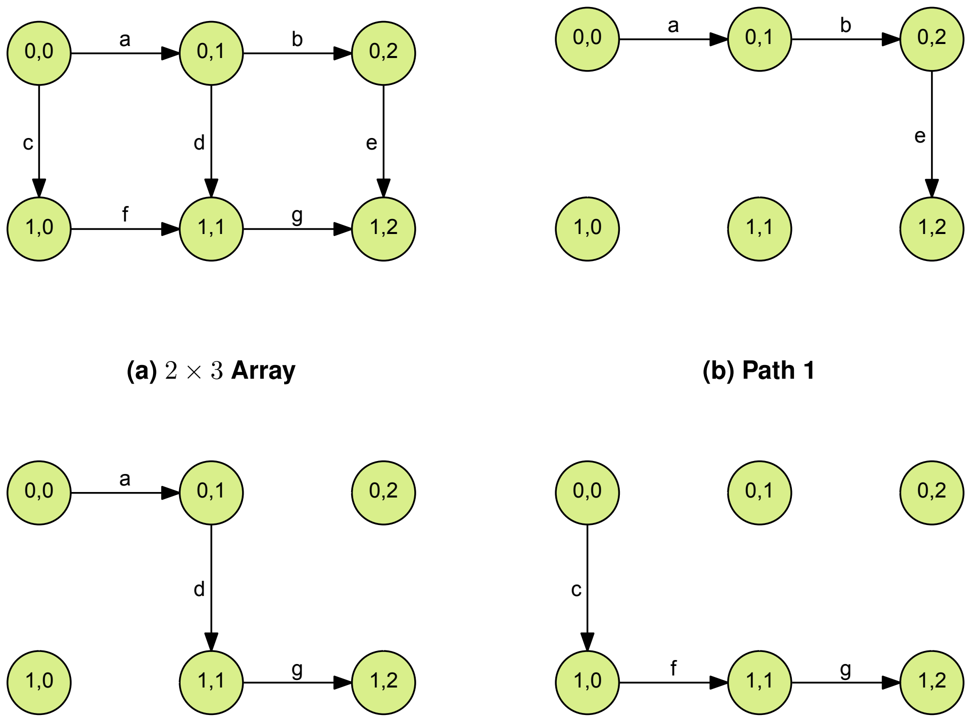
(i) Network with 2 Link faults, 2 Significant Links and 1 Critical Link

Figure 6.7 – Networks illustrating Critical and Significant links

6.4.3 Vulnerability to Significant Link Failure Metric

Failure of a significant link is less disruptive than a critical link because, although some packets may be lost, there are alternative paths for the packets from the source to the target. A metric is required to give a value to represent the impact of the failure of a significant link.

Failure of a significant link can affect multiple paths of a ComPair as illustrated in Figure 6.8 where significant link g is a component of paths 2 and 3, while significant links e and f are both on only one path. The impact of link g failing is more severe than if link e or f fails. To account for this, it is important to know the number of paths each significant link affects. The links of the graphs in Figure 6.7 are labelled with the number of paths that use the link. If the number of paths that use each significant link are added together for a ComPair then



(a) 2×3 **Array** **(b) Path 1**

(c) Path 2 **(d) Path 3**

Figure 6.8 – Paths Through a 2×3 Array

this gives a metric for the *vulnerability to significant link failure* for the ComPair.

$$Mvslf_c = \sum_{l=1}^{Q_l} \begin{cases} l_{n_p}, & \text{if link } l_n \text{ is a significant link} \\ 0, & \text{otherwise} \end{cases} \tag{6.19}$$

Where:

$Mvslf_c$	=	The vulnerability to significant link failure for ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
Q_p	=	The total number of paths of the ComPair subnet.

6.4.4 Vulnerability to Link Failure Metric

The remaining task is to integrate the two metrics of *Vulnerability to Critical Link Failure* and *Vulnerability of Significant Link Failure* into a single metric that gives a single measure of the *Vulnerability to Link Failure*. Normal links are not included in the metric because they do not cause loss data. The definition of both of the above metrics is identical in that they both measure the number of paths that are affected by the failure of the link. As an initial attempt at creating a single metric, the individual metrics are summed together to produce a single value representing the total number of paths that are affect by a critical or significant link. Figure 6.8 list values of the sum of the two metrics in the column $\sum$ under "Vulnerability Metric", for the subnets in Figure 6.7.

Table 6.5 – Comparison of Vulnerability Metrics

Graph	Graph Dimension		Paths		Vulnerability Metric		
Ref	Height	Width	Total	Affected	$\sum$	$100 \times \frac{\sum}{C_p}$	$100 \times \frac{\sum}{C_p^2}$
6.7a	1	2	1	1	1	100	100
6.7b	1	3	1	1	2	200	200
6.7c	2	2	2	2	2	100	50
6.7d	2	3	3	4	4	133	44
6.7e	2	4	3	7	7	233	77
6.7f	2	5	3	10	10	333	111
6.7g	3	3	6	8	8	133	22
6.7h	3	3	2	6	6	300	150
6.7i	1	2	2	2	4	200	100

Relating the metric values to the graphs, it is evident that this simple sum does not give a consistent value for the purposes of comparing the vulnerability of different graphs. Graph Figure 6.7a has the best fitness of 1, while graphs Figure 6.7b and Figure 6.7c have the same fitness of 2. However graph 6.7c is clearly less vulnerable than either 6.7a or 6.7b. The reason is that graph 6.7c is a larger graph and has more paths, so there are more paths that can be affected and the metric is greater. This implies that the metric should take into account the number of paths in the graph between the source and target nodes.

Two additional metrics have been calculated, which are the sum divided by the number of paths in the ComPair and the sum divided by the square of the number of paths in the ComPair, in an attempt to correct the deficiencies of the simple sum metric. Dividing by the number of paths improves the metric but still gives anomalous values, for example the graph of Figure 6.7b and 6.7c have the same metric value even though the graph 6.7c has two paths so is clearly less vulnerable than Figure 6.7b. Dividing by the square of the number of paths gives comparative values that work well, for example the metric value for graph 6.7h is three times greater than the metric value for 6.7c because, although they both have two paths, 6.7h has three times more significant links than 6.7c, so is three times more vulnerable. The additional metrics are shown in columns $\frac{\sum}{p_l}$ and $\frac{\sum}{p_l^2}$ with the values having been multiplied by 100 to give useful integer values.

The final metric calculation for *Vulnerability to Link Failure* is given by Equation 6.20.

$$Mvlf_c = \frac{100}{{C_p}^2} \sum_{l=0}^{C_l} \begin{cases} l_{n_p}, & \text{if link } l \text{ is a critical or significant link} \\ 0, & \text{otherwise} \end{cases} \tag{6.20}$$

Where:

$Mvlf_c$	=	The vulnerability to significant link failure for ComPair C.
C_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
C_p	=	The total number of paths of the ComPair subnet.

6.4.5 Link Fault Objective

This section describes an objective that combines the metrics for all ComPairs into a single value representative of the vulnerability of the whole mapping to link failure.

The *link fault objective* is to minimize the sum of the metric values for all ComPairs in the application process graph. The link fault metric gives a value for a single ComPair so the value for the objective is the sum of the fault metric values for all ComPairs in the application process graph:

$$Jlink = \sum_{l=1}^{C_l} Mvlf_c \tag{6.21}$$

Where:

$Mvlf_c$	=	The vulnerability to link failure for ComPair C.
C_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
C_p	=	The total number of paths of the ComPair subnet.

6.5 Network Power Metric and Objective

Communication traffic is a major contributor to power consumption in a many-core array [183, 184]. The total traffic flows in the many-core array can be used as an approximation for power consumed by communication traffic. This section will explore the measurement of the traffic in a many-core array and develop an objective designed to direct a search algorithm to find solutions that minimize the traffic.

6.5.1 Problem Description

The power consumed by traffic between a ComPair, for a given router/NoC architecture, is a function of both the traffic volume and the distance in terms of routing nodes and links that the traffic has to traverse between the source and target nodes. Reducing the total network traffic of all ComPairs will reduce the power consumption of the many-core array. The volume of traffic between a ComPair is predetermined by the application and cannot be influenced by the many-core system whereas the distance the traffic has to travel is dependent on the relative position of the source process node and the target process node, which is under the control of the many-core system.

6.5.2 Distance Between Communicating Core Pairs

To relate traffic to power it is necessary to know both the amount of traffic and the distance it has to travel, as the product of these gives us the total amount of traffic that is processed by the routing nodes and transmitted down the links between the source and target nodes. The distance between a ComPair is the rectilinear distance between the source and target nodes and, assuming minimal length paths, represents the number of links the data will travel through from the source node to the target node. Faults on paths between the source and target have no effect on the length of the path that the data will travel along, providing the faults do not sever all paths between the source and target. For a ComPair Q_n corresponding the edge e_n of the application process graph with source location $Q_n(s_r, s_c)$ and target location of $Q_n(t_r, t_c)$ the *Distance Between a ComPair* metric $Mdccp_{q_n}$ (i.e the distance between the source node and the target node), is given by Equation 6.22.

$$Mdccp_{q_n} = |(s_r - t_r)| + |(s_c - t_c)| \tag{6.22}$$

If the assumption is made that traffic volume between all ComPairs is the same then the distance between cores can itself be used as an approximation for the traffic and therefore the power consumption. This is not a particularly good assumption as the traffic between different ComPairs can be very different, so a more precise calculation is desired.

6.5.3 Traffic Volume Metric

An improvement to the distance between ComPairs as a metric is to use a measure that takes into account both the distance and the traffic volume for each ComPair, which will typically be different for each pair.

For a ComPair Q_n corresponding the edge e_n of the application process graph the traffic volume Q_{n_d} is the attribute traffic volume, d, from edge e_n of the application process graph. A metric that measures the *Traffic Volume Between a ComPair*, $Mtccp_{q_n}$, is given by the Equation 6.23.

$$Mtccp_{q_n} = Mdccp_{q_n} \times Q_{n_d} \tag{6.23}$$

6.5.4 Power Objective

The corresponding power objective is to minimize the total power consumption across the whole network.

For the objective, a power consumption of zero is assigned when the source and target of a ComPair are adjacent, so that if the source and target nodes of every ComPair are adjacent then the value for the mapping will be zero. However, when a ComPair's source and target nodes are adjacent, the traffic volume metric $Mtccp_{q_n}$ will have a value of Q_{n_d}.

Therefore when the traffic metric is used in the objective it must be adjusted down by the value of Q_{n_d}. The power objective is given in Equation 6.24.

$$Jpower = \sum_{e=1}^{E_g} Mtccp_{q_n} - Q_{n_d} \tag{6.24}$$

6.6 Excess Traffic Metric and Objective

This section examines traffic flow through the many-core array that can lead to *excess traffic*. Each link in a many-core array may carry traffic from multiple ComPairs, the total of which could exceed the bandwidth of the link. The excess traffic is the traffic above the bandwidth of a link that the link is required to carry. This can be described as a link-centric with, since the excess traffic is related to each individual link, although the details of the traffic from all the ComPairs that have paths that use the link.

A link that is required to carry excess traffic is referred to as an *overloaded* link. Excess traffic for a link will eventually fill the buffers of the routing node, creating a bottleneck so potentially causing disruption throughout the network as traffic above the link's capacity is rerouted down alternative paths which may then overload other links causing more rerouting which can result in a cascade of overloaded links. If there are alternative paths to a path that uses an overloaded link, then the effect of the bottleneck may be mitigated by using an adaptive routing algorithm.

A routing node can monitor the status of the links it is using to transmit data. If a bottleneck develops on a link, the routing node is able to detect this and then employ an adaptive routing algorithm to redirect traffic down alternative paths, if they exist, using another link. In doing so, the alternative paths must still be minimal-length paths. If no alternative paths exists for the traffic, because the traffic is critical traffic, adaptive routine cannot alleviate the bottleneck. The excess traffic metric is only an approximation of the expected traffic volumes based on the traffic volume data containing the the application process graph. Only a cycle accurate simulation, with knowledge of how each process generates traffic and an understanding of the intended routing algorithm, will be able to accurately predict traffic volumes. As discussed in Chapter 7 *Graceful Degradation and Amelioration*, one of the roles of the Monitor will be to update the traffic volume attributes of the edges in the application process graph with actual observed traffic volume data to increase the accuracy of calculations of metrics and objectives.

Critical traffic cannot be rerouted; significant traffic can be rerouted by the first routing node, back up the path, that has a choice of paths for transmission of the data; normal traffic can be rerouted by the routing node link. The severity of the effect of a bottleneck will therefore depend on whether the link is carrying critical, significant or normal traffic.

Each ComPair in a mapping will have at least one, and often many, paths between the source and the target through which traffic will travel. Each path will consist of at least one link, and typically a chain of links. Each link in the array will typically be part of many paths between many ComPairs and the role of the link in each path may be either critical, significant or normal. As a result the *traffic profile* of each link is complex, consisting of the aggregation of a variety of types of traffic from many paths between many ComPairs.

In this section, a metric will be developed to represent the excess traffic of a link and evolutionary algorithm objectives designed to minimize the potential disruption of excess traffic.

6.6.1 Problem Description

The impact an overloaded link has on the network as a whole will depend of the volume of each traffic type that uses the link. Critical traffic has no other option than to use the link, so a link that is overloaded with critical traffic will cause an unavoidable bottleneck. Significant traffic, does have other available routes, but these can only be used when the congestion of traffic has propagated back up the path until it reaches a node than can direct traffic down an alternative path. Excess normal traffic can be rerouted immediately by a congested link's routing node, so will have lower impact than critical or significant traffic. This suggest the concept of *weighted excess traffic* metric which is a measure of excess traffic obtained by giving a weight to each type of excess traffic so that critical traffic makes a greater contribution to excess traffic metric than significant traffic, which in turn makes a greater contribution to the metric than normal traffic. This penalizes mappings with excess critical and excess significant traffic. For a fault free array, critical links can be avoided completely while significant links can be reduced in number but not eliminated completely since the links attached directly to the target node cannot be normal (see Section 5.1 *Communicating Core Pair (ComPair)*).

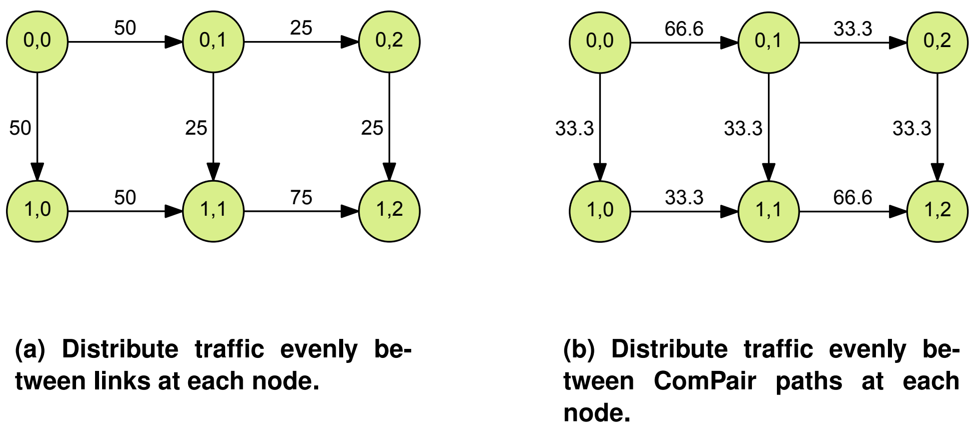
A metric that measures the excess traffic of a link can only suggest which links could be overloaded in normal operation. Calculating the expected traffic through each link, using either the non-weighted or weighted calculations, is only an approximation used to reduce computational time, based on the average predicted network load taken from the application process graph, which might prove to be very different to the actual traffic resulting from adaptive routing. In a working many-core array the on-line Monitor, discussed in Chapter 7 *Graceful Degradation and Amelioration*, will be be able to detect real bottlenecks and actual data flows between ComPairs, which can then be used in these metrics to increase accuracy.

6.6.2 Excess Traffic Metric

In this section the metric will be developed starting with a simple calculation using raw traffic followed by a more sophisticated approach involving link level traffic weighting.

Traffic Distribution
Traffic flowing between the source and target of a ComPair will be distributed amongst the

available paths by the routing algorithm used by the routing nodes between the source and the target nodes. Two algorithms for distribution of the traffic through the ComPair subnet have been considered. The first is that the traffic is split equally between the available paths at each routing node, illustrated in Figure 6.9a; the second it that the traffic is split equally between the paths from the source to the target, illustrated in Figure 6.9b. Both strategies would require an adaptive routing algorithm that has the ability to understand the strategy and keep track of the traffic that has been sent down each link so that it sends each new packet down the link that will maintain the desired balance.



(a) Distribute traffic evenly between links at each node.

(b) Distribute traffic evenly between ComPair paths at each node.

Figure 6.9 – Traffic distribution across the links used by a ComPair. The numbers on the links are percentages of the total ComPair traffic.

When the traffic is divided equally between links at each node (Figure 6.9a) the minimum and maximum traffic volumes on individual links is between are 25% and 75%, giving a range of 50% When the traffic is divided equally among the ComPair paths at each node (Figure 6.9b) the minimum and maximum traffic volumes on individual links is between 33.3% and 66.6% giving a range of 33.3%. As it is preferable for the traffic to be as evenly distributed as possible the smaller range of 33.3% is more attractive than 50%. Consequently in the sections that follow the traffic will be evenly distributed between ComPair paths.

Non-Weighted Excess Traffic

The starting point is to calculate the actual traffic that is expected to flow through a link. The method is to consider each ComPair, and for each pair calculate the traffic that will travel through each link between the source and target nodes of the ComPair. Between the source and target nodes there may be many paths. In many cases paths will overlap i.e. they will use links that are used by other paths.

As an example, take a ComPair with source and target nodes that are one row and two columns apart as in Figure 6.8a. There are a total of three paths in the ComPair, illustrated in Figures 6.8b, 6.8c and 6.8d. Links b, c, e and f are used by only one path while links a, d and g are used by multiple paths. The volume of traffic between each ComPair is taken from the application process graph which provides a traffic volume for each edge in the graph. Traffic for a ComPair is distributed evenly between each of the paths between the source and target, as discussed in Subsection 6.6.2. Traffic that travels along a path must travel through each link of the path. Each path is analysed in turn with traffic from each path being added to each link in the path. Each ComPair must be analysed and traffic added to each link in each path of each ComPair to produce a total traffic for each link.

Having calculated the total traffic for each link the *excess traffic* is calculated which is the traffic above the bandwidth for the link. Each overloaded link will have a value for excess traffic that is greater than zero while links that are not overloaded will have an excess traffic value of zero.

The traffic volume through a link l_n of a ComPair is the network load, Q_d, of the ComPair multiplied by the proportion of paths for the ComPair that include the link and is given by Equation 6.25.

$$l_{n_t} = \frac{l_{n_p}}{Q_p} \cdot Q_d \tag{6.25}$$

Where:

l_{n_t}	=	the traffic through link l_n of ComPair Q
l_{n_p}	=	the number of paths through link l_n of ComPair Q
Q_p	=	the total number of paths in ComPair Q
Q_d	=	the network load of ComPair Q

Given that there is a mapping from each link in a ComPair path to a link in the hardware map represented by the function $f : L_{(q_e)} \rightarrow L_{(h)}$ each link l_m of the hardware map has an associated set of traffic volumes from the ComPair paths that use the hardware link l_m. If each of the paths from all ComPairs of the mapping is assigned an integer identifier from 1 to Q_{tp}, where Q_{tp} is the total number of paths from all ComPairs of the mapping, then the traffic volume $t_{(l_m,q)}$ is the traffic contributed from path q to hardware link l_m. The traffic volumes for each ComPair link calculated using Equation 6.25 can be mapped to a traffic volume $t_{(l_m,q)}$ which allows the definition of traffic volume metrics in terms of $t_{(l_m,q)}$.

The metric of total traffic for link l_m is the sum of all traffic volumes from the ComPair paths that use the link and is given by Equation 6.26.

$$Mtt_{(l_m)} = \sum_{q=1}^{Q_{tp}} t_{(l_m,q)} \tag{6.26}$$

The excess traffic for link l_m is given by Equation 6.27

$$Mxt_{(l_m)} = \begin{cases} 0, & \text{if } Mtt_{(l_m)} \leq l_{m_b} \\ Mtt_{(l_m)} - l_{m_b}, & \text{otherwise} \end{cases} \tag{6.27}$$

Where:

$Mtt_{(l_m)}$	=	The total traffic through link l_m.
$Mxt_{(l_m)}$	=	The excess traffic through link l_m.
l_m	=	The m_{th} link of the hardware map.
l_{m_b}	=	The bandwidth of link l_m.
Q	=	The number of paths of all ComPairs for the mapping.
$t_{(l_m,q)}$	=	The traffic through link l_m contributed by path q.

Weighted Excess Traffic

A more sophisticated approach is to apply weighting to traffic through a link, based on the criticality of the link within each ComPair, producing an artificially high traffic value when the link is critical or significant. The rationale for this approach is that if a link is a critical link, then there are no alternative paths for the data, so the effect of the link being overloaded will be more severe than if the link is not a critical link. Similarly, if a link is a significant link, then there is only a single path between the significant link and the target node of a ComPair even though there is at least one other alternative path from the source node to the target node of the ComPair. The effect of a significant link being overloaded will be less severe than for a critical link but more severe than for a normal link. This section will propose a method of modifying the value of the traffic metric by applying weighting to traffic passing though different types of link.

For each link in a path the link may be critical, significant or normal. A link may be a critical link in one path, while being a significant link in another path and a normal link in yet another path.

The approach taken is to calculate the total critical, significant and normal traffic through a link from all the paths that use the link, then apply weighting in the following manner:

1) If the critical traffic is above the link's bandwidth then the weighted traffic is calculated by taking the excess critical traffic and applying the *critical traffic weight*, adding the significant traffic weighted by the *significant traffic weight* and adding the normal traffic.

2) If the critical traffic is below the link's bandwidth then the sum of critical traffic and significant traffic is compared to the link's bandwidth, and if greater than the bandwidth then the weighted traffic is the excess of the sum of critical traffic and significant traffic weighted using the significant traffic weight added to the normal traffic.

3) If sum of critical traffic and significant traffic is below the the link's bandwidth, then the sum of the critical traffic, significant traffic and normal traffic is compared to the link's bandwidth, and if greater than the bandwidth then the weighted traffic is the excess of the sum of critical traffic the significant and the normal traffic.

4) If the sum of the critical traffic, significant traffic and normal traffic is below the link's bandwidth then the weighted traffic is zero.

The weighting applied to critical traffic is greater than the weighting applied to significant traffic, while the normal traffic is not weighted. The initial values for weights are set at 5 for critical traffic and 2 for significant traffic. These values are arbitrary and will be explored by experiments.

Table 6.6 shows the *non-weighted excess traffic (NWET)* and the *weighted excess traffic (WET)* for a selection of bandwidths and fixed traffic on a link, illustrating that the weighted traffic calculation produces larger traffic and excess traffic values than the non-weighted calculation for the same traffic profile. For bandwidths above 500 the actual traffic is less than the bandwidth, but the presence of critical and significant traffic exaggerates the calculated traffic which would cause the search algorithm to discriminate against such a mapping in favour of one with less critical and significant traffic.

Table 6.6 – Comparison of Non-Weighted and Weighted Excess Traffic

Traffic	Excess	Link	Weighted	Bandwidth			
Criticality	Weight	Traffic	Traffic	500	1000	1500	2000
				Weighted Excess Traffic			
Critical	5	200	1000	500	0	0	0
Significant	2	200	400	400	400	0	0
Normal	1	600	600	600	600	500	0
WET Total			2000	1500	1000	500	0
NWET Total		1000		500	0	0	0

Table 6.7 shows a variety of traffic profiles for a fixed bandwidth illustrating how the weighted traffic calculation affects the excess traffic when compared to the non-weighted traffic. The four different traffic profiles are indistinguishable when weighting is not used. When weighting is used there is a clear difference in the calculated traffic values between the profiles, which will direct the search algorithm to select solutions with lower critical and significant traffic.

Table 6.7 – Comparison of Non-Weighted and Weighted Excess Traffic

Traffic	Excess	Bandwidth : 5000							
Criticality	Weight	Tr	WET	Tr	WET	Tr	WET	Tr	WET
Critical	5	1500	2500	1000	0	500	0	500	0
Significant	2	1000	2000	1500	3000	1000	0	500	0
Normal	1	500	500	500	500	1500	1000	2000	500
Weighted Total		10000	5000	8500	3500	6000	1000	5500	500
Non-Weighted Total		3000		3000		3000		3000	

The excess traffic metric is calculated for each link in the network using the following set of equations. First the critical, significant and normal traffic components through each link from each path are defined given that $T_{(l,q)}$ is the traffic through link l contributed by path q.

$$Mct_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is critical for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{6.28}$$

$$Mst_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is significant for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{6.29}$$

$$Mnt_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is normal for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{6.30}$$

Now the components for each traffic type for each link are summed to give a total of each traffic type for each link.

$$Mct_l = \sum_{q=1}^{Q} Mct_{(l,q)} \tag{6.31}$$

$$Mst_l = \sum_{q=1}^{Q} Mst_{(l,q)} \tag{6.32}$$

$$Mnt_l = \sum_{q=1}^{Q} Mnt_{(l,q)} \tag{6.33}$$

Now the excess traffic for each link can be defined in terms of each of the individual traffic types and the traffic type weights.

$$Mxwt_l = \begin{cases} (Mct_l - Bw_l) \times Wc + & \\ \quad Ts_l \times Ws + Mnt_l, & \text{if } Mct_l > Bw_l \\ (Mct_l + Ts_l - Bw_l) \times Ws + & \\ \quad Mnt_l, & \text{if } (Mct_l + Ts_l) > Bw_l \\ Mct_l + Ts_l + Mnt_l - Bw_l, & \text{if } (Mct_l + Mst_l + Mnt_l) > Bw_l \\ 0, & \text{otherwise} \end{cases} \tag{6.34}$$

Where:

$Mxwt_l$	=	The excess weighted traffic through link l.
Mct_l	=	The critical traffic through link l.
$Mct_{(l,q)}$	=	The critical traffic through link l from path q.
Mst_l	=	The significant traffic through link l.
$Mst_{(l,q)}$	=	The significant traffic through link l_n from path q.
Mnt_l	=	The normal traffic through link l.
$Mnt_{(l,q)}$	=	The normal traffic through link l from path q.
Wc	=	The critical traffic weight.
Ws	=	The significant traffic weight.
$T_{(l,q)}$	=	The traffic through link l contributed by path q.

6.6.3 Excess Traffic Objectives

The previous section described methods for calculating the traffic metric for a link. The next step is to determine how the link excess traffic metric can be used to produce objective measures for an evolutionary algorithm. The link traffic metric can be analysed in a number of ways, each of which has different merits.

Link Traffic Landscape
To evaluate the alternatives it is necessary to have an understanding of the *link traffic landscape* that is desirable. If we view the links as points on a 2-dimensional plane surface where the traffic metric can be viewed as the height of each point above plane. When all links have a zero excess traffic, which is the ideal landscape, the surface will be perfectly flat.

A small number of links with significantly higher values than the other links would be represented by a surface with a small number of high peaks. Links with high values would represent bottlenecks which could seriously degrade the whole network, so are undesirable. Another possibility is that the excess traffic is evenly distributed across the network, which would be represented by a raised but relatively level surface.

The landscape has two properties of interest: the first is the total of the excess traffic, which is the total height of each point above the plane; the second is how smooth or rugged the landscape is, i.e. how varied is the height of the points. These properties are independent of each other so require two different objectives for the evolutionary algorithm to minimize: *Total Excess Traffic* and *Excess Traffic Variance*.

Sum of Excess Traffic
The excess traffic of all links in the network are summed giving a measure of the total excess traffic for the mapping. The sum of excess traffic does not distinguish between two mappings that have a similar sum but significantly different landscapes, one with a small number of very high values alongside many small values, the other where there is a larger number of similar values without a single very large value. The sum of excess traffic of all links is a candidate for the total excess traffic metric. The sum of excess traffic of all links

in the network, Jxt_{sum} is given by Equation 6.35.

$$Jxt_{sum} = \sum_{l=1}^{L} Mxwt_l \tag{6.35}$$

Mean Average of Excess Traffic

The excess traffic values of all links in the network are summed and then divided by the number of links in the network. This measure gives the same information as the simple sum since the number of links in a network is constant. The mean average of excess traffic of all links in the network is another candidate for the total excess traffic metric. The mean average of excess traffic of all links in the network, Jxt_{mean} is given as:

$$Jxt_{mean} = \overline{Mxwt} = \frac{1}{L}\sum_{l=1}^{L} Mxwt_l \tag{6.36}$$

This measure of traffic suffers from the same inability as the simple sum measure to distinguish between mappings with similar traffic values but significantly different landscapes.

Maximum Value of Excess Traffic

This objective considers only the highest value of excess traffic of a single link from all the links in the network. Minimising the objective reduces the maximum single value of excess traffic in the network but does not consider the overall level of excess traffic, which is the opposite of using a simple sum measure of excess traffic. This is a candidate for a simple excess traffic variance objective, since a lower maximum single value will also reduce the variance. The maximum value of excess traffic of all links in the network, Jxt_{max} is given by Equation 6.37.

$$Jxt_{max} = \max\{Mxwt_l : l = 1, \ldots, L\} \tag{6.37}$$

Standard Deviation (SD) of Excess Traffic

The standard deviation gives a measure of how much variation from the mean there is in a population. A value of zero means that all values are equal, so in the case of excess traffic a zero value translates to all links having equal levels of excess traffic, however the SD does not give any information about the absolute level of excess traffic across the whole network. The standard deviation of excess traffic of all links in the network is a candidate for the excess traffic variance objective. The standard deviation of excess traffic of all links in the network, Jxt_{sd} is given as:

$$Jxt_{sd} = \sqrt{\frac{1}{L}\sum_{l=1}^{L}(Mxwt_l - \overline{Mxwt})^2} \tag{6.38}$$

On its own, using standard deviation will not help to reduce the overall level of excess traffic as a zero SD can be obtained for any level of excess traffic.

Absolute Mean Deviation (AMD) of Excess Traffic

Absolute Mean Deviation is an alternative to standard deviation that is simpler to calculate because it uses the absolute difference in place of the square of the difference. For a calculation that will be repeated many millions of times the difference in computation time between finding the absolute difference and computing the square of a difference and then later obtaining a square root, is significant. The absolute mean deviation of excess traffic of all links in the network is a candidate for the excess traffic variance objective. The absolute mean deviation of excess traffic of all links in the network, Jxt_{amd} is given as:

$$Jxt_{amd} = \frac{1}{L}\sum_{l=1}^{L} |Mxwt_l - \overline{Mxwt}| \tag{6.39}$$

It can be argued that the AMD is as good as and in some cases a better measure than SD [185]. The same comments made for the suitability of SD can also be made for AMD, however due to the simpler computation AMD would be preferred over SD.

6.7 Determining Paths Through a Lattice

During the development of the metrics and objectives, a formal description of a ComPair was provided. ComPairs are an important and heavily used construct in the calculation of the metrics involving the communication of traffic through the many-core array. An essential piece of information, in at least one of the metric calculations, is the number of paths that exist between the source and target nodes of a ComPair or, more generally, between any two nodes in a square lattice. There is a well known formula for calculating the number of paths between two nodes in a fault-free lattice, however the author has been unable to find any references regarding a calculation in a lattice with faults, i.e. with broken links between nodes. Without the availability of a calculation, an algorithm would have to trace and count each viable path, which is a problem with $O(n!)$ complexity. As an activity that is required for every ComPair, for every mapping for every generation, this is a serious computational problem for any but the smallest of array sizes.

In response to this problem, an algorithm has been developed that can calculate the number of paths between two nodes in a faulty network, using only the position of the nodes and a list of faults in the network.

This section provides the mathematics required to calculate the number of fault-free paths between two nodes in a square lattice.

Using set notation we will show that the number of fault-free paths through a lattice with faulty edges can be represented by:

$$\begin{aligned} P_{ff}(S,T) &= R(S,T) \\ &\qquad - R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_f^n \ \in\ \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k+1 \\ &\qquad + R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_f^n \ \in\ \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k \end{aligned} \tag{6.40}$$

6.7.1 Definitions

This section gives definitions used in the following sections.

Definition 1. Lattice : A lattice consists of nodes arranged in equally spaced rows and columns. In general a lattice can be considered to have an infinite number of rows and columns. For practical purposes lattices can be made finite by specifying a size of R rows and C columns.

Definition 2. Location : A location is a 2-tuple specifying the row and column coordinates, for an infinite lattice $loc = (r, c) \mid r, c \in \mathbb{Z},\ r \geq 0,\ c \geq 0$ and for a finite lattice $loc = (r, c) \mid r, c \in \mathbb{Z},\ 0 \leq r \leq (R-1),\ 0 \leq\ c \leq (C-1)$. Figure 6.10 illustrates the locations in a 5×5 lattice.

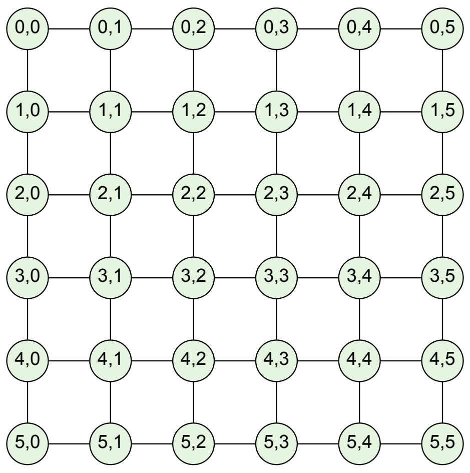


Figure 6.10 – A 6×6 Lattice showing the location coordinates of each node.

Definition 3. Origin : The Origin is location $(0, 0)$, denoted by O.

Definition 4. Subnet : A subnet is any portion of a lattice defined by a source location $s_{loc} = (s_r, s_c)$ and a target location $t_{loc} = (s_r, s_c)$ where $s_r \text{ is not necessarily } < t_r$ and $s_c \text{ is not necessarily} < t_c$.

Definition 5. Node : A node is uniquely identified by it's location $loc(r, c)$; being the row and column coordinates of the node within the lattice. Each node can be connected, via *edges*, to each of its north, south, east and west neighbours (where they exist) as in figure 6.10.

Definition 6. Edge : An edge connects two nodes and is identified by the locations of the nodes that it connects. An edge has a *source node*, the node closest the origin of a lattice or source node of the subnet, and a *target node*, the node furthest from the origin of a lattice or closest to the target node of the subnet. An edge e is, therefore, identified by an ordered 2-tuple of node locations, $(s_{loc}, t_{loc}) = ((s_r, s_c), (t_r, t_c))$.

Definition 7. Minimum Length Path : A minimum length path from the Source to the Target of a lattice or subnet is a path where each traversal of an edge, *a step*, increases the distance from the source and decreases the distance to the Target; there are no steps going "backwards" towards the source. All minimum length paths between the Source and Target are the same length consisting of $(R-1)$ horizontal steps and $(C-1)$ vertical steps for a subnet of size $R \times C$.

Definition 8. *Ordered Set* of Edges : An ordered set of edges $\{e_1, \ldots, e_n\} \mid n > 1$ is a set of edges ordered such that:

$$(s_{k_r} \leq s_{k+1_r}) \wedge (s_{k_c} \leq s_{k+1_c}) \wedge (t_{k_r} \leq t_{k+1_r}) \wedge (t_{k_c} \leq t_{k+1_c}),\ \forall k \mid 1 \leq k \leq (n-1)$$

Definition 9. *Path-Coincident Edges* : Given a lattice with a set of edges $\mathcal{E}_l$, take an ordered set of n edges from $\mathcal{E}_l$, such that $1 \leq n \leq E_l$, then if:

$$(t_{k_r} \leq s_{k+1_r}) \wedge (t_{k_c} \leq s_{k+1_c}),\ \forall k \mid 1 \leq k \leq (n-1)$$

at least one minimal length path traverses all of the selected edges. Such a set of edges is described as a set of *path-coincident edges*, denoted by $\mathcal{E}_p^n$, of order n where n is the number of edges in the set. In addition, each individual edge is considered to be a path-coincident set of order 1.

Using $\mathcal{P}(\mathcal{E}_l)$ to represent the power set of $\mathcal{E}_l$ the set $\mathcal{E}_p^n(\mathcal{E}_l)$ is the set of all sets in $\mathcal{P}(\mathcal{E}_l)$ of order n that are path-coincident.

Definition 10. *Edge-Coincident Paths* : The set of paths that traverse every edge in the set of edges $\mathcal{E}_p^n \in \mathcal{E}_p^n(\mathcal{E}_l)$, are described as the set of edge-coincident paths for the set of edges $\mathcal{E}_p^n$ and are denoted by $\mathcal{R}(\mathcal{E}_p^n)$ which is a member of the set of edge-coincident sets of paths denoted by $\mathcal{R}^n(\mathcal{E}_l)$, where n is the number of edges in the set $\mathcal{E}_p^n$, that is $\mathcal{R}(\mathcal{E}_p^n) \in \mathcal{R}^n(\mathcal{E}_l)$ and $\mathcal{R}^n(\mathcal{E}_l) = \{\mathcal{R}(\mathcal{E}_p^n) \mid \mathcal{E}_p^n \in \mathcal{E}_p^n(\mathcal{E}_l)\}$.

The set of edge-coincident paths $\mathcal{R}(\mathcal{E}_p^n)$ is the intersection of the sets of paths that pass through each edge in $\mathcal{E}_p^n$ individually, that is $\mathcal{R}(\mathcal{E}_p^n) = \mathcal{R}(\{e_1\}) \cap \ldots \cap \mathcal{R}(\{e_n\})$.

Definition 11. Number of Minimum Length Paths in a Subnet : The number of minimum length paths in a subnet is denoted by $R(s,t)$ where s is the source location and t is the target location.

Definition 12. Number of Edge-Coincident Paths : The number edge-coincident paths that traverse a set of edges $\mathcal{E}_p^n$ is denoted by $R(\mathcal{E}_p^n)$. If, for clarity, the subnet needs to be identified, then the source and target locations identifying the subnet follow the set of edges e.g. $R(\mathcal{E}_p^n, (S,T))$

Definition 13. Faulty Edge : A faulty edge is an edge that cannot be used by a path, in effect removing the edge from the lattice.

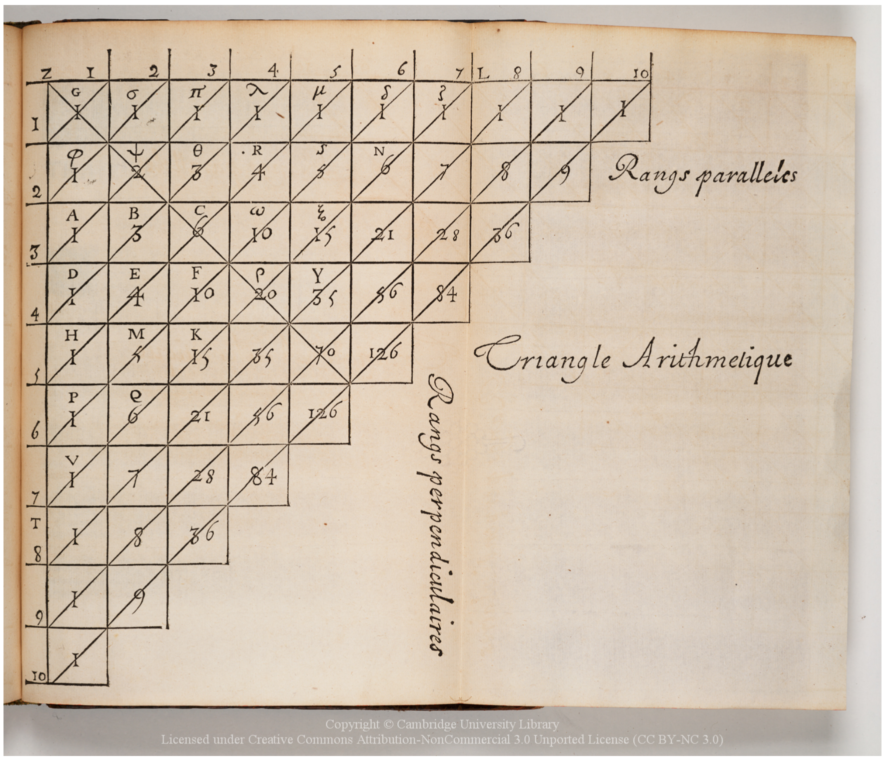
Definition 14. Faulty Node : A faulty node is a node that cannot be visited by a path, in effect removing, from the lattice, the node and all edges leading to and from the node.

Definition 15. Fault-Free : A fault-free lattice is a lattice where all nodes and edges are present and can be used by paths.

6.7.2 Pascal's Triangle

This section begins with a study Pascal's triangle in preparation for showing its relationship to the number of paths in a lattice.

Pascal studied what he referred to as the "Triangle Arithmetique", show in figure Figure 6.11 as Pascal originally presented it in his 1665 treatise [186]. Pascal describes a method of generating the numbers in each cell by placing a number, the "generator", in the top-right hand cell and then describes how the value of every other cell is generated by summing the values of the cells immediately above and to the right of a given cell [187]. The generator can be any value; Pascal used the value of 1 which is also the value relevant for this work.



Z	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	7	8	9	
3	1	3	6	10	15	21	28	36		
4	1	4	10	20	35	56	84			
5	1	5	15	35	70	126				
6	1	6	21	56	126					
7	1	7	28	84						
8	1	8	36							
9	1	9								
10	1									

Rangs paralleles
Triangle Arithmetique
Rangs perpendiculaires

Figure 6.11 – Pascal's "Arithmetical Triangle," as presented in his 1665 Traite du Triangle Arithmetique [186] (Treatise on the Arithmetical Triangle) (Image ©Cambridge University Press).

Pascal notes a number of properties of the numbers in the triangle, one of which is that the numbers correspond to the binomial coefficients.

The coefficient of the $(k+1)^{th}$ term of an expanded binomial to the n^{th} power, where k

takes the values from 0 to n, is given by the binomial coefficient formula:

$$\binom{n}{k} \equiv \frac{n!}{k!(n-k)!} \tag{6.41}$$

This can be related to the value in Pascal's triangle by denoting the rows of Pascal's triangle by r and columns by c, with both r and c both starting at 0, the value in each cell can be calculated by using the binomial coefficient formula by substituting n for $r + c$ and k for r (or equivalently k for c) giving Equation 6.42

$$\binom{r+c}{r} \equiv \frac{(r+c)!}{r!c!} \tag{6.42}$$

As we shall see, Pascal's triangle and the binomial coefficients are relevant when determining the number of paths through a lattice.

6.7.3 The Number of Minimal Length Paths in a Fault Free Lattice

Theorem 1. *The number of minimal length paths between opposite corners of an $r \times c$ lattice, is given by $\binom{r+c}{r}$.*

Proof. Using the principle of mathematical induction:

Given the number of paths from the Origin of a lattice to the node at location $loc(r, c)$ is denoted by $R(O, (r, c))$ then the number of paths can be calculated as:

Base cases: If the source and target nodes are on the same row or column, there will be exactly one path between the source and the target. Since the any node is on the same row and column as itself, we take the number of paths from a node to itself as 1. We therefore have the following base cases:

$$R(O, (0, c)) = 1 \tag{6.43}$$

$$R(O, (r, 0)) = 1 \tag{6.44}$$

$$R(O, O) = 1 \tag{6.45}$$

Induction hypothesis: assume the theorem is true for $r = m$ and $c = n$ i.e. location $loc(m, n)$, so

$$R(m, n) = \frac{(m+n)!}{m!n!} \tag{6.46}$$

Induction step:

$$\begin{aligned}
R(O,(m+1,n+1)) &= R(O,(m,n+1)) + R(O,(m+1,n)) \\
&= \frac{(m+n+1)!}{(m(n+1))!} + \frac{(m+1+n)!}{((m+1)n)!} \\
&= \frac{(m+n+1)!((m+1)!n! + m!(n+1)!)}{(m+1)!n!m!(n+1)!} \\
&= \frac{(m+n+1)!}{(m+1)!(n+1)!}\left(\frac{(m+1)!n! + m!(n+1)!}{!n!m!}\right) \\
&= \frac{(m+n+1)!}{(m+1)!(n+1)!}\left(\frac{(m+1)!}{m!} + \frac{(n+1)!}{!n!}\right) \\
&= \frac{(m+n+1)!}{(m+1)!(n+1)!}(m+n+2) \\
&= \frac{(m+n+2)!}{(m+1)!(n+1)!} \qquad (6.47)
\end{aligned}$$

Hence we have shown that $R(O,O),\ R(0,c)$ and $R(O,(r,0))$ hold and $\forall m \geq 1,\ \forall n \geq 1,\ R(O,(m,n) \Rightarrow R(O,(m+1,n+1))$. Therefore $R(O,(r,c))$ is true $\forall r \geq 1,\ \forall c \geq 1$ by mathematical induction. □

Corollary 1.1. *Given a subnet with a Source node (S) and Target node (T), the number of minimal length paths between the Source and Target that traverse an edge $e = (e_s, e_t)$, denoted by $R(\{e\})$, is the product of the number of minimal length paths, $R(S, e_s)$, between the Source and the edge's source node and the number of minimal length paths, $R(e_t, T)$, between the edge's target node and the Target.*

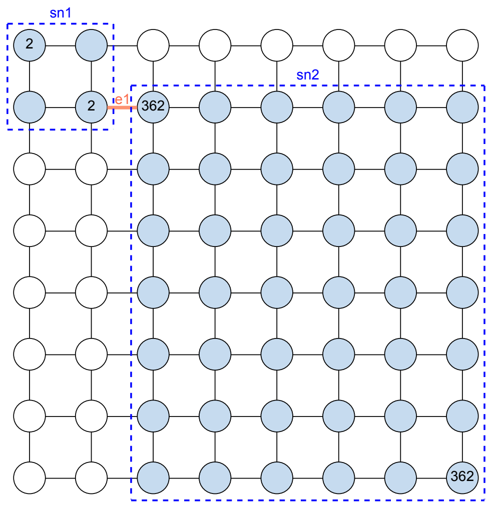
$$R(\{e\}) = R(S, e_s) \cdot R(e_t, T) \qquad (6.48)$$

Figure 6.12a illustrates a subnet of size 8×8 with an edge between $(1,1)$ and $(1,2)$ through which traverses $2 \times 362 = 2708$ paths, giving the number of fault-free paths through the subnet from the source to the target of $3432 - 724 = 2708$.

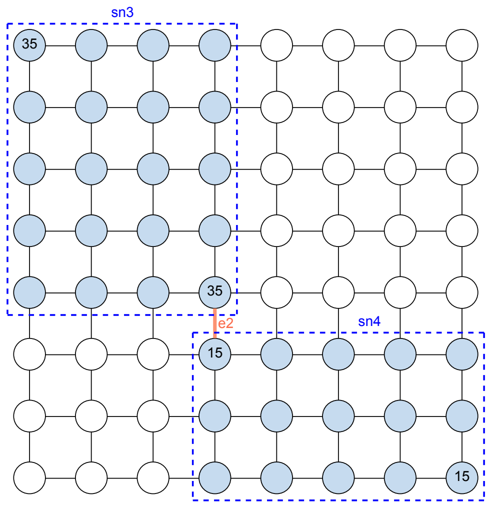
Similarly, Figure 6.12b illustrates a subnet of size 8×8 with an edge between $(4,3)$ and $(5,3)$ through which traverses $35 \times 15 = 525$ paths, giving the number of fault-free paths through the subnet from the source to the target of $3432 - 525 = 2907$.

6.7.4 Number of Paths Traversing Path-Coincident Edges

The equation for calculating the paths through a single edge given in Equation 6.48 can be generalized for multiple path-coincident edges.



(a) The number of paths that traverse edge $e_1 = 2 \times 362 = 724$.



(b) The number of paths that traverse edge $e_2 = 35 \times 15 = 525$.

Figure 6.12 – Calculating the number of paths that traverse an edge.

Figure 6.13 illustrates a network with a pair of edges, e_1 and e_2, that are path-coincident.

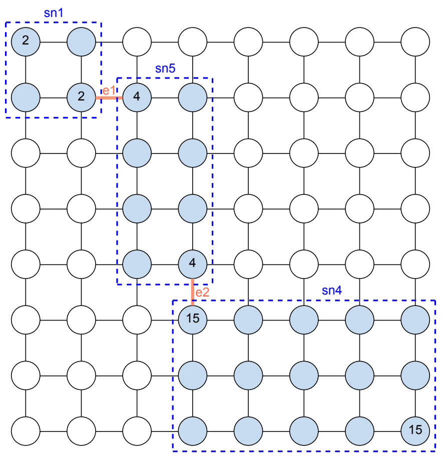


Figure 6.13 – Example of two edge faults.

Theorem 2. *The number of minimal length paths that pass through an ordered pair of path-coincident edges, $\mathcal{E}_p^2 = \{e_1, e_2\}$, is given by:*

$$R(\mathcal{E}_p^2) = R(S, e_{1_s}).R(e_{1_t}, e_{2_s}).R(e_{2_t}, T) \tag{6.49}$$

Proof. From Equation 6.48, the number of paths of the subnet that traverse edge e_1 is given by:

$$R(\{e_1\}) = R(S, e_{1_s}).R(e_{1_t}, T) \tag{6.50}$$

The set of edges $\{e_1, e_2\}$ are ordered and path-coincident, so edge e_2 is in the subnet defined by the locations (e_{1_t}, T) and $R(e_{1_t}, T)$ is the number of minimal length paths in subnet (e_{1_t}, T). Using Equation 6.48, the number of paths of the subnet (e_{1_t}, T) that traverse edge e_2 is given by:

$$R(\{e_2\}, (e_{1_t}, T)) = R(e_{1_t}, e_{2_s}).R(e_{2_t}, T) \tag{6.51}$$

From equations 6.50 and 6.51, the number of paths that traverse both e_1 and e_2 is:

$$R(\{e_1, e_2\}) = R(S, e_{1_s}).R(e_{1_t}, e_{2_s}).R(e_{2_t}, T) \tag{6.52}$$

□

More generally for any ordered set of edges, $\mathcal{E}_f$, taken from the set of edges, $\mathcal{E}_l$, of a subnet of size $R \times C$ there can be multiple sets of path-coincident edges $\mathcal{R}^n \mid 1 \leq n \leq \min(E_f, (R-1)\cdot(C-1))$ where $E_f = |\mathcal{E}_f|$ and n represents the order of the path-coincident set.

Theorem 3. *The number of minimal length paths that pass through a set of path-coincident edges, $\mathcal{E}_f^n = \{e_1, \ldots, e_n\}$, is given by:*

$$R(\mathcal{E}_f^n) = R(S, e_{1_s}).R(e_{1_t}, e_{2_s}). \ldots .R(e_{(n-1)_t}, e_{n_s}).R(e_{n_t}, T) \tag{6.53}$$

Proof. Using the principle of mathematical induction:

Base case: From Equation 6.48, the number of paths that traverse edge e_1 is given by:

$$R(\{e_1\}) = R(S, e_{1_s}).R((e_{1_t}, T)) \tag{6.54}$$

Induction hypothesis: assume the theorem is true for n = k so:

$$R(\{e_1, \ldots, e_k\}, (S, T)) = R(S, e_{1_s})). \ldots .R(e_{(k-1)_t}, e_{k_s}).R(e_{k_t}, T) \tag{6.55}$$

Induction step: The set of edges $\{e_1, \ldots, e_n\}$ are ordered and path-coincident, so edges $\{e_{(k+1)}, \ldots, e_n\}$ are in the subnet defined by the locations (e_{k_t}, T). From Equation 6.48, the number of paths of the subnet (e_{k_t}, T) that traverse edge $e_{(k+1)}$ is given by:

$$R(\{e_{(k+1)}\}, (e_{k_t}, T)) = R(e_{k_t}, e_{(k+1)_s}).R(e_{(k+1)_t}, T) \tag{6.56}$$

From equations 6.55 and 6.56 we have:

$$R(\{e_1, \ldots, e_{(k+1)}\}, (S, T)) = R(S, e_{1_s})). \ldots .R(e_{k_t}, e_{(k+1)_s}).R(e_{(k+1)_t}, T) \tag{6.57}$$

Hence we have shown that $R(\{e_1\})$ holds and $\forall k \geq 1,\ R(\{e_k\}) \Rightarrow R(\{e_{(k+1)}\})$. Therefore $R(\{e_1,\ \ldots\ ,e_n\})$ is true $\forall n > 1$ by mathematical induction. □

6.7.5 Number of Paths Through Multiple Faulty Edges

Theorem 4. *Given a lattice with $\mathcal{E}_l$, edges and a set of m faulty edges $\mathcal{E}_f = \{e_1,\ \ldots\ ,e_m\} \mid \mathcal{E}_f \subseteq \mathcal{E}_l$, then the number of paths that traverse at least one faulty edge is given by:*

$$\begin{aligned} P_{fx}(S,T) &= +\ R(\mathcal{E}_p^n),\ \forall\ \mathcal{E}_f^n\ \in\ \mathcal{R}^n(\mathcal{E}_f) \mid \exists\ k \in \mathbb{Z} : n = 2k+1 \\ &\qquad\qquad -\ R(\mathcal{E}_f^n),\ \forall\ \mathcal{E}_p^n\ \in\ \mathcal{R}^n(\mathcal{E}_f) \mid \exists\ k \in \mathbb{Z} : n = 2k \end{aligned} \tag{6.58}$$

Proof. If none of the faulty edges are path-coincident then all of the sets of paths that pass through each faulty edge, represented by $R^1(\mathcal{E}_f)$, are disjoint. Then the number of paths that traverse a fault edge, denoted by $P_{fx}(S,T)$ is:

$$P_{fx}(S,T) = R^1(\mathcal{E}_f) \tag{6.59}$$

If the set of faulty edges includes pairs of edges that are path-coincident, then the number of paths that traverse both edges in a pair of path-coincident edges are also included in the sets of paths that traverse each edge individually. Each set of paths in $R^2(\mathcal{E}_f)$ are included twice in $R^1(\mathcal{E}_f)$, once for each edge that is path-coincident. Therefore to correctly calculate the number of paths that traverse a set of fault edges when there is at least one pair of path-coincident edges (and no triplets of path-coincident edges), $R^2(\mathcal{E}_f)$ must be subtracted from the total:

$$P_{fx}(S,T) = R^1(\mathcal{E}_f) - R^2(\mathcal{E}_f) \tag{6.60}$$

If the set of faulty edges includes triplets of edges that are path-coincident, then the number of paths that traverse all three edges in a path-coincident set of edges are also included in the sets of paths that traverse each pair of the three edges. Each set of paths in $R^3(\mathcal{E}_f)$ are included three times in $R^2(\mathcal{E}_f)$, once for each pair of edges in the path-coincident triplet. Therefore to correctly calculate the number of paths that traverse a set of fault edges when there is at least one triplet of path-coincident edges (and no sets of four edges that are path-coincident), $R^3(\mathcal{E}_f)$ must be added to the total:

$$P_{fx}(S,T) = R^1(\mathcal{E}_f) - R^2(\mathcal{E}_f) + R^3(\mathcal{E}_f) \tag{6.61}$$

In set theory this requirement for subtracting and adding alternate sets of higher order intersections of sets to correctly calculate the number of elements in the union of intersecting sets is referred to as the *principle of inclusion and exclusion* [188].

Therefore, for a set of faulty edges with arbitrary sized sets of path-coincident edges the calculation for the number of paths that traverse at least one faulty edge is:

$$\begin{aligned} P_{fx}(S,T) &= + R(\mathcal{E}_p^n),\ \forall\, \mathcal{E}_f^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k+1 \\ &\quad - R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_p^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k \end{aligned} \tag{6.62}$$

□

6.7.6 Number of Paths Through a Lattice with Multiple Faulty Edges

Theorem 5. *: Given a lattice with $\mathcal{E}_l$ edges and a set of m faulty edges $\{e_1, \dots, e_m\} = \mathcal{E}_f \subseteq \mathcal{E}_l$, then the number of fault-free paths from the source to the target of the lattice is given by:*

$$\begin{aligned} P_{ff}(S,T) &= P(S,T) \\ &\quad - R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_p^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k+1 \\ &\quad + R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_p^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k \end{aligned} \tag{6.63}$$

Proof. Given a lattice with $\mathcal{E}_l$, edges and a set of m faulty edges $\mathcal{E}_f = \{e_1, \dots, e_m\} \mid \mathcal{E}_f \subseteq \mathcal{E}_l$, then the number of fault-free paths in the lattice is the number of paths that traverse at least one faulty edge subtracted from the number of paths in the equivalent fault-free lattice.

The number of paths in a fault-free lattice is:

$$P(S,T) = R^1(\mathcal{E}_f) - R^2(\mathcal{E}_f) + R^3(\mathcal{E}_f) \tag{6.64}$$

The number of paths that traverse at least one faulty path is given by Equation 6.58.

The number of fault-free paths in a lattice is therefore:

$$\begin{aligned} P_{ff}(S,T) &= P(S,T) \\ &\quad - R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_p^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k+1 \\ &\quad + R(\mathcal{E}_f^n),\ \forall\, \mathcal{E}_p^n \in \mathcal{R}^n(\mathcal{E}_f) \mid \exists\, k \in \mathbb{Z} : n = 2k \end{aligned} \tag{6.65}$$

□

6.7.7 Summary

This section presented a calculation for the number fo paths through a faulty network using only the location of the nodes and a list of the faulty edges in the network.

This calculation has been implemented in an algorithm has been designed to work with directional edges and with the source and target nodes of a ComPair in any possible orientation. Although the algorithm has been designed for a two dimensional lattice, the principles are valid for a regular lattice of any dimension. In addition it is possible to employ the algorithm to calculate paths through non-regular lattices by first converting the non-regular lattice to a regular lattice with faults and then using the algorithm to calculate the number of paths in the faulty lattice.

6.8 Multi-Objective Correlation

The experiments in this section have been designed to explore the level of correlation that there exists between the evolutionary objectives described in this and previous chapters.

6.8.1 Correlation

If two objectives are highly correlated, then, when they are plotted on a graph, the plotted points will be aligned along an axis such as the $x = y$ axis and will present only a few closely positioned points as a Pareto front. A pair of highly correlated objectives can be replaced by just one of the objectives - the second objective does not add any useful information.

If two objectives are poorly correlated, then their plotted points will form a cloud and the Pareto front will consist of many points on a curve with a shape of, for example, $y = ax^{-1}$ or $y = ax^{-2}$. Such a Pareto front provides a range of points that trade-off one objective against the other from which one can be selected on the basis of additional criteria. For example if the objectives of core fault tolerance and power consumption are poorly correlated then a Pareto front will have a range of points: at one end of the Pareto front there will be a point that has good core fault tolerance and poor power consumption, while at the other end of the Pareto front there will be a point with poor core fault tolerance and good power consumption. In between these two points there will be numerous points that have different degrees of trade-off between the two objectives. Such a Pareto front allows a particular point to be chosen to promote a particular property.

The selection criteria do not have to remain fixed so different points of the Pareto front can be selected at different times in response to changing demands and environmental conditions.

In multi-dimensional space, axes are described as orthogonal, which is the multi-dimension equivalent of perpendicular axes in two and three dimensional space. Each each pair of axes $\{x, y\}, \{x, z\}$ and $\{y, z\}$ in three dimensional space are orthogonal to each other and are said to form an orthogonal set. We can use the same terminology to describe a pair of objectives as orthogonal if they are not highly correlated.

For multi-dimensional problems we want to select a set of objectives that are all pair-wise orthogonal and as a whole form an orthogonal set, so that each objective is adding a useful influence to the selection of solutions from the solution space. Exploring the correlation of pairs of objectives in a 2-dimensional multi-objective problem is an efficient way to ensure that an effective selection of objectives can be made for higher dimensional multi-objective problems.

6.8.2 Objectives

The objectives explored in this set of experiments are:

- Core fault tolerance
- Link fault tolerance
- Network Power
- Excess traffic (mean traffic)
- Excess traffic (traffic AMD)

Experiments are performed for each pair combination of the objectives.

For excess traffic, both the mean value and the deviation measure of AMD are included to determine, experimentally, what degree of correlation there is between the mean and AMD measures.

6.8.3 Evolutionary Algorithm Parameters

The experiments are run using the evolutionary algorithm parameters selected in Section 5.5 and listed in Table 6.8. The hardware is fault free throughout these experiments.

6.8.4 Graph Selection

The experiments are repeated for each of the three densities of graphs listed in Table 6.9 The graph names are based on the number of processing nodes, source nodes and sink nodes, so the first graph with 28 processing (p) nodes,1 source (s) node and 2 sinks (k) nodes is referred to as p28s1k2.

The graphs all have 28 processing nodes which will be mapped onto a 6×6 array within a 8×8 hardware map with a border of $B((b, 1), (b, 1), (b, 1), (b, 1))$. Graphs with 28 processing nodes were chosen because, based on the experiment results in Chapter 4

Table 6.8 – Multi-Objective Fault Free Experiment Parameters

Parameter	Value
Hardware Map Size	8×8
Hardware Map border	$B((b,1),(b,1),(b,1),(b,1))$
Many-Core Array Size	6×6
Number of generations	1000
Population size	100
Elite	10 fittest individuals cloned
Parents	20 fittest individuals used as parents
Descendants	4 from each parent via permutation
Novel	10 randomly generated individuals
Mutation Pattern	g15

Table 6.9 – Test Application Process Graphs

Parameter	Value
Graph p28s2k1	28 Process Node
	2 Source Nodes
	1 Sink Nodes
	Sparsely Connected
	Figure 6.1
Graph p28s3k2	28 Process Node
	3 Source Nodes
	2 Sink Nodes
	Moderately Connected
	Figure 6.2
Graph p28s1k3	28 Process Node
	1 Source Nodes
	3 Sink Nodes
	Densely Connected
	Figure 6.3

Core Fault Tolerance presented in Figure 4.15, it was known that the minimum core fault tolerance value was non-zero and that mappings are reasonably difficult to find.

For each pair of experiments the sources and sink processing nodes are randomly allocated to the source and sink nodes of the environment. No attempt has been made to force all the experiments to place the sources and sinks at the same locations. The actual position of the source and sink processes will have an effect on the values of the objectives, but is not expected to have an effect on the correlation of a pair of objectives.

The results of experiments for the densely connected graph p28s1k3 are presented in this chapter. The results for the sparsely and moderately connected graphs are presented in *Multi-Objective Plots for Sparsely and Moderately Connect Graphs* as the results are recognizably similar to those of the densely connected graph and do not merit repeating the detailed analysis.

6.8.5 Data Presentation

In each experiment 100,000 individuals are evaluated and plotted using graded colours representing the generation when the individual first appeared. The colour bar relating the generation of creation to a colour is shown in Figure 6.14. The points of the last generation are plotted first, and the first generation plotted last so that points in the early generations are not masked by the points of later generations.

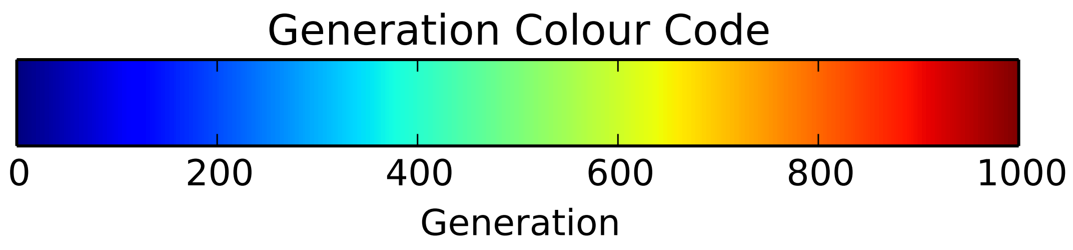


Figure 6.14 – Generations Colour Bar

Plots of Individuals

When interpreting the plots, the absolute value of the objectives is not important. Of more interest is the how the objectives relate to each other. Therefore, the range of values, and scale of axes have been chosen for each individual plot to achieve the best presentation of the data to illustrate the relationship between the two objectives.

Some objectives have a very large range which can have the effect of compressing the interesting data into a very small area close to an axis. All results start with a plot of all individuals, to give a perspective of the whole data set. Where the interesting data is compressed into a small portion of the dataset close to one of the axes, additional plots have been added which focus on the interesting data points.

Pareto Front Plots

The Pareto front plots show the Pareto front for generations 1, 10, 100 and 1000 which prove to be appropriate generations to illustrate the improvement of the Pareto front over the 1000 generations of the experiments.

The scale of the axes has been chosen to illustrate the improvement of the Pareto front from generation 1 to generation 1000, and are not the scales used to plot the individuals. In some cases, for example in Figure 6.15b, this results in the Pareto front appearing flattened, which may give the false impression that the values for one of the objectives are not very different, and therefore are not materially different. It is important to realise that it is not the presentation of the Pareto front, but the number of points on the Pareto front that is significant, as a larger number of points gives more flexibility in choosing a suitable solution based on other criteria.

6.8.6 Results for a Densely Connected Graph

This section presents the results of the experiments using the densely connected graph p28s1k3.

Core Fault Tolerance with Network Power - Figure 6.15
Figure 6.15 shows the results for the multi-objective combination of Core Fault Tolerance and Network Power.

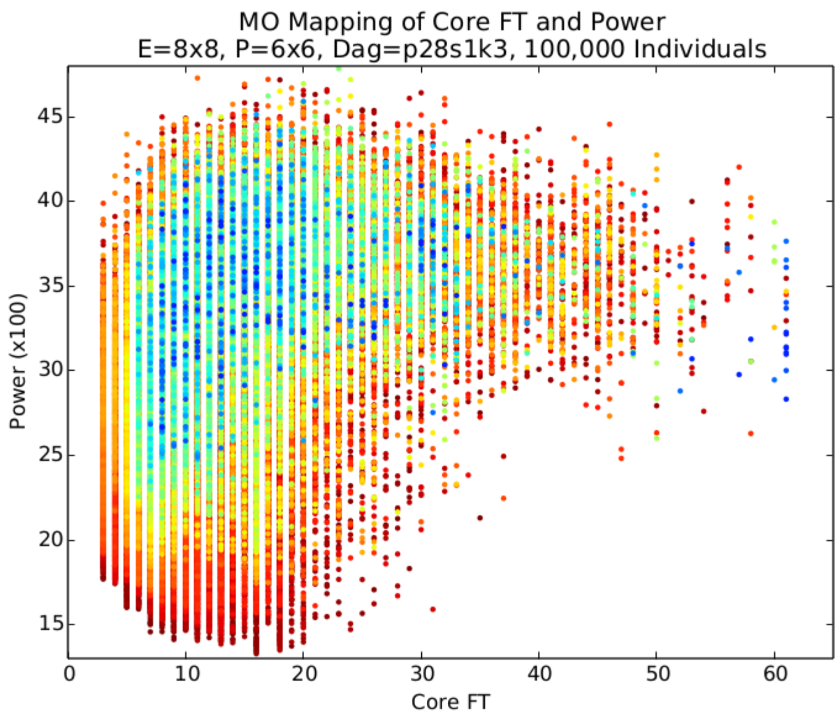
Figure 6.15a exhibits two features that can be seen in all the experiments where one of the objectives is core fault tolerance, which is the discrete nature of the core fault tolerance values with a minimum value of 3. This is explained by noting that the values for core fault tolerance have a relatively low range and that a graph with 28 processing nodes being mapped on to a 6×6 array has a minimum possible core fault tolerance value of 3. Both of these features were observed and discussed in Chapter 4.

Since the evolutionary algorithm is finding the minimum value for core fault tolerance, it can be deduced that there is a sufficiently large number of optimum solutions for core fault tolerance that they are reasonably easy to find.

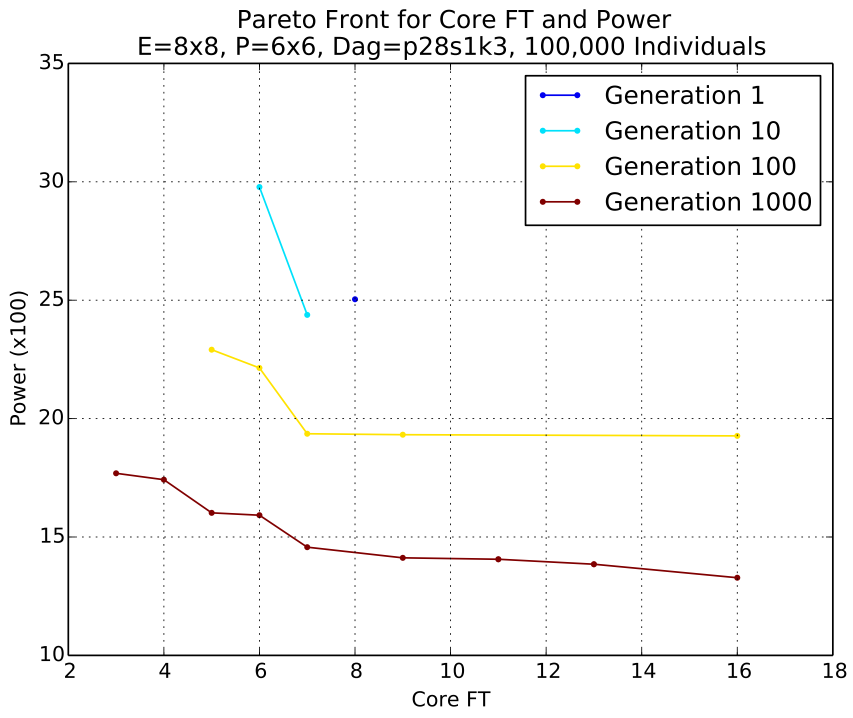
The Pareto front for generation 1000 has nine well-spaced points. The colour of the plotted points show that the evolutionary algorithm continued to improve on the network power objective after solutions with optimal core fault tolerance were found. This indicates that optimal solutions for core fault tolerance are ’easier’ to find than those for network power, which is consistent with the single objective experiments that explored the evolutionary algorithm parameters in Section 5.5, and is due to there being many possible mappings which have an optimal arrangement of processing cores for core fault tolerance.

Core Fault Tolerance with Link Fault Tolerance - Figure 6.16
The core fault tolerance with link fault tolerance plot has a similar profile to the core-power. A scale of $\times 100$ has been used to enable the plot to show all individuals. The use of the scale has the effect of flattening the Pareto front, which is revealed by a plot that focuses on the lower half of the the link scale as in Figure 6.16b.

The plot of individuals suggests that the two objectives are not correlated because it is not possible to predict the value of one objective from the value of the other objective however, the small number of points on the Pareto front reveals that there are solutions that have good values for both objectives simultaneously. The Pareto front has a sharp corner, indicating that there are some solutions that simultaneously have good core fault tolerance and good link fault tolerance. The sharp corner also means that a small improvement in one objective is balanced with a relatively large deterioration in the other objective. It



(a) Individuals

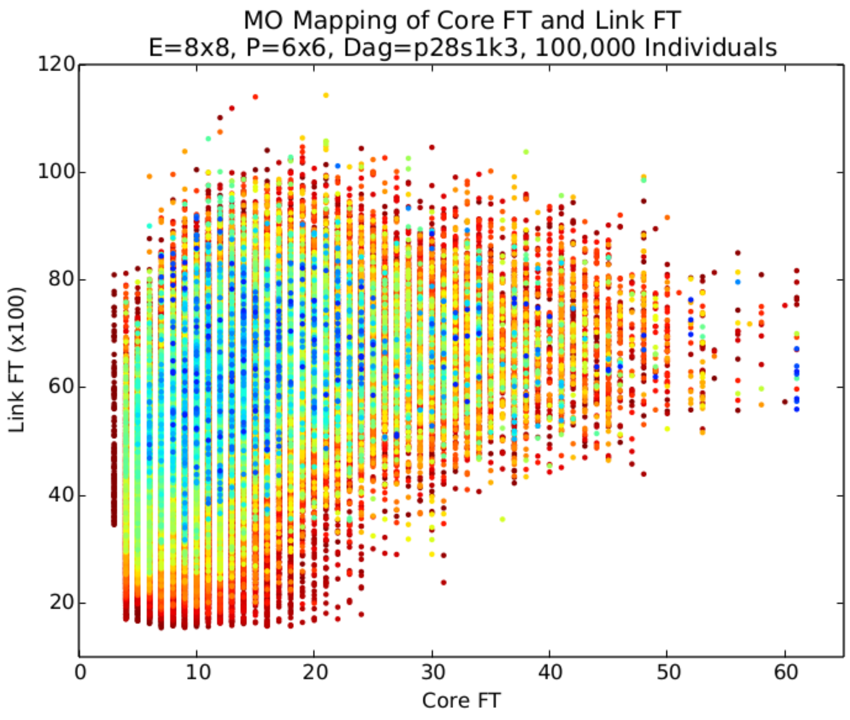


(b) Pareto Front Evolution

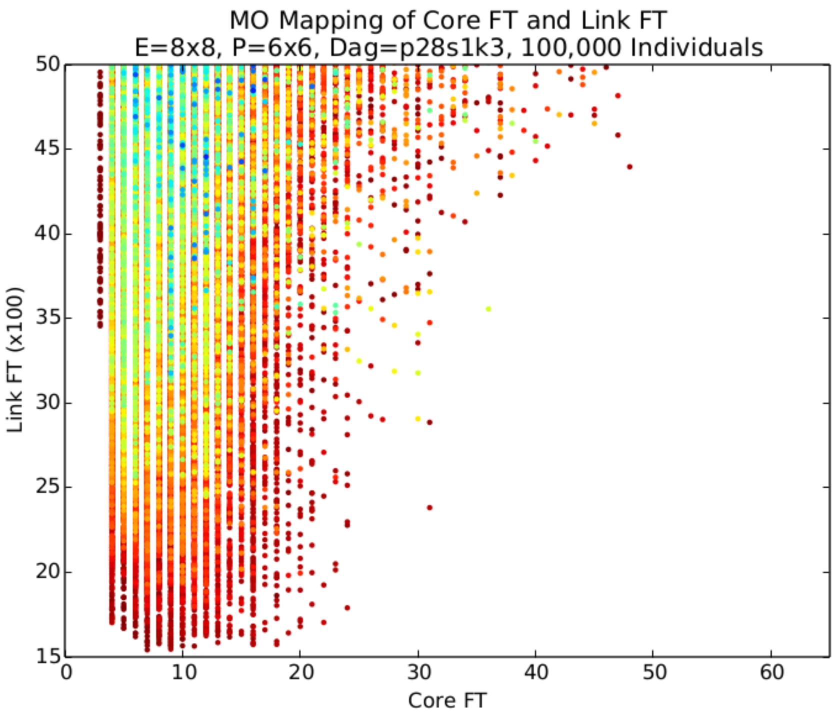
Figure 6.15 – Core FT with Network Power

is important to remember, when reading these plots, that the scale for the plot has been chosen to illustrate the improvement of the Pareto front over 1000 generations and not to illustrate the detail of the final Pareto front.

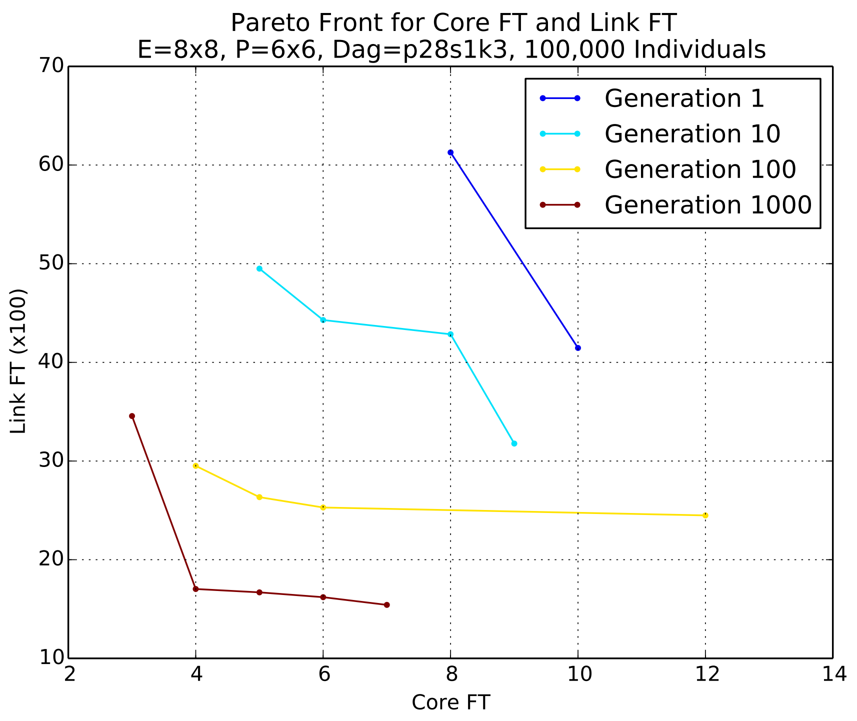
To understand these results requires an examination of the effect of the metric calculations on the placement of process core and idle cores. The core fault tolerance calculation acts to evenly distribute cores across the array, while the relative position between processing cores has no effect on the metric. Therefore, there can be many alternative arrangements



(a) Link FT 0 - 12,000



(b) Link FT 1,500 - 5,000



(c) Pareto Front Evolution

Figure 6.16 – Core FT with Link FT

of processing cores, all of which have the same core fault tolerance fitness. The link fault tolerance objective gives an evolutionary pressure to avoid critical links, which results in the placement of the source and target nodes of ComPairs in different rows and columns. Evenly distributed idle cores are not detrimental to the link fault tolerance and may actually be helpful.

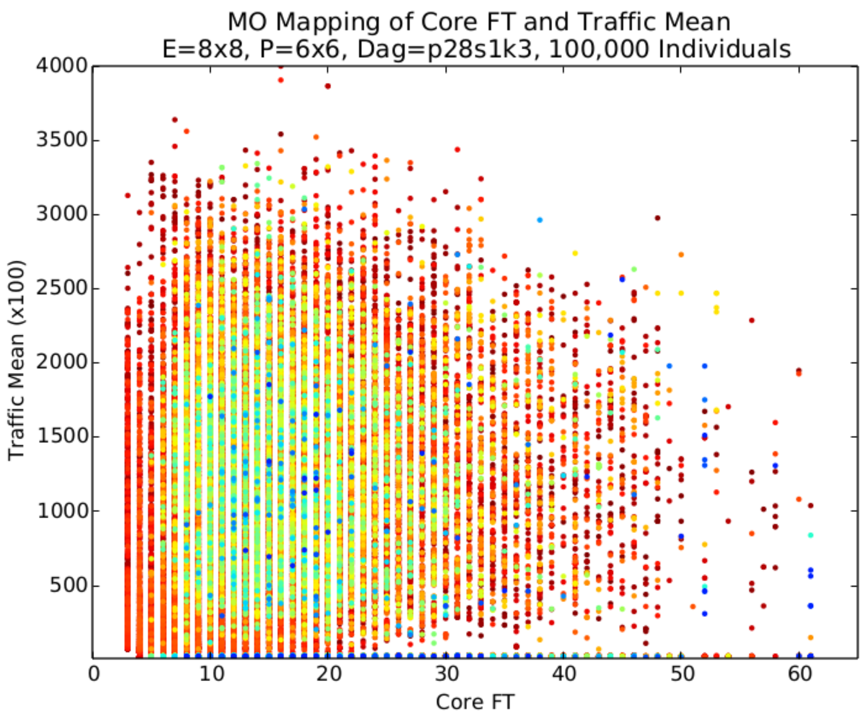
Although the sharp corner shows that there are some solutions that simultaneously have good core fault tolerance and good link fault tolerance, the spread of solutions does not imply that core fault tolerance and link fault tolerance are high correlated objectives.

This analysis suggests the possibility of alternative mechanisms for finding optimal solutions for this pair of objectives by, for example, using a single objective evolutionary algorithm on core fault tolerance to find an optimum arrangements of idle cores, and then fixing the idle cores and using a single objective evolutionary algorithm on link fault tolerance to find an optimal arrangement of the processing cores around the idle cores. This strategy may reduce the processing time required for the particular objective pairing.

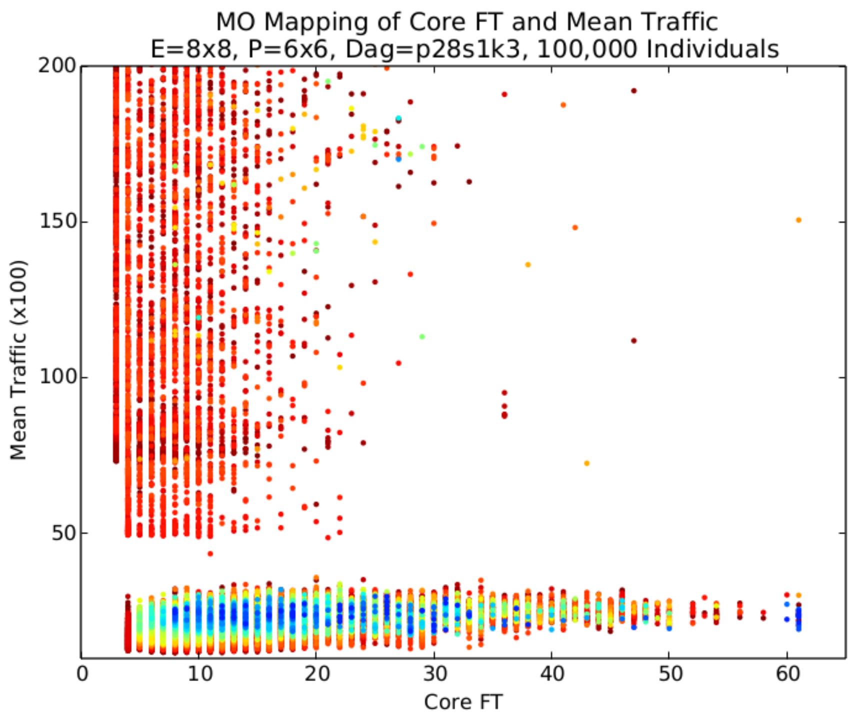
Core Fault Tolerance with Mean Traffic - Figure 6.17

Core Fault Tolerance with mean traffic is similar to the core-link plot in respect of the shape and number of points in the final Pareto front, while the plots of the individuals show the there is not a strong correlation between the two objectives.

The scale of the plot of all individuals in Figure 6.17a is so large that the detail of the

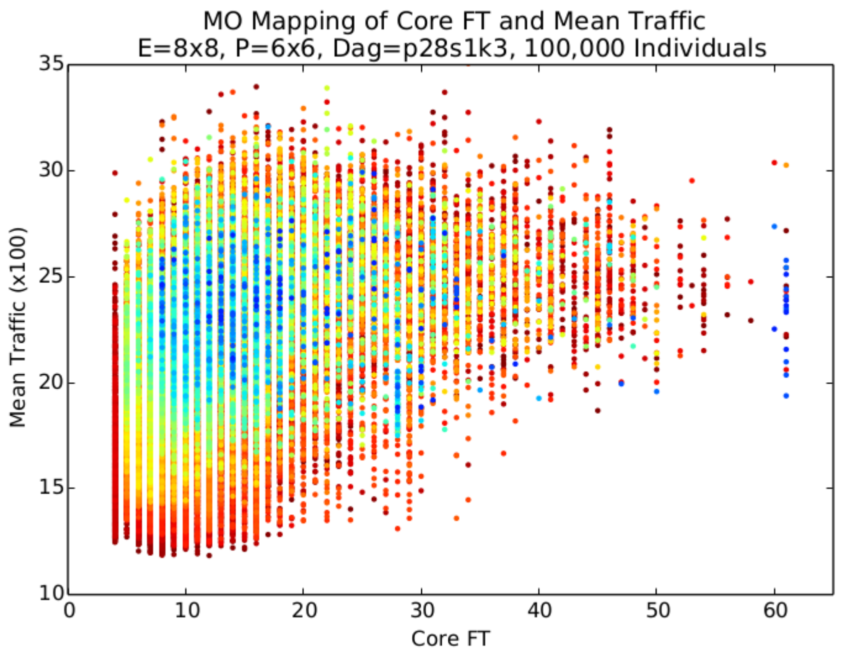


(a) Mean range 0 - 400,000

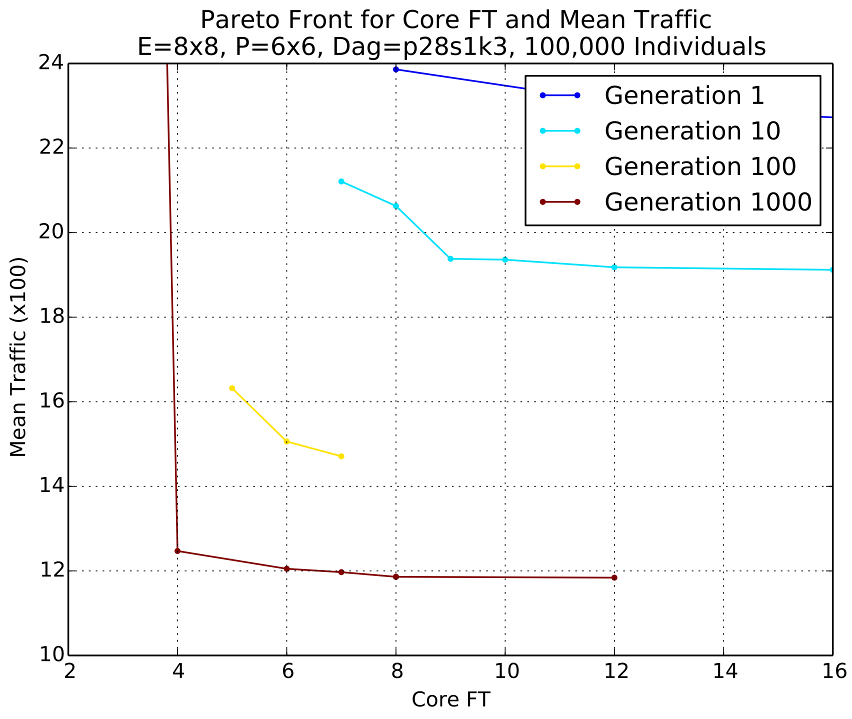


(b) Mean range 0 - 20,000

Pareto front is compressed into a very small region above the core FT axis. Two more plots are provided with smaller scales to reveal the detail of the dataset close to the core FT axis. Figure 6.17b shows that there is a "break" in the data between traffic objective value between 3.500 and 5,000, while Figure 6.17c shows a Pareto front corner similar,



(c) Mean range 1,000 - 3,500



(d) Pareto Front Evolution

Figure 6.17 – Core FT with Mean Traffic

but sharper, than that for the combination of core Fault Tolerance with link Fault Tolerance.

Looking at the details of the metric calculation for excess traffic, the effect the metric will have on the placement of the nodes of a ComPair are that it will attempt to make the distance between the nodes as small of possible to reduce overall traffic, but will also attempt to avoid overloading individual links by avoiding arrangements where the nodes are on the same row and column. The second of these is similar to the effect of link fault tolerance discussed for core fault tolerance with link fault tolerance.

The conclusion is that, like link fault tolerance, evenly distributed idle cores have no detrimental affect on the mean traffic objective value and suggests the single objective approach may also work for this objective pairing.

The area above this plot which is devoid of an data points is an interesting feature which requires explanation. The excess traffic calculation uses a weighted approach that penalises critical links more than significant links which are penalised more than normal links. The weight approach is designed to discriminate against mappings with critical links. The weighting of critical links is the cause of the sudden increase in the traffic objective value. The mapping below the break will be free of critical links while the solutions above the gap will have at least one critical link. This is shows that the weighting of links based on criticality does discriminate against critical links and vindicates their use.

Core Fault Tolerance with Traffic AMD - Figure 6.18
The plots for core Fault Tolerance with traffic AMD are very similar those of the core-mean plots in Figure 6.17. This indicates that the mean traffic and traffic AMD objectives are similar due to them being based on the same underlying metric of excess traffic. The relationship between mean traffic and traffic AMD is explored later in this section.

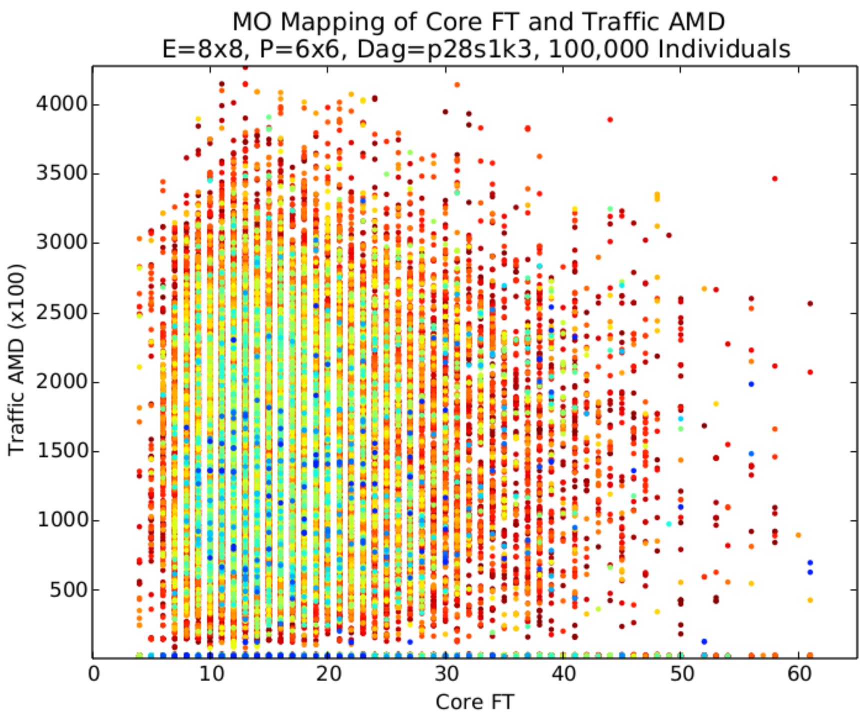
The analysis of the core FT with mean traffic is also relevant to this experiment.

Link Fault Tolerance with Traffic AMD - Figure 6.19
The dataset for link Fault Tolerance with traffic AMD has some interesting detail which only becomes evident with a series of fours plots with progressively focusing on smaller subsets of the whole dataset.

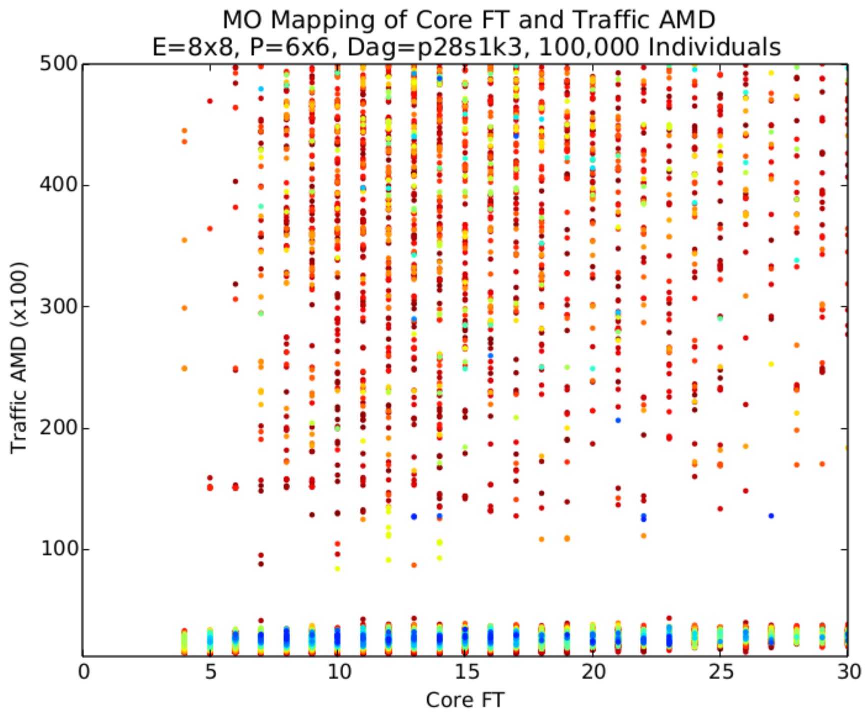
The traffic AMD objective shows a number of distinct "strata" of data points. Deviation measures such as AMD and SD tell us about the variation between the values of the data set. The stratification of the traffic AMD values tells us that the amount of variation of traffic through links is increasing, i.e the difference between those will the lowest amount of traffic and those with the greatest amount of traffic. The effect of the weighting of critical links will, when the number of critical links in a mapping increments, suddenly increase the metric value for some links while leaving others unchanged, so increasing the variation. This analysis suggests that each stratum will be related to a specific number of critical links.

Why the core-traffic AMD dataset only exhibits two levels of data instead of multiple levels, is not obvious and would merit additional investigation.

The Pareto front has many more points arranged in a more identifiable curve and does not exhibit the sharp corner of solutions of the previous experiment groups, which could be described as a good quality. The shape and number of points on the Pareto front offer a large selection of degrees of trade off between the two objectives.

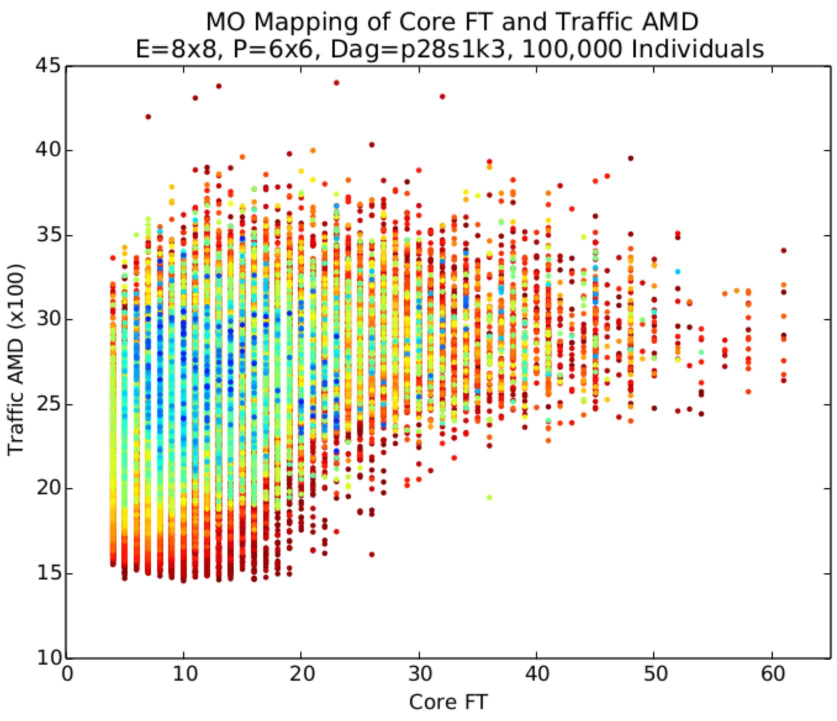


(a) AMD range 0 - 400,000

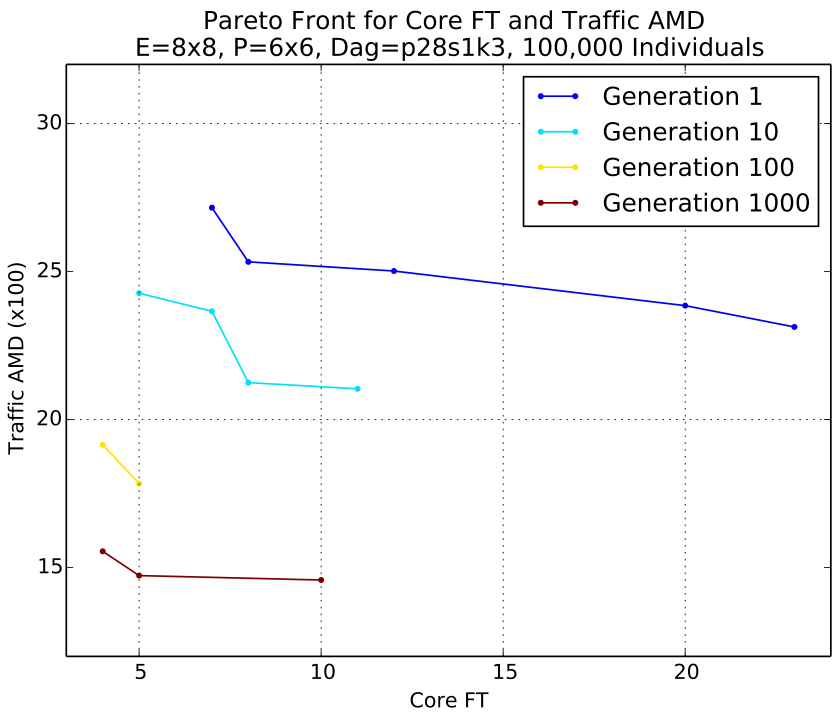


(b) AMD range 0 - 50,000

The quality of the Pareto front indicates that the underlying metric calculations of the objectives are competing against each other in a manner that has not been present in previous objective pairings. A particular example of this is that to achieve the smallest traffic volume ComPairs node pairs need to be adjacent to each other, however this it not



(c) AMD range 1,000 - 4,500

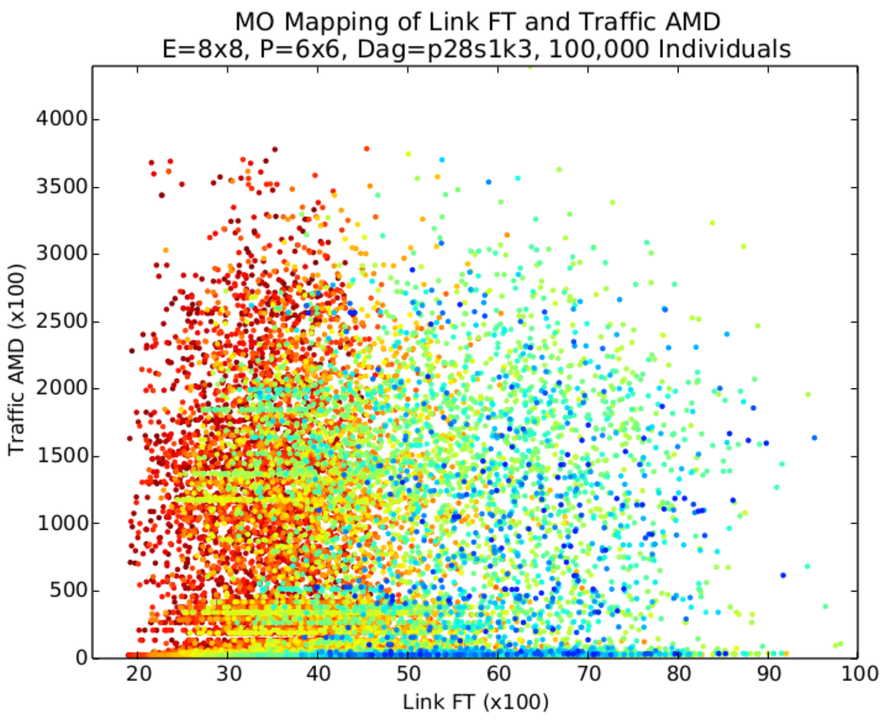


(d) Pareto Front Evolution

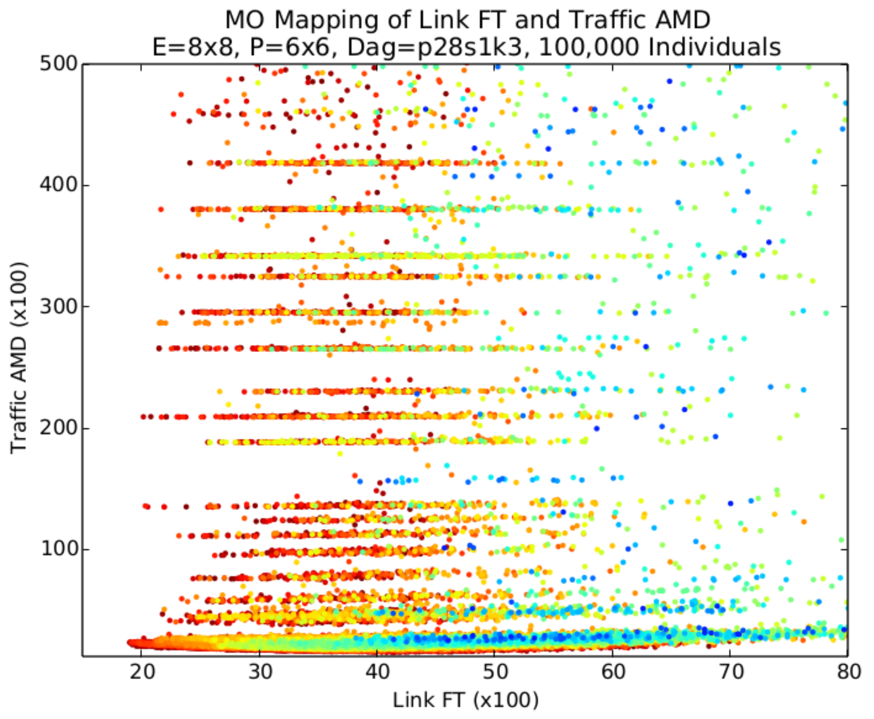
Figure 6.18 – Core FT with Traffic AMD

good for link FT because it would result in a large number of critical links.

The pairing of link Fault Tolerance with traffic AMD is a good example of a pairing that



(a) AMD range 0 - 400,000

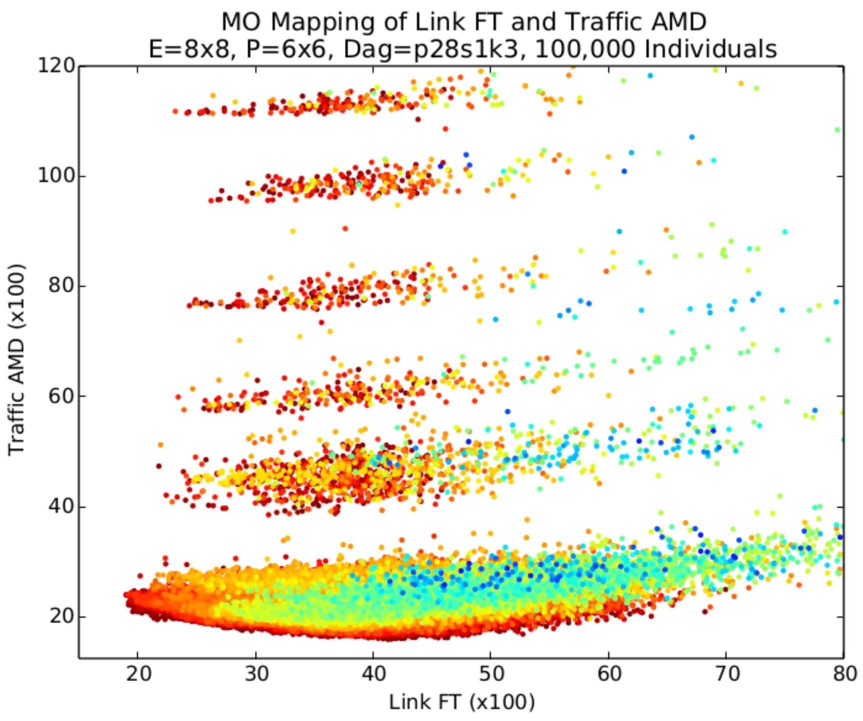


(b) AMD range 0 - 50,000

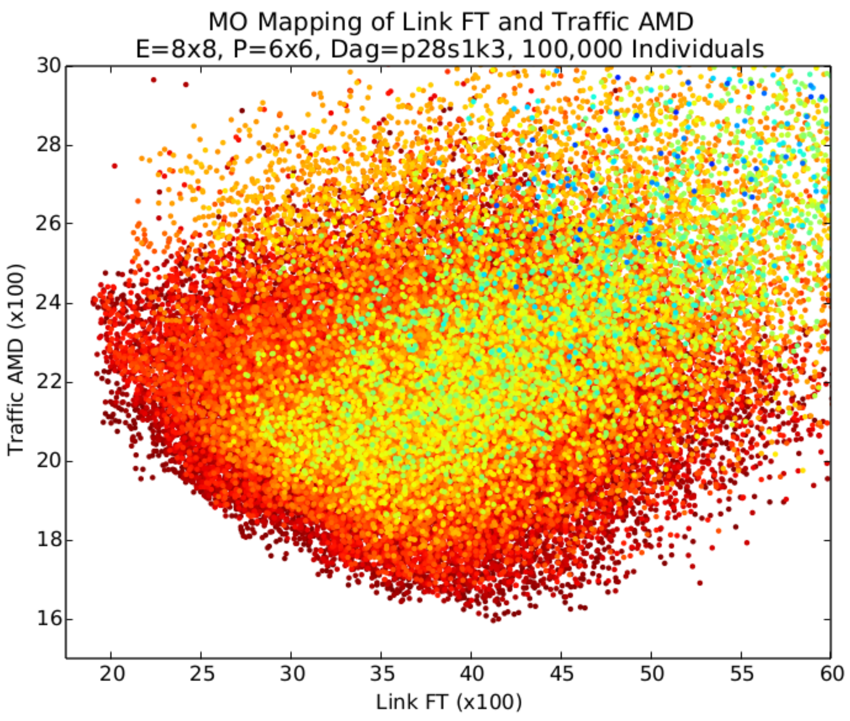
works well as a multi-dimensional problem.

Link Fault Tolerance and Mean Traffic - Figure 6.20

The plots for link Fault Tolerance and mean traffic have a varied Pareto front, similar to the link-AMD Pareto front. The data plots exhibit two strata instead of the multiple strata

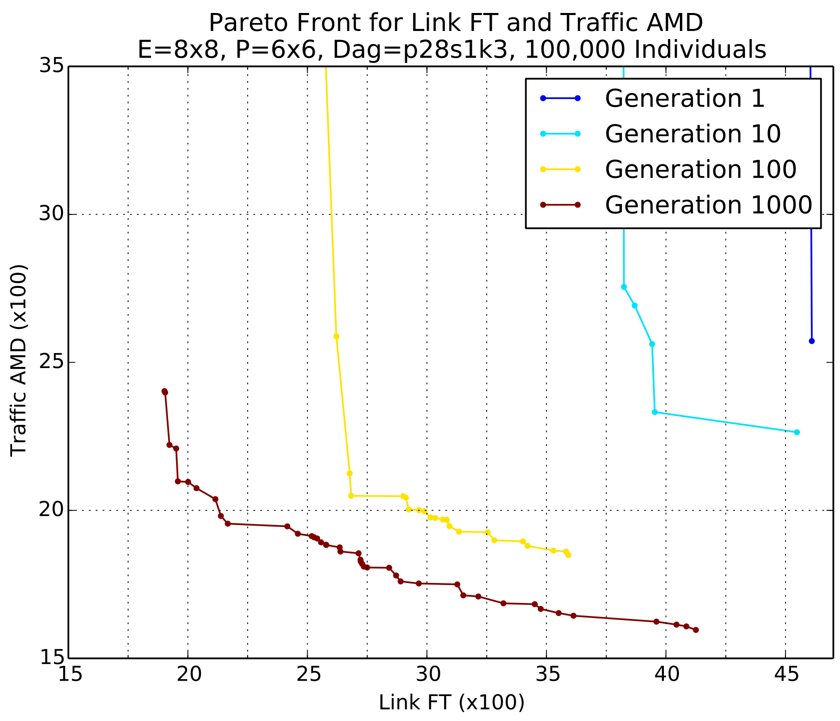


(c) AMD range 1,000 - 12,000



(d) AMD range 1,500 - 3,000

of the link FT and traffic AMD plots. This highlights the qualitative difference the mean of traffic and the traffic AMD: if there are two mappings that differ by a single critical link, as discussed eariler, the AMD measure will jump significantly, however the mean will rise by a smaller amount because the mean calculation averages the rise of value on one link



(e) Pareto Front Evolution

Figure 6.19 – Link FT with Traffic AMD

across all links in the network.

Link FT and Network Power - Figure 6.21

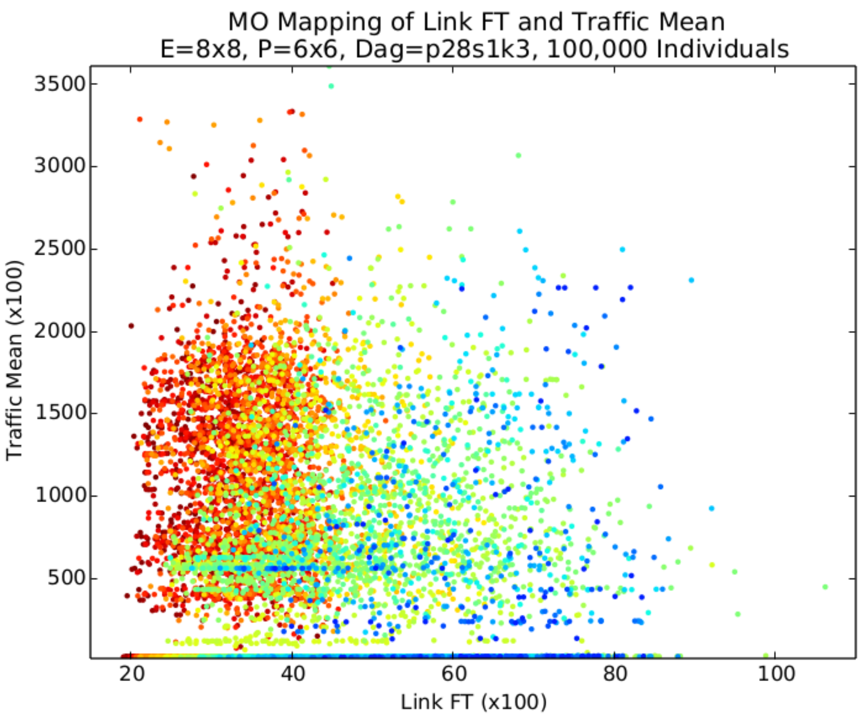
The link FT and network power pairing produces a good quality Pareto front with many points forming a smooth curve, providing many options for trade off between the two objectives. The shape of the plot of individuals and the Pareto front suggest that the two objectives are not correlated and may be orthogonal.

As previously discussed, the link FT objective is lowest when all ComPair node pairs are on different rows and columns. The network power objective is at a minimum when all ComPair node pairs are adjacent. The results of the experiment of this pairing shows that these two objectives work against each other and so form a good pairing in a multi-objective scenario.

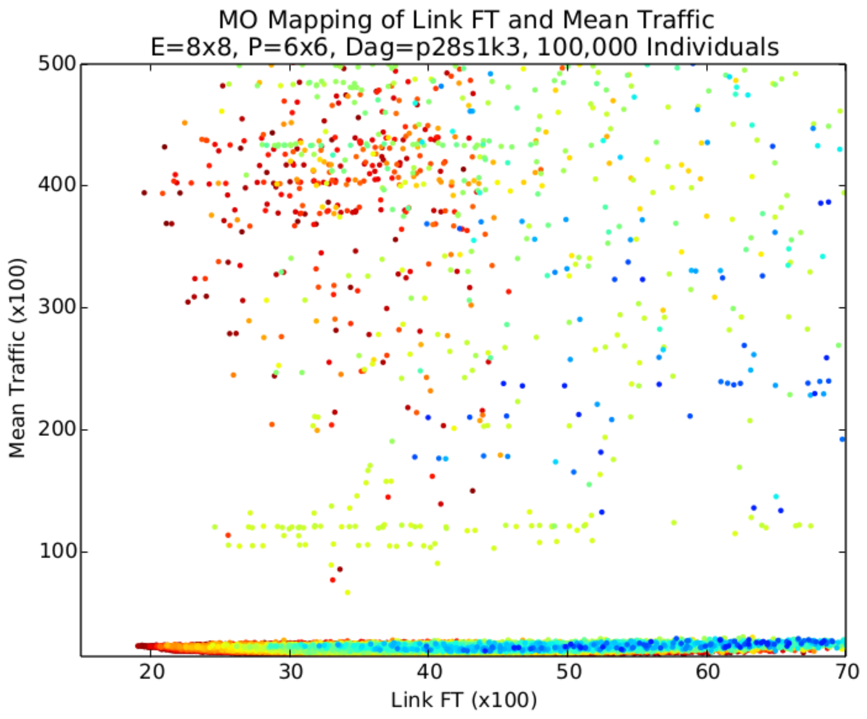
Mean Traffic and AMD Traffic - Figure 6.22

The Mean Traffic and AMD Traffic data plot is very different from all the previous plots. The narrow, long cigar shape aligned along the $x = y$ diagonal is a clear illustration that the mean traffic and AMD traffic are highly correlated which means that only one of the objectives is required.

The mean is a measure of the average traffic that flows through all the links. The AMD measures the deviation of the excess traffic, of the links, from the mean traffic. The expectation was that it should be possible to have all combinations of low/medium/high

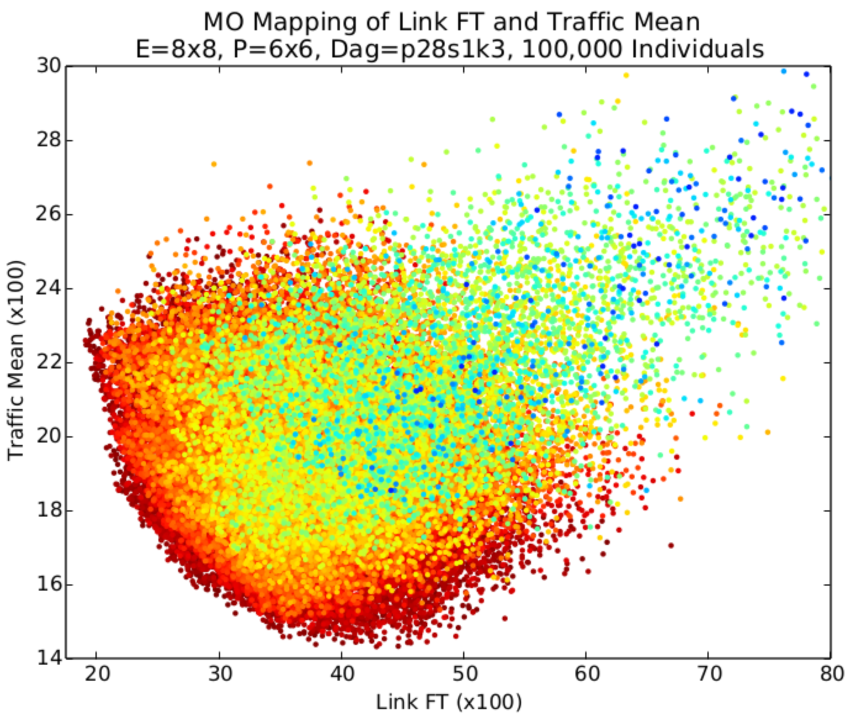


(a) Mean range 0 - 350,000

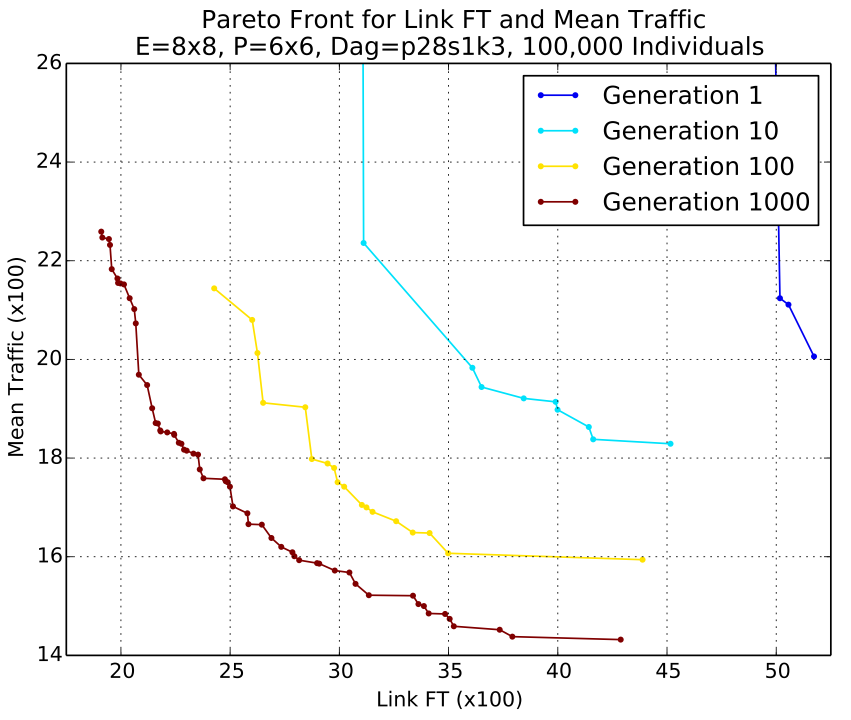


(b) Mean range 0 - 50,000

average mean traffic with low/medium/high deviation. It was also expected, that it should be possible to find solutions where, for example, the mean average was high, but the deviation from the mean was low which would indicate that the excess traffic was evenly spread across the network. That this is not the case requires explanation. A possible



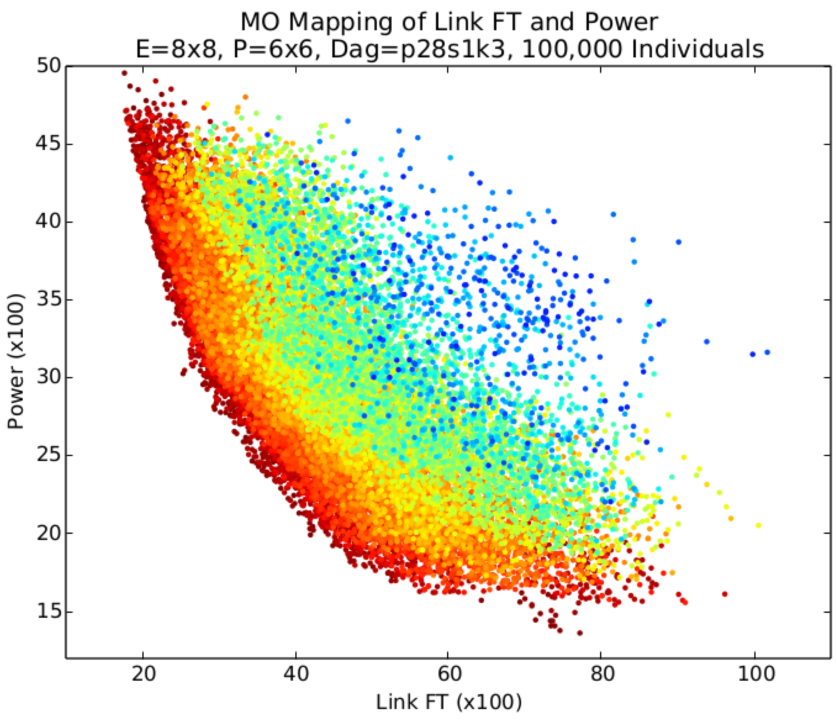
(c) Mean range 1,400 - 3,000



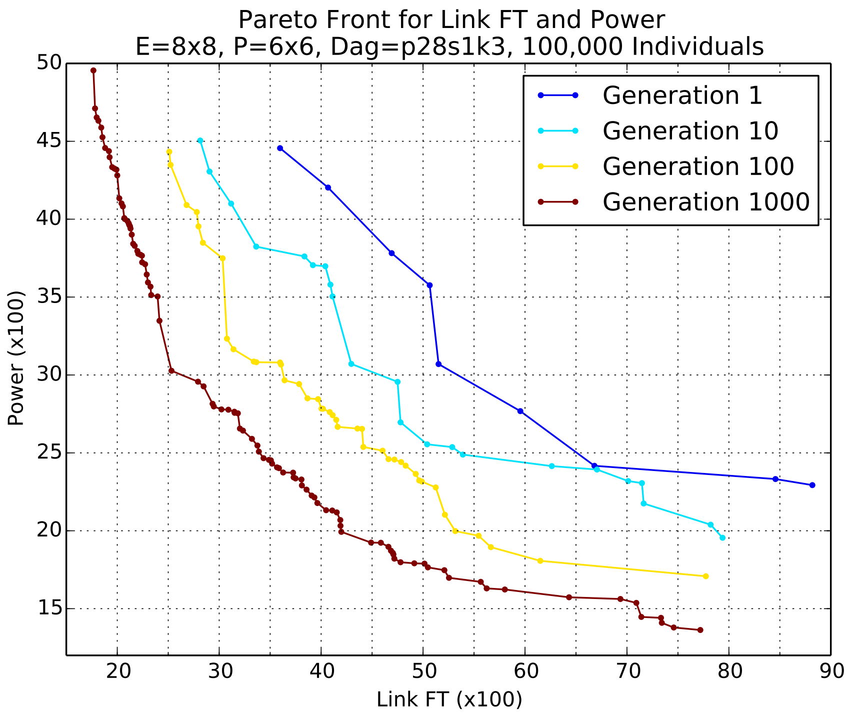
(d) Pareto Front Evolution

Figure 6.20 – Link FT with Mean Traffic

explanation is that the underlying data for both metrics is the excess traffic and the excess traffic is dominated by a small number of critical links because of the weighting applied to critical traffic. This would cause the deviation of the values of excess traffic to increase in



(a) Individuals

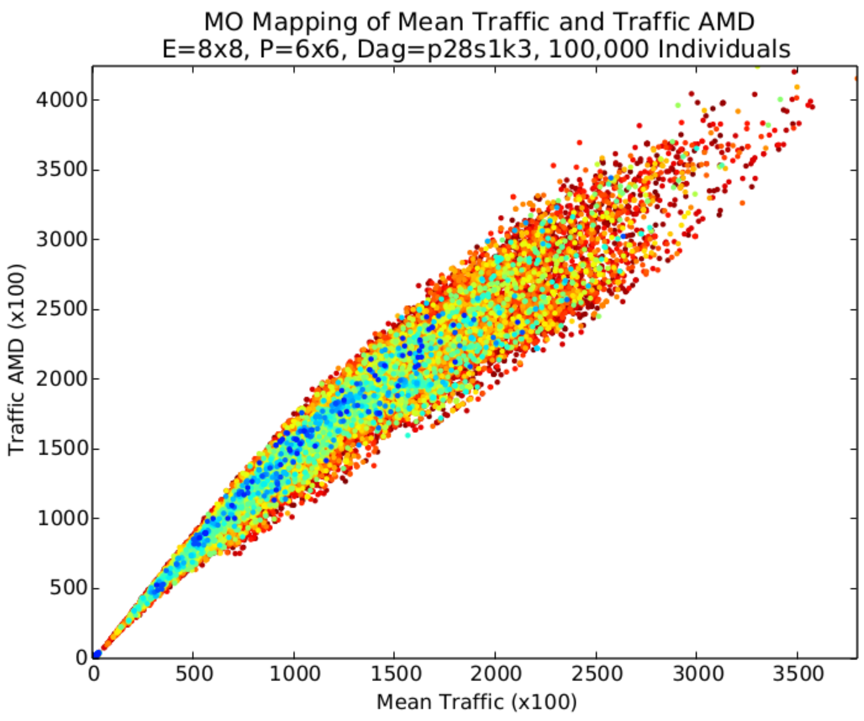


(b) Pareto Front Evolution

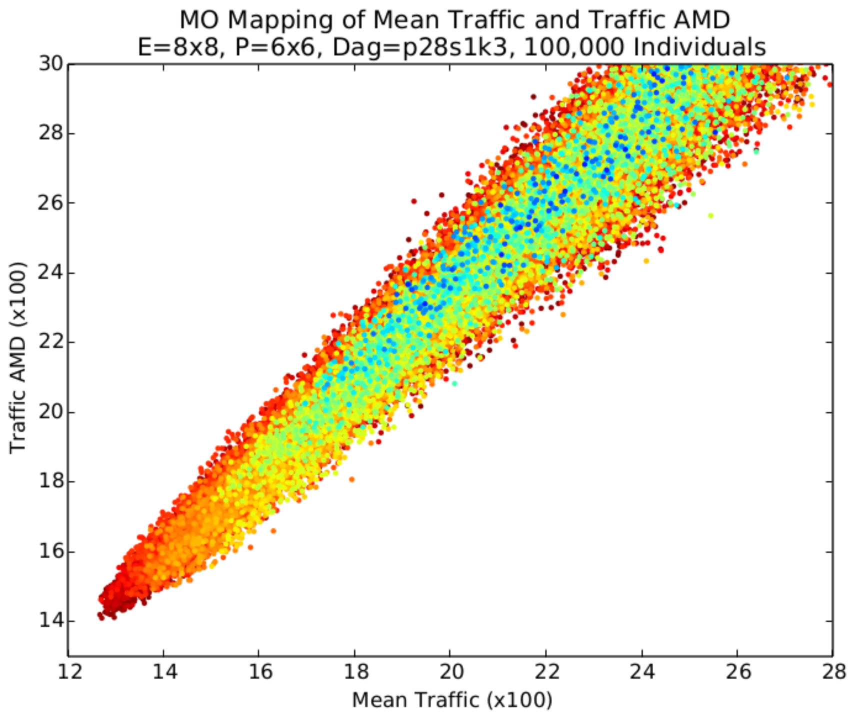
Figure 6.21 – Link FT with Network Power

proportion to the increase in the value of the mean the excess traffic.

Network Power and Traffic AMD - Figure 6.23



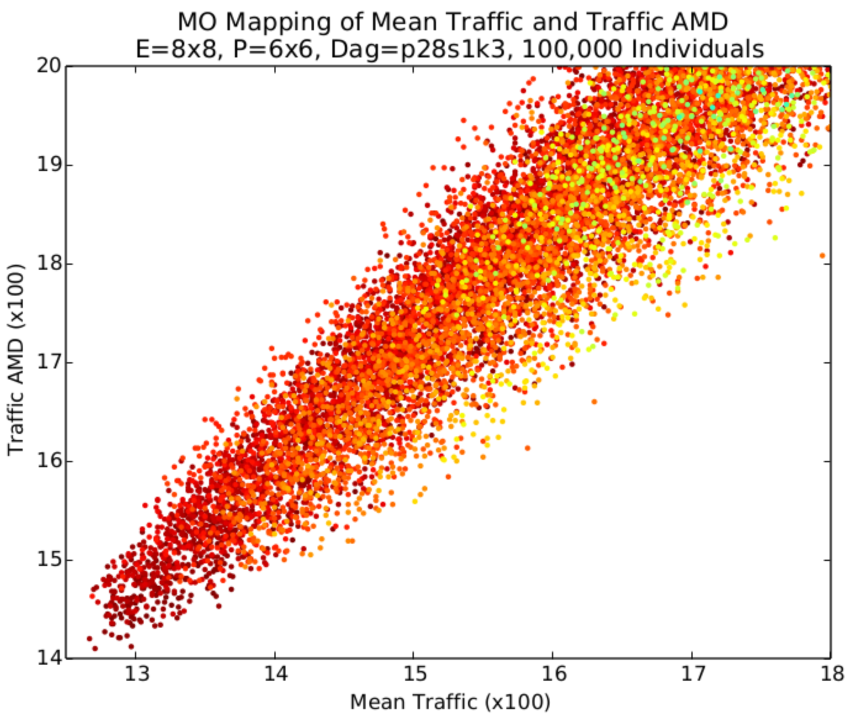
(a) Individuals



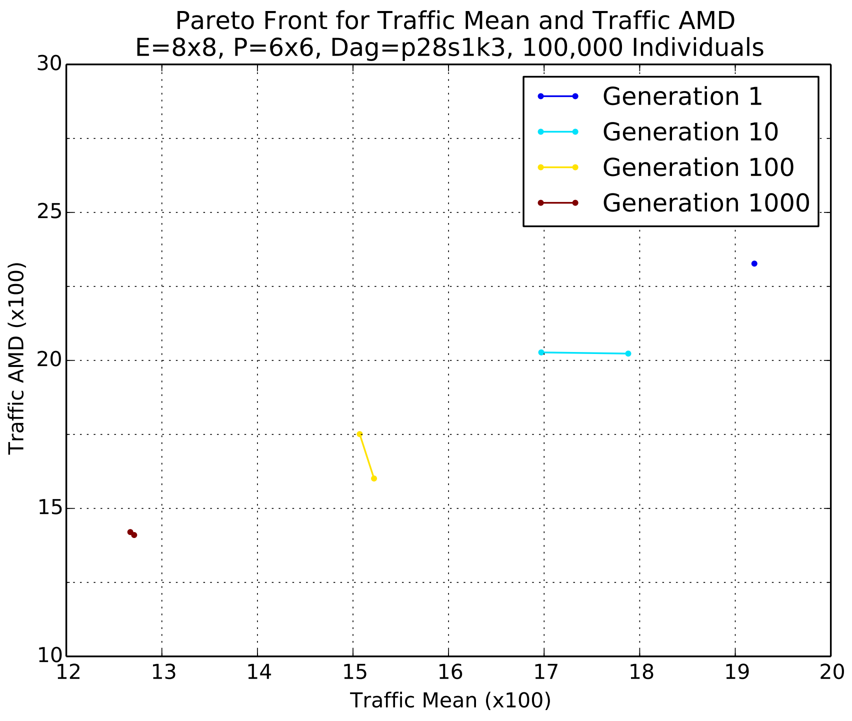
(b) Individuals

Network power and traffic AMD share similar properties to link-mean pairing shown in Figure 6.20, although the stratification is not as evident as for the link-mean pairing.

The Pareto front contains many points offering a good selection of trade-offs between the two objective.



(c) Individuals

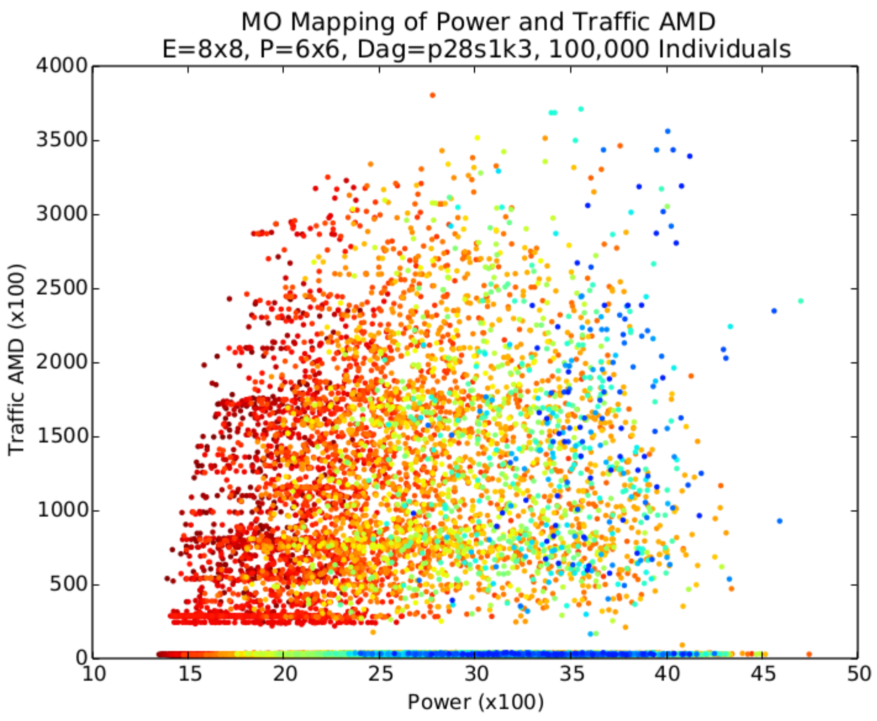


(d) Pareto Front Evolution

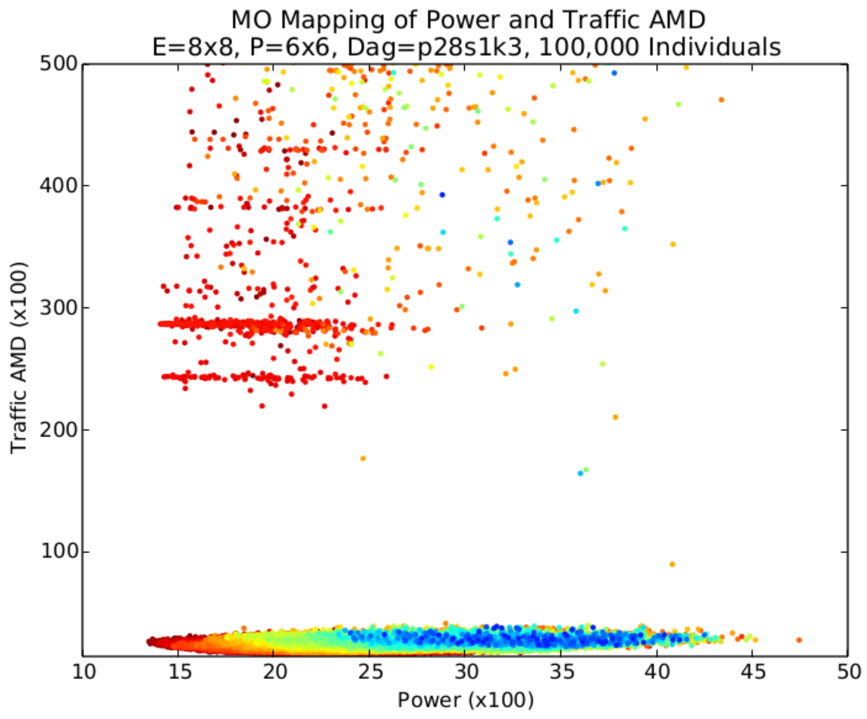
Figure 6.22 – Mean Traffic with Traffic AMD

Network Power and Mean Traffic - Figure 6.24

Network Power and Traffic Mean and Network Power and Mean Traffic both share similar properties to link-AMD. The Pareto front contains many points offering a good selection of

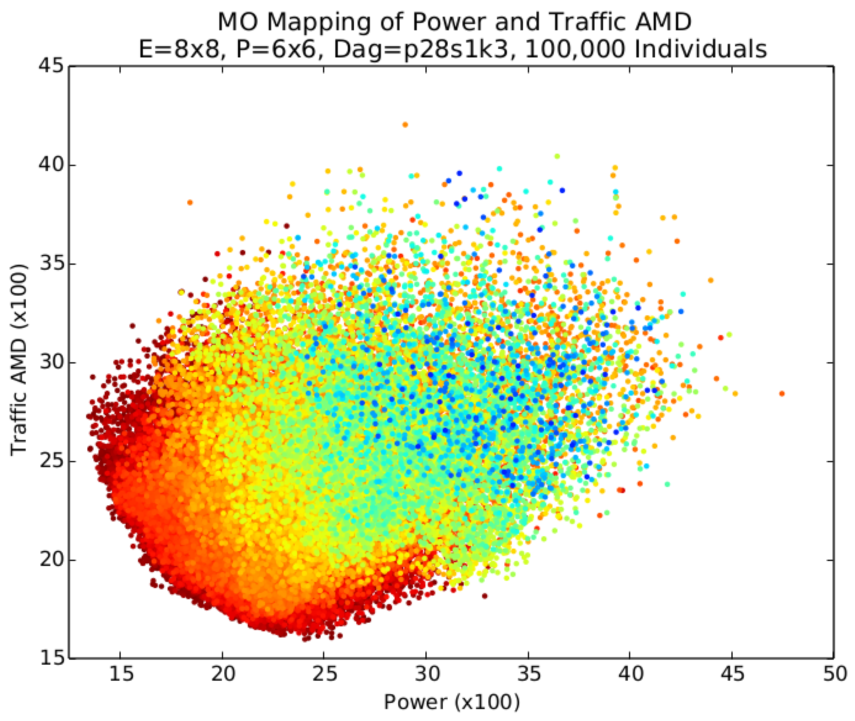


(a) AMD range 0 - 400,000

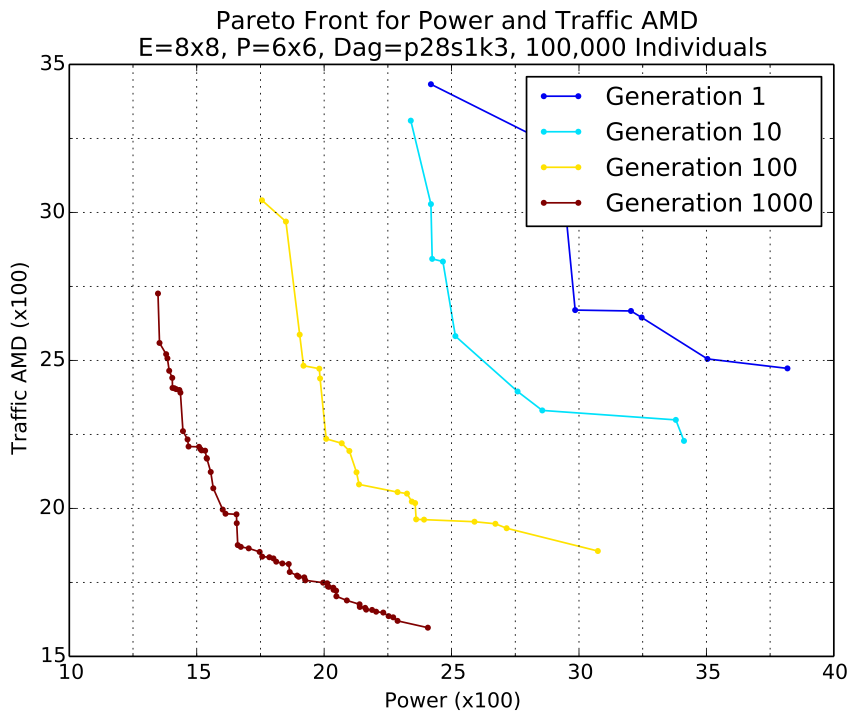


(b) AMD range 0 - 50,000

trade off between the two objectives, although the dataset in Figure 6.24 shows signs of alignment of data points along a $x = ky$ axis. Although not as pronounced as the cigar shape for the traffic mean and traffic AMD pairing, this does suggest that there is some correlation between the power and mean traffic objectives.



(c) AMD range 1,500 - 4,500

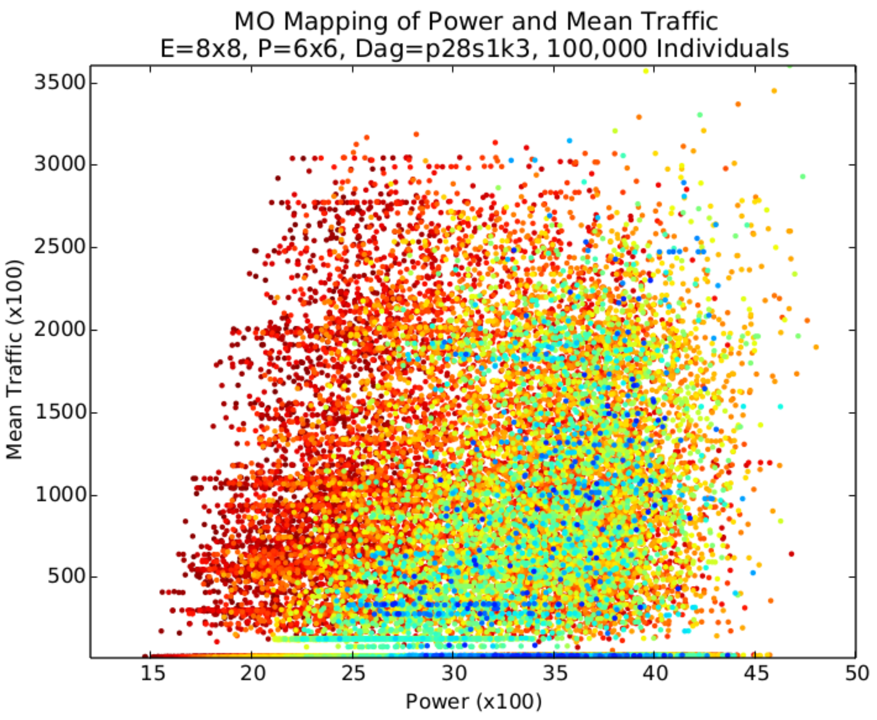


(d) Pareto Front Evolution

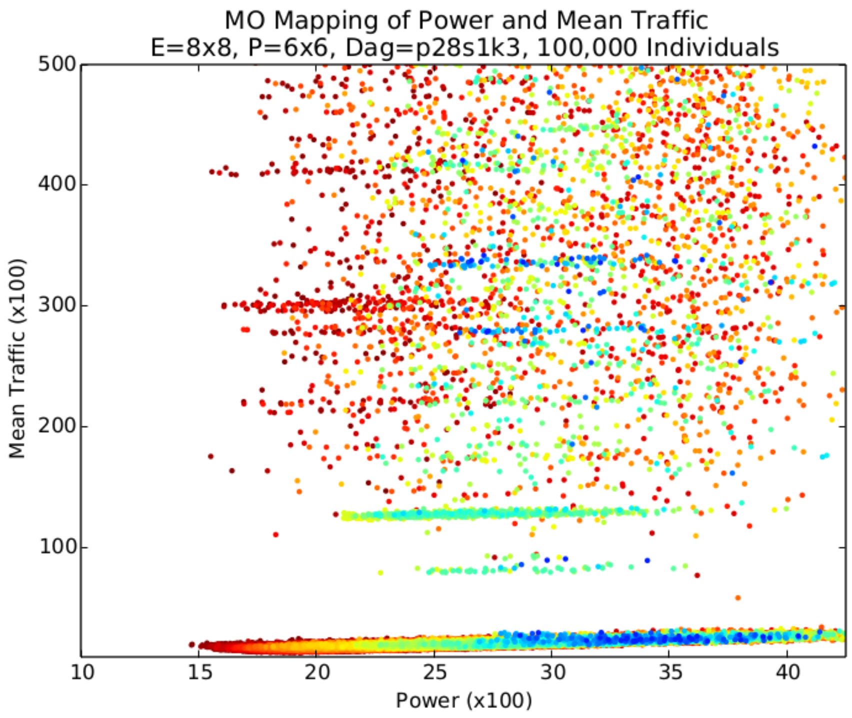
Figure 6.23 – Network Power with Traffic AMD

Examining the metrics, the power metric will be lowest when the nodes of a ComPair are close or adjacent, while the excess traffic metric will be lowest when the nodes of ComPairs are close together and on different rows and columns. The conditions for low metric values for this pair of objectives are similar, but not identical, which accounts for there being moderate correlation between them.

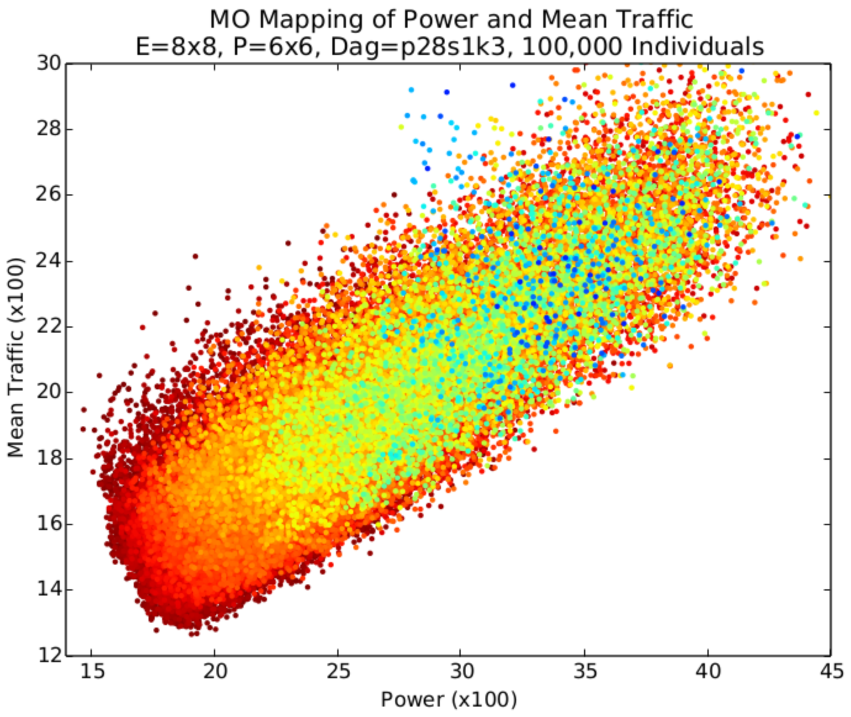
It is interesting to note the the correlation of traffic AMD, based on the dataset plot of Figure 6.22c, appears to be weaker than for the mean traffic dataset in Figure 6.24c, indicating the the correlation between mean traffic and traffic AMD may be strong but not absolute.



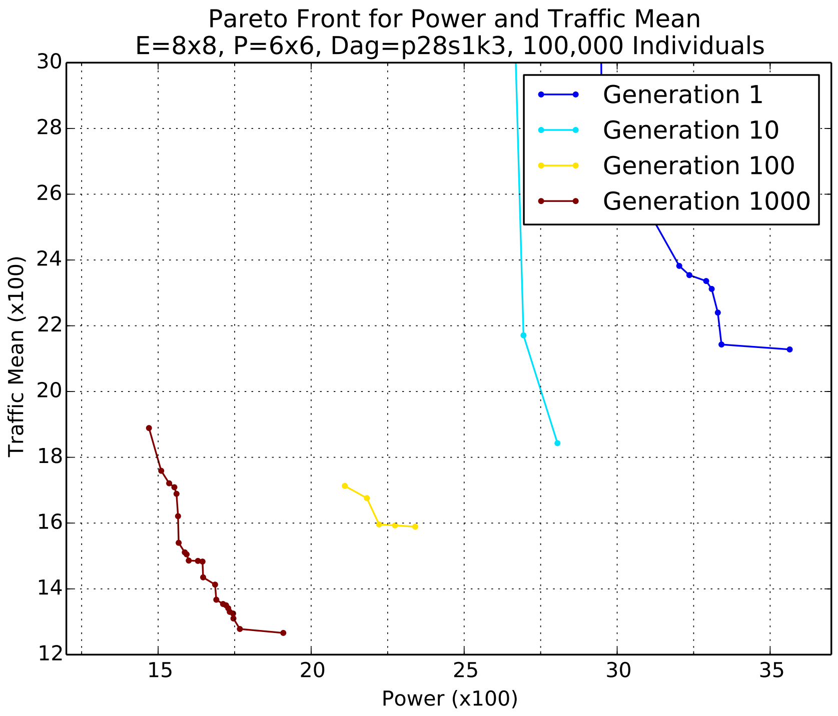
(a) Mean range 0 - 350,000



(b) Mean range 0 - 50,000



(c) Mean range 1,200 - 3,000



(d) Pareto Front Evolution

Figure 6.24 – Network Power with Mean Traffic

6.9 Conclusion

The results show that, for the majority of the objective pairs, the multi-objective evolutionary approach can find a good range of solutions with degrees of trade-off between the objectives, the most orthogonal pair of objectives being link fault tolerance and mean traffic.

The most correlated pair is the mean traffic and traffic AMD pairing. With existing critical weight parameters influencing the mean traffic and traffic AMD objectives, making them strongly correlated, only one of these objectives is required. Of these two objectives, the mean traffic objective is the natural choice, given that it is a cheaper calculation than the AMD objective.

Both the link fault tolerance and network power objectives work reasonably well with the traffic objectives even though there appear to be some correlation between the metrics.

The behaviour of the core fault tolerance objective with each of the others suggest that there may be a better approach than to include it as an objective in a multi-objective problem. The suggested approach is to first find optimal arrangements of the idle cores using core fault tolerance as a single objective. The core fault tolerance objective is very cheap to calculate and there are many possible arrangements of idle cores with the minimum possible objective value that can be discovered quickly with an evolutionary algorithm. A discovered idle core arrangement can then be selected and fixed for a multi-objective evolution of link fault tolerance, network power and mean traffic.

The results suggest that a multi-objective problem using core fault tolerance, link fault tolerance, network power and mean traffic will be effective in finding solutions that have a range of trade-offs of the competing demands of the objectives. An alternative to using four objectives is to fix idle core positions using a single objective evolution with core fault tolerance, and then perform a three objective evolution with the remaining three objectives.

For the multi-objective experiments in the next Chapter 7 *Graceful Degradation and Amelioration* will use the pairings of core fault tolerance with network power, link fault tolerance with network power and link fault tolerance with mean traffic.

During the development of the metrics and objectives, a formal description of a ComPair was provided. ComPairs are an important and heavily used construct in the calculation of the metrics involving the communication of traffic through the many-core array. An essential piece of information in at least one of the metric calculations is the number of paths that exist between the source and target nodes of a ComPair or, more generally, between any two nodes in a square lattice. There is a well known formula for calculating the number of paths between two nodes in a fault free lattice, however the author has been unable to find any references regarding a calculation in a lattice with faults, i.e. with broken links between nodes. Without the availability of a calculation, an algorithm would have to trace and count each viable path, which is a problem with $O(n!)$ complexity. As an activity that is required for every ComPair, for every mapping for every generation, this is a serious computational problem for any but the smallest of array sizes.

In response to this problem, an algorithm has been developed that can calculate the number of paths between two nodes in a faulty network, using only the position of the nodes and a list of faults in the network. The algorithm has been designed to work with directional edges and with the source and target nodes of a ComPair in any orientation. Although the algorithm has been designed for a two dimensional lattice, the principles are valid for a regular lattice of any dimension. In addition, it is possible to employ the algorithm to calculate paths through non-regular lattices by first converting the non-regular lattice to a regular lattice with faults and then using the algorithm to calculate the number of paths in the faulty lattice.

Chapter 7

Graceful Degradation and Amelioration

In this chapter the Monitor process and fault-recovery cycle are presented. The Monitor is an executive program that is responsible for collecting information from the many-core system and managing all aspects of the routers and cores of the many-core region which it oversees. The information that the Monitor collects is used to determine whether fault conditions have occurred. The fault-recovery cycle is managed by the Monitor and implements the concepts of graceful degradation and graceful amelioration to maintain fault tolerance and performance when the system suffers from faults.

When a fault occurs that requires the immediate migration of a process the migration is described as a *repair* to the mapping. The repositioning of the process within the array is likely to adversely affect the performance, an example of graceful degradation. The level of disruption that the application experiences during the migration is assumed to be dependent upon the distance between the cores between which the process is being migrated, the disruption increasing as the distance increases.

The results of two sets of experiments, the first set using single objectives followed by a set using two objectives, are presented that demonstrate that the fault-recovery cycle is effective in maintaining performance and fault tolerance when the system is affected by a series of faults. In the first set of experiments, within this chapter, the fault-recovery cycle uses each of core fault tolerance and link fault tolerance as single objectives. The second set of experiments applies the fault-recovery cycle to the combinations of: core fault tolerance and power, link fault tolerance and power, and link fault tolerance and mean traffic. Both sets of experiments, compare the fitness of the original mappings with re-evolved mappings through a series of fault-recovery cycles, that demonstrate the benefit of evolving new mappings after each fault.

7.1 The Monitor

The Monitor process is responsible for collecting performance data and managing the resources of the many-core array to comply with the performance parameters supplied to the Monitor. Examples of the functions that the Monitor node is responsible for are:

- Collecting traffic data
- Collecting thermal data
- Controlling voltage and frequency of individual cores
- Detecting fault conditions
- Managing the fault-recovery cycle
- Maintaining the hardware map with core and link faults
- Maintaining the process map
- Maintaining the application process graph with actual process-process traffic volumes
- Informing routers of the location of processes and faulty links
- Managing process migration
- Managing evolutionary runs to search for process mappings
- Maintaining Pareto Front *Pf0* between evolutionary runs
- Selecting suitable mappings from Pareto Front *Pf0*
- Communicating with adjacent regions (where they exist)

Note that if there are multiple regions, then there will be one Monitor process for each region. A Monitor processes will occupy one of the homogeneous processing cores in the region of the many-core array for which it is responsible, the specific core being determined during the initialization phase of the system; no special hardware is required for the Monitor process. What happens if the core fails, where the Monitor is located, is not within scope of this thesis.

Tasks of the Monitor, listed above, will be discussed as part of the fault recovery cycle where they apply.

7.2 Fault/Recovery Cycle

Previous chapters have demonstrated how the evolutionary algorithm using core fault tolerance and link fault tolerance objectives can produce mappings that are robust to faults. The core fault tolerance objective directs the search to find mappings that place idle cores close to processing cores to minimize the disruption to processing while repair and recovery takes place. The link fault tolerance objective directs the search to find mappings that will minimize disruption of communication between ComPairs when a link fails.

The mappings that have been found by the search directed by the fault tolerance objectives minimizes the disruption caused by the fault while the fault-recovery cycle returns the system back to normal operation.

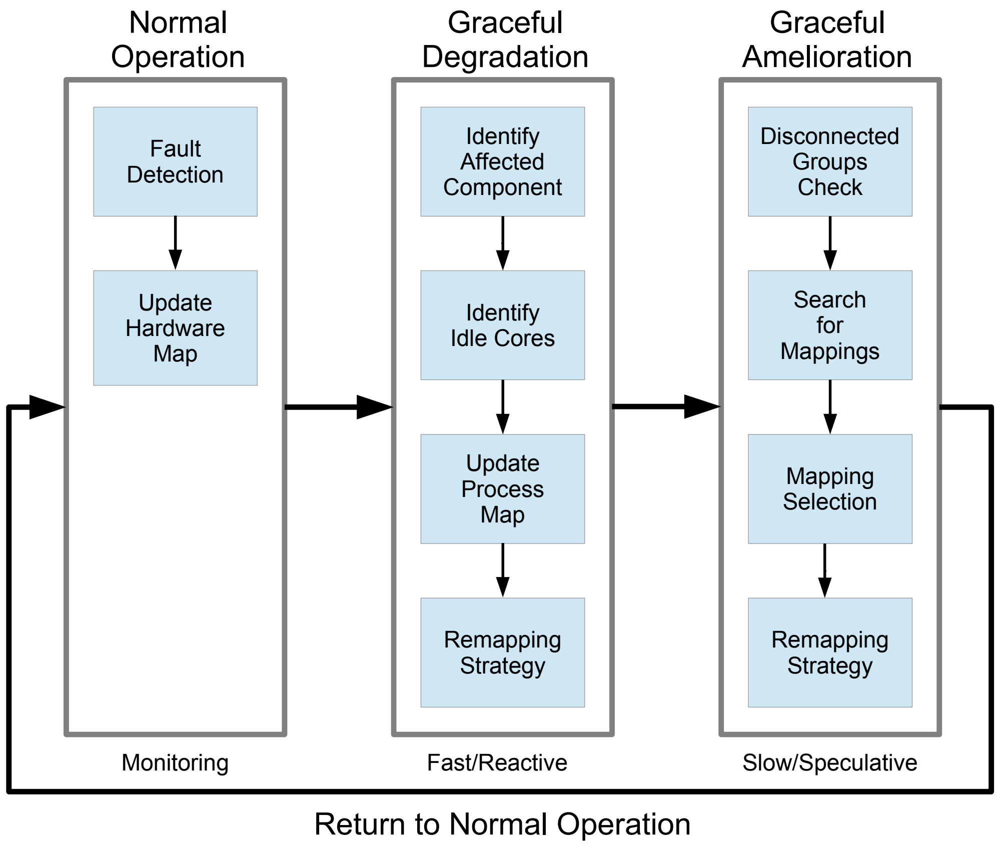


Figure 7.1 – **The Fault-Recovery Cycle**

The fault-recovery cycle is illustrated in Figure 7.1 which shows the state of the system being one of:

- Normal Operation
- Graceful Degradation
- Graceful Amelioration

When discussing the fault recovery cycle, it is assumed that mechanisms for task migration and state recovery and recovery of lost packets are available.

The remainder of this section will describe the elements of the fault-recovery cycle in detail.

7.2.1 Normal Operation

Normal operation is the desired steady state of the many-core system, during which cores perform the processing required of them. During normal operation the Monitor collects performance data that enables it to detect fault conditions.

Fault Detection
One of the functions of the Monitor is to use the performance data to identify when a fault condition occurs and then take appropriate action to return the system to normal operation.

The detection of faults is not a trivial problem, and is not one that this thesis attempts to solve. Analysis of traffic flow data can form the basis for identifying faulty cores, faulty links and traffic bottlenecks. Similarly analysis of thermal data can be used to identify hot-spots. Any of the these fault conditions can trigger the fault-recovery cycle. When the Monitor detects a fault event the state of the system moves to graceful degradation.

Examples of fault conditions are discussed below:

Core Faults are defined as the failure of a processing core. The function of the routing node and links to adjacent nodes are unaffected. If a core is processing a task when failure occurs, then the task needs to be migrated to a spare core.

Link faults are defined as the failure of a link between two adjacent routing nodes. The function of router nodes and their associated cores, at either end of the link are unaffected other than their inability to the use the affected link.

Routing node faults are defined as a failure of the routing node with the associated loss of the attached core and links, equivalent to the simultaneous failure of the processing core and the links to all adjacent nodes (which is how it is modelled in the hardware map). If the core is processing a task when the routing node fails, then the task needs to be migrated to a spare core.

Excess traffic destined for a link can have an adverse affect across the whole of the many-core array as the excess traffic is routed through alternative links, which may in turn create excess traffic on those links and etc. It may be possible to reduce traffic by slowing the clock rate of the processes that are generating high traffic volumes. However, it may be necessary to search for new mappings that reduce the overall traffic volume by rearranging processes to reduce the total distance between ComPairs. The Monitor will substitute the traffic volumes figure of the application process graph (which are only estimates), with the observed actual traffic volumes to increase the accuracy of the calculation of the excess traffic objectives.

Hot-spots are areas of the chip that are heating up to a level that may be harmful to the device if no action is taken. Hot spots can be alleviated in a number of ways, for example by reducing the clock rate and voltage of the processing cores in the hot-spot. If such adjustments are not effective then the Monitor can change the status of cores in the hot-spot to faulty in the hardware map. A search for new mappings will then produce only mappings that do not use the hot-spot cores. The Monitor will continue to monitor temperatures of the hot-spot cores and reinstate them as idle cores when the temperature has returned to an acceptable level making them eligible for inclusion in future mappings.

The fault conditions that have been used in the experiments of this section are core faults, link faults.

Update Hardware Map

It is essential that the hardware map reflects the actual status of the many-core hardware so the hardware map is updated as soon as the fault is detected.

Move to Fault-recovery cycle Graceful Degradation State
The Monitor moves the fault-recovery cycle to graceful degradation.

7.2.2 Graceful Degradation

The concept of *graceful degradation* is to allow an application to continue running on faulty hardware, although with degraded performance. Detection of a fault may require an immediate response to repair the fault.

Identify Affected Component
If a core fault has been detected then the Monitor will determine if the core was an idle core or a processing core. If the core was an idle core then no repair is required. If the core was a processing core then the Monitor will mange the repair by selecting a suitable idle core and migrating the process to it, allowing the system as a whole to continue to function even though performance may be degraded.

Identify Idle Cores
If core failure requires the migration of a process then the Monitor will execute a search to find the nearest idle core the to the failed core. This process is described in Section 4.7 *Nearest Spare Core Search*.

The failure of a normal link will not cause packet loss but may cause bottlenecks to appear in the remaining links, so does not require any immediate action to repair the fault.

Failure of a significant link will cause loss of some packets until the first up-stream routing node with a choice of paths to the target node establishes that a link has failed. Failure of normal or significant links do not need an immediate repair from the Monitor, although the Monitor may play a role in informing upstream routers of the failed link.

If a critical link fails, then either a non minimal path routing algorithm [74, 189] will be required to re-route traffic over non-minimal paths or a new mapping will have to be found that has at least one minimal length path for each ComPair. The source and target nodes, of a ComPair that has a critical link, share either a common row or common column with the critical link, the failure of which can be repaired by migrating the target node off the common row or column. If a failed link is critical to multiple ComPairs, then to guarantee repair, the target node of each ComPair must be migrated off the common row or column. Although possibly requiring multiple migrations, the migrations are sufficiently easy to identify and limited in number that they could be regarded as an immediate repair. If it is not possible to migrate each of the target nodes of the affected ComPairs then an graceful degradation is not possible and the state of the fault-recovery cycle will move to graceful amelioration.

This emphasises that a mapping with no critical links is highly desirable as no immediate action is required in the event of a link failure and the state of the fault-recovery cycle can immediately move to graceful amelioration.

Update Process Map

If repair requires the migration of one or more processes to spare cores and then the process map will be updated to reflect the new mapping.

Remapping Strategy
Having identifies the processes that need to be migrated and the spares cores that are the target of the migration, the Monitor will supervise the migration of the processes.

Move to Fault-recovery cycle Graceful Amelioration State
The values of the optimization objectives for the revised process map and hardware map are likely to be degraded compared to the original mapping, so must be recalculated to ensure comparison of the objectives with alternative mappings are accurate. The recalculation of the objectives can wait until after the migration of tasks is complete.

After the graceful degradation phase is complete the Monitor will move to the graceful amelioration mode.

7.2.3 Graceful Amelioration

Having recovered from the immediate fault, the next step is to recover fault tolerance and performance through graceful amelioration. The fault-recovery cycle therefore moves from degradation to amelioration.

Disconnected Groups Check
Multiple link faults in a many-core array will eventually lead to groups of cores being disconnected. When this occurs the metric calculations for ComPairs with their source and targets in disjoint groups of cores will fail. To prevent this only one group of cores can be used for the mapping of processes.

The fault-recovery cycle needs to include a process that will analyses the hardware map to determine if the cores have become disjoint groups. If the analysis determines that there are disjoint groups, all but the largest group will have all cores and link set to faulty. This ensures that the metric calculations will function correctly.

Note that the presence of disjoint groups of cores may have left the many-core system in a state where are no viable mappings (see Section 7.3 *Viable Mappings*).

Search for New Mappings
The Monitor will invoke the evolutionary algorithm to search for new optimal mappings using the updated hardware map, process map and application process graph. The evolutionary algorithm will use, as its initial population, the repaired mapping and all the individuals in Pareto Front *Pf0* from the last evolution. There is an assumption here that the addition of a single fault is likely to require only small changes to the mappings of the Pareto Front *Pf0* from the last evolution, to produce mappings that dominate the repair mapping.

Mapping Selection
The Monitor selects of a suitable mapping from Pareto Front *Pf0*.

The criteria for selecting an appropriate mapping will be determined by the relative importance of the objectives used in the search algorithm, which may change over time. How the relative importance of the objectives is determined is not within the scope of this research.

Remapping Strategy

Having selected a new mapping the Monitor will supervise the migration of processes to implement the new mapping, so achieving graceful amelioration. Multiple migrations may be required to implement the new mapping. If the system is still functioning then the migrations can be scheduled over a period of time, however, if the fault caused the system to become inoperable, then all migrations have to be completed before the system can resume processing.

Move to Fault-recovery cycle normal Operation State

Having completed graceful amelioration the Monitor will move the state of the fault-recovery cycle to normal operation.

7.3 Viable Mappings

When considering the fault-recovery cycle, the evolutionary algorithm will start with a many-core array with pre-existing faults, which can cause the evolutionary algorithm to create individuals that are not *viable*. A viable mapping is one for which there are sufficient functioning cores to host all of the processes from the application process graph and where all ComPairs have at least one communication route.

If the number of core faults results in there being fewer functioning cores than there are processes, then it will not be possible to find any viable mappings resulting in system failure.

Where there are sufficient cores for the processes, then faulty links could also result in some mappings being not viable. The evolutionary algorithm has the responsibility to ensure that it identifies and discards any non-viable mappings. It is possible to ensure that a mapping is viable by calculating the number of paths between the nodes of each ComPair. If there is at least one ComPair with no paths, then the mapping is not viable.

After individuals have been created through cloning, mutation, random generation or engineered, they are checked to ensure they are viable before being added to the intermediate population. If an individual is not viable, then another individual can be created to add to the intermediate population. Checking the viability of individuals at this point prevents a full objective evaluation being carried out on non-viable individuals.

If there is a series of link failures then there will come a point where there are either no viable mappings, or so few that they become unreasonably difficult to find. To ensure that the processes that create individuals for the intermediate population do not enter into an infinite loop while attempting to find a viable mapping, a parameter P_v, is used to determine the maximum number of consecutive non-viable individuals that can be created by a process before the attempt is abandoned. This ensures that the evolutionary algorithm will terminate when it fails to find any viable mappings.

7.4 Single Objective Graceful Degradation and Amelioration

In these experiments the cumulative effect of faults and the recovery process, for a single objective problem, are illustrated by graphs that plot the following four sets of fitness values:

- Evolved Fitness
- Faulty Fitness
- Re-evolved Fitness
- Original Repaired Fitness

Evolved Fitness
The evolved fitness plot is the minimum fitness found for the mappings discovered by the evolutionary algorithm. There are typically many possible mappings with the minimum fitness, one of which will be selected to map processes to cores in the many-core array. For each fault the evolved fitness is the fitness of the mapping before the fault occurred.

Faulty Fitness
When a fault occurs the fitness of the evolved mappings are re-evaluated to give the 'faulty fitness'. This is the best case scenario for the mapping currently in use by the many-core system. The faulty fitness of the evolved mappings can vary significantly as some will be more severely affected than others. This fitness gives a measure of the minimum disruption that the many-core system will experience from the fault.

Re-evolved Fitness
After a fault event the Monitor node will run the evolutionary process to search for mappings that are an improvement on the existing mapping. The re-evolved fitness is the minimum fitness of the newly evolved mappings. The re-evolved mappings for fault n will become the evolved mappings for fault $n + 1$.

Original Repaired Fitness
The original mappings are retained throughout all of the fault-recovery cycles. When each new core fault occurs, the original mappings are repaired by migrating the process from the failed core to an idle core. If this is not done then the original mapping will quickly become non-viable. If the fault is a link fault then no repair is attempted. The objective is then re-evaluated so that the fitness of the repaired original mappings can be compared with the re-evolved mappings.

Fault Cycles
For core fault experiments, the graphs show seven iterations of the fault repair cycle. The eighth fault removes the last idle core which renders the core fault tolerance measure moot

as there is no idle core that the metric calculation can use. For this reason only the first seven faults are included.

For link faults the decision was made to use a maximum of seventeen fault-recovery cycles which is 15% of the number of links in a 6×6 array. This limit may seem somewhat arbitrary but proved to be sufficient to demonstrate the value of the fault-recovery cycle, while being able to complete the tests within an acceptable time period. Experiments with link failures have the potential of not being able to find viable mappings. To ensure the tests terminated, a limit of twenty attempts to find a viable mapping at each step of the evolutionary algorithm was imposed. Some experiments ended without finding viable mappings before the maximum number of link faults had been applied.

7.4.1 Evolutionary Algorithm Parameters

The experiments are run using the evolutionary algorithm parameters selected in Section 5.5 and listed in Table 7.1.

Table 7.1 – Multi-Objective Fault Free Experiment Parameters

Parameter	Value
Hardware Map Size	8×8
Hardware Map border	$B((b,1),(b,1),(b,1),(b,1))$
Many-Core Array Size	6×6
Number of generations	1000
Population size	100
Elite	10 fittest individuals cloned
Parents	20 fittest individuals used as parents
Descendants	4 from each parent via permutation
Novel	10 randomly generated individuals
Mutation Pattern	g15
Non-viable mapping limit	20

7.4.2 Graph Selection

The experiments are repeated for each of the three density of graphs listed in Table 7.2 The graph names are based on the number of processing nodes, source nodes and sink nodes, so the first graph with 28 processing (p) nodes, 2 source (s) nodes and 1 sinks (k) nodes is referred to as p28s2k1.

The graphs all have 28 processing nodes which will be mapped onto a 6×6 array within a 8×8 hardware map with a border of $B((b,1),(b,1),(b,1),(b,1))$. For each pair of experiments the source and sink nodes are randomly allocated to the source and sink nodes of the environment. No attempt has been made to force all the experiments to place

Table 7.2 – Test Application Process Graphs

Parameter	Value
Graph p28s2k1	28 Process Node 2 Source Nodes 1 Sink Nodes Sparsely Connected Figure 6.1
Graph p28s1k3	28 Process Node 1 Source Nodes 3 Sink Nodes Densely Connected Figure 6.3

the sources and sinks at the same locations. The actual position of the source and sink processes will have an effect on the values of the objectives, but is not expected to have an effect on the correlation of a pair of objectives.

7.4.3 Data Presentation

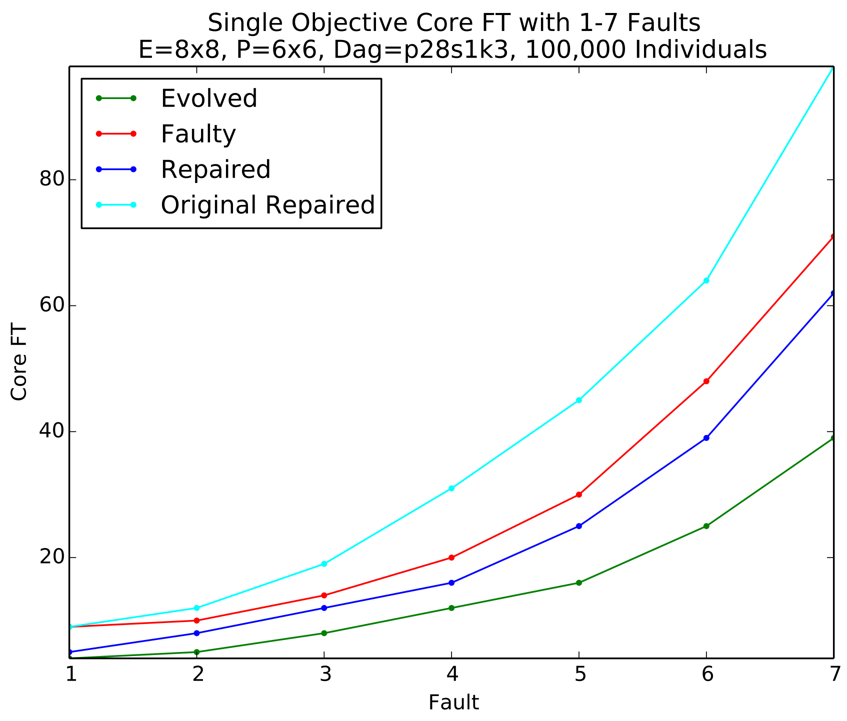
When interpreting the plots, the absolute value of the objectives is not important, of more interest is the how the objectives relate to each other, therefore the range of values, and scale of axes have been chosen for each individual plot to achieve the best presentation of the data to illustrate the relationship between the two objectives.

7.4.4 Results

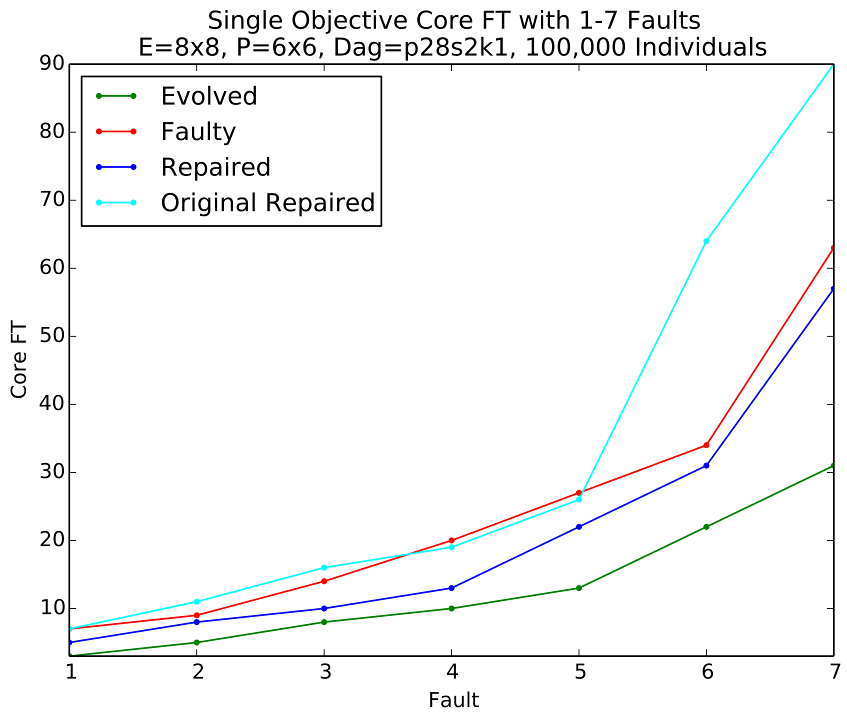
Core Faults - Core Fault Tolerance - Figure 7.2
The graph in Figure 7.2a show the results when a series of core faults are applied to the mapping of a densely connected application process graph optimized for core fault tolerance, while the graph in Figure 7.2b show the results when a series of core faults are applied to the mapping of a sparsely connected application process graph also optimized for core fault tolerance.

The evolved fitness (green line) is the pre-fault fitness immediately before each fault is applied. When a fault is applied there is a clear deterioration of the fitness, represented by the faulty fitness (red line). The re-evolved fitness (blue line) shows that re-evolution improves upon the repaired mappings. The cyan line represents the original mapping that is repaired after the application of each fault and can be seen to quickly become worse than either the repaired or re-evolved mappings, as the number of faults accumulate.

The sparsely connected graph maintains lower fitness values than those of the densely



(a) Single Objective Core Fault Tolerance for a Densely Connect Graph



(b) Single Objective Core Fault Tolerance for a Sparsely Connect Graph

Figure 7.2 – Single Objective Fault Recovery for Core Fault Tolerance

connect graph, through all fault-recovery cycles. The cyan line, which represents the original mapping that is repaired after the application, appears to show greater resilience to the series of faults than the densely connected mapping. However, since the density of the graph has no effect on the core fault tolerance objective the observed differences between the two graphs must be normal statistical variations the core fault tolerance is unaffected by the number of ComPairs in the network.

These results demonstrate that the repair process of migrating a process from a failed core to an idle core can keep an original mapping viable, but to maintain the core fault tolerance objective as low as possible requires re-evolution.

Link Faults - Link Fault Tolerance - Figure 7.3
The graph in Figure 7.3a shows the results when a series of link faults are applied to a mapping of a densely connected application process graph optimized for link fault tolerance, while the graph in Figure 7.3b shows the results when a series link faults are applied to a mapping of a sparsely connected application process graph also optimized for clink fault tolerance.

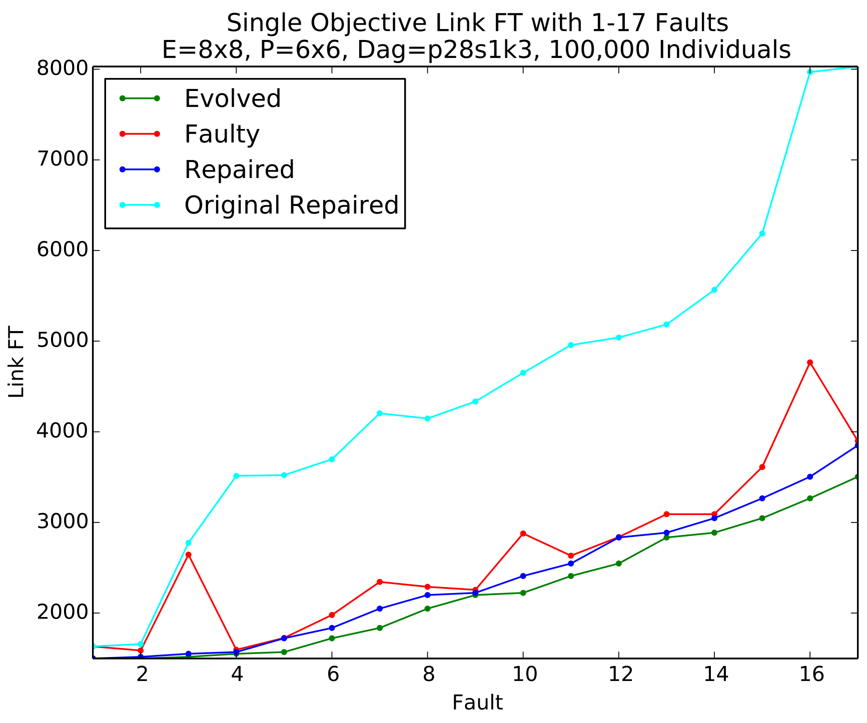
These graphs show that the effect of a link failure on the objective value of link fault tolerance can vary considerably. In many cases the effect is moderate, in others the effect is significant. The significant effect on fitness is due to a significant link becoming a critical link, which will have the dual effect of the reducing the number of links between a ComPair to one and then penalising the link because it is the only remaining link. Evolving new mappings works well, bringing the fitness back down towards the original evolved value and in some cases finding mappings with a superior fitness to the original evolved mappings.

The fitness of the original mapping that is left unchanged through the series of fault injections, quickly deteriorates, demonstrating the value of the fault-recovery cycle. In Figure 7.3a, the original mapping became non-viable after the tenth fault. Following the seventeenth fault the evolutionary algorithm did not find ay viable mappings so the recovery phase of the cycle failed.

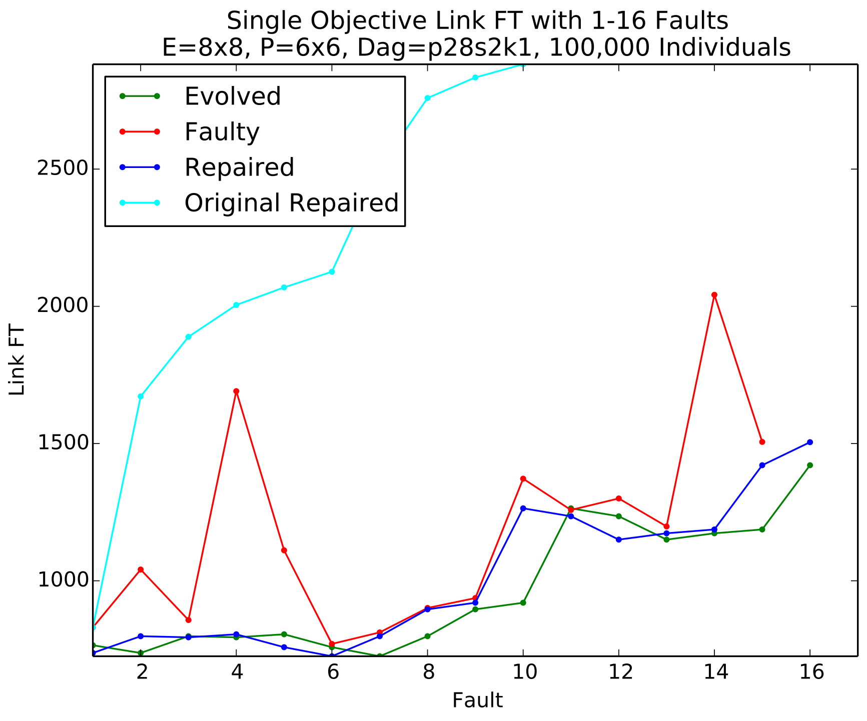
These graphs suggest an improvement to the fault recovery cycle: if the change to the fitness value is small, then the improvement in the objective value that can be achieved through evolution is minimal, suggesting that this step could be omitted. This approach would be useful for avoiding unnecessary and unproductive evolutionary cycles.

7.4.5 Summary

This set of experiments demonstrated that the fault-recovery cycle implementing graceful degradation and amelioration can maintain the performance of a many-core system by using an evolutionary algorithm to search for new mappings after a fault occurs. A refinement of the process could take into account the level of degradation of the objective values to decide if using the evolutionary algorithm is an effective use of processing time compared to the benefits in performance or fault tolerance that would result.



(a) Single Objective Link Fault Tolerance for a Densely Connect Graph



(b) Single Objective Link Fault Tolerance for a Sparsely Connect Graph

Figure 7.3 – Single Objective Fault Recovery for Link Fault Tolerance

7.5 Multi-Objective Graceful Degradation and Amelioration

In these experiments the effect of faults and the recovery process for a multi-objective problem are illustrated by a sets of four graphs each plotting a series of Pareto fronts for each occurrence of a fault. The graphs display the following Pareto fronts:

- Evolved
- Faulty
- Re-evolved
- Original repaired

The Pareto fronts are the multi-objective equivalents of the single objective plots for the experiments in Section 7.4.

Evolved
The Pareto front of the evolved solution at the start of the cycle before any a fault occurs.

Faulty
After a fault occurs and any repairs that are possible have been made, the points on the Pareto front are re-evaluated. If an individual becomes non-viable, it is removed from the list.

Re-evolved
The evolutionary algorithm is used to search for a new Pareto front in an attempt to improve performance over the repaired individuals. This Pareto front will become the evolved Pareto front for the start of the next cycle.

Original mapping
The original mappings, repaired if possible, are retained throughout all of the fault/repair cycles. This Pareto front represents the original mappings re-evaluated with the current hardware map. If an individual becomes non-viable, it is removed from the Pareto front.

7.5.1 Evolutionary Algorithm Parameters

The same parameters that are specified in Section 7.4 are used for these experiments.

7.5.2 Graph Selection

The same graphs that are specified in Section 7.4 are used for these experiments.

7.5.3 Data Presentation

Pareto Front Plots
When interpreting the plots, the absolute value of the objectives is not important, of more

interest is the how the objectives relate to each other, therefore the range of values, and scale of axes have been chosen for each individual plot to achieve the best presentation of the data to illustrate the relationship between the two objectives.

Core Faults applied to Core fault tolerance with Power - Figure 7.4

Figure 7.4 shows the results for a series of core faults impacting a many-core system using mappings optimized for the multiple objectives of core fault tolerance and network power.

These results show the Pareto front for 7 iterations of the fault-recovery cycle, colour coded for each fault. Comparing Figure 7.4b with Figure 7.4a shows the effect of the faults on the fitness of the individuals in the Pareto front, showing a clear degradation in fitness. In this case the degradation resulting from each of the first five faults is moderate, however as the pool of idle cores becomes depleted the effect of each new fault becomes more severe.

The benefit of re-evolving is illustrated in Figure 7.4a, which demonstrates an improvement of the individuals on the Pareto front compared to the faulty Pareto front mappings of Figure 7.5b and the repaired mappings of Figure 7.5c.

The original re-evaluated Pareto front in Figure 7.4d illustrates how the fitness of the original mappings rapidly deteriorate if the mappings are repaired and remain in use when the next fault occurs.

These results are consistent with the results of Section 7.4, giving confidence that the evolutionary algorithm works effective in single and multi-objective modes.

The effect of faults appears more erratic in these experiments when compared to the single objective results. This is due to the single objective experiments only using an individual with the best fitness. The Pareto front of mappings optimized for multiple objects result in mappings with a range of fitnesses of the twp objectives ranging from poor to good. Each mapping, representing a different balance of the two objectives, can respond very differently to the same fault.

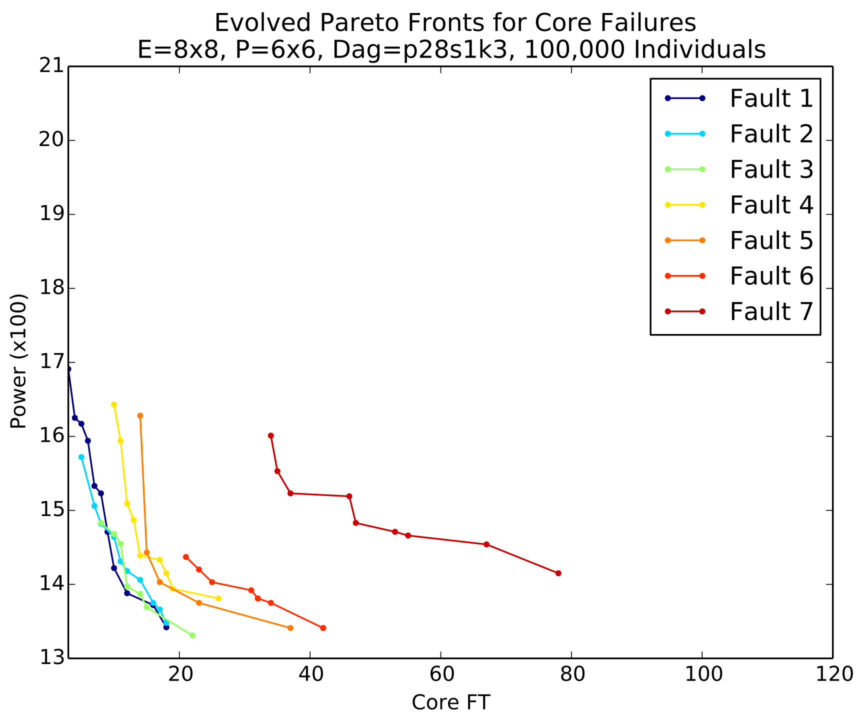
A second set of results in Figure 7.5, with only the connection density of the graph changing from dense to sparse, show a similar picture to the densely connected graph.

Core Faults applied to Link fault tolerance with Power - Figure 7.6

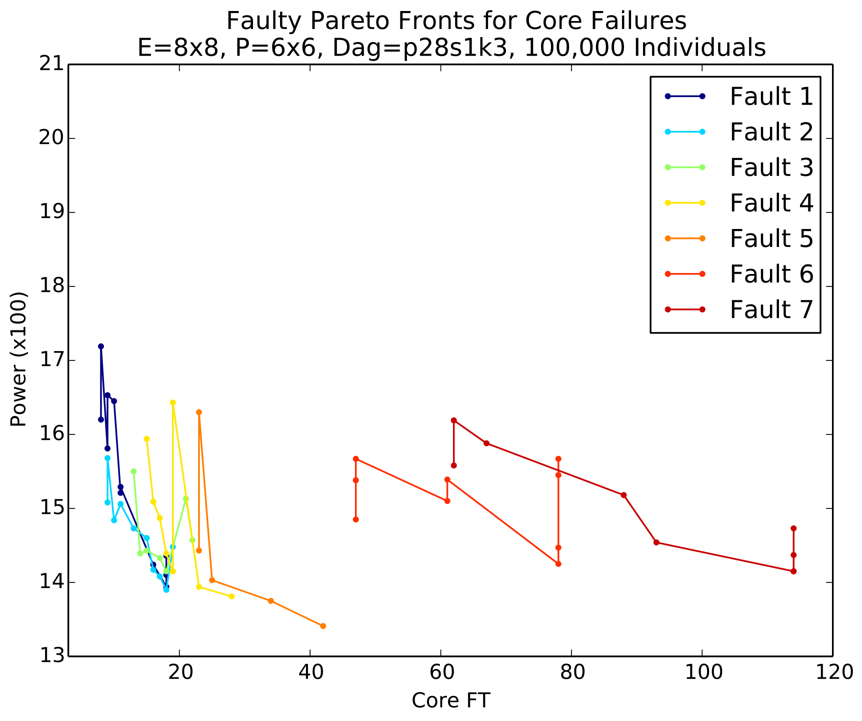
Figure 7.6 shows the results for a series of core faults impacting a many-core system using mappings optimized for the multiple objectives of link fault tolerance and network power.

The plots of the objective values for the mappings after a fault has occurred, Figure 7.6b, exhibit a series of near vertical lines connecting the individuals. This indicates that the network power fitness is affected while the link fault tolerance fitness remains unchanged. The loss of a core, which can require the migration of a process to repair, can affect the value of the network power objective by significantly increasing the path length for some ComPairs, however the link fault tolerance objective value is unaffected by a core failure.

The vertical lines show that there are many instances where there are two individuals that are very closely placed on the Pareto front where the change in fitness in reaction to the

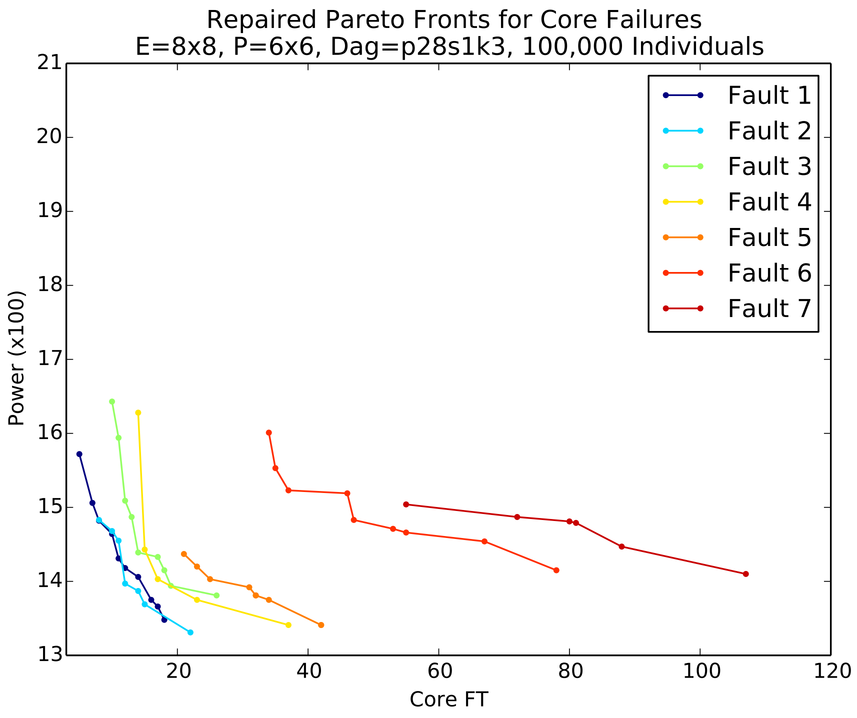


(a) Evolved

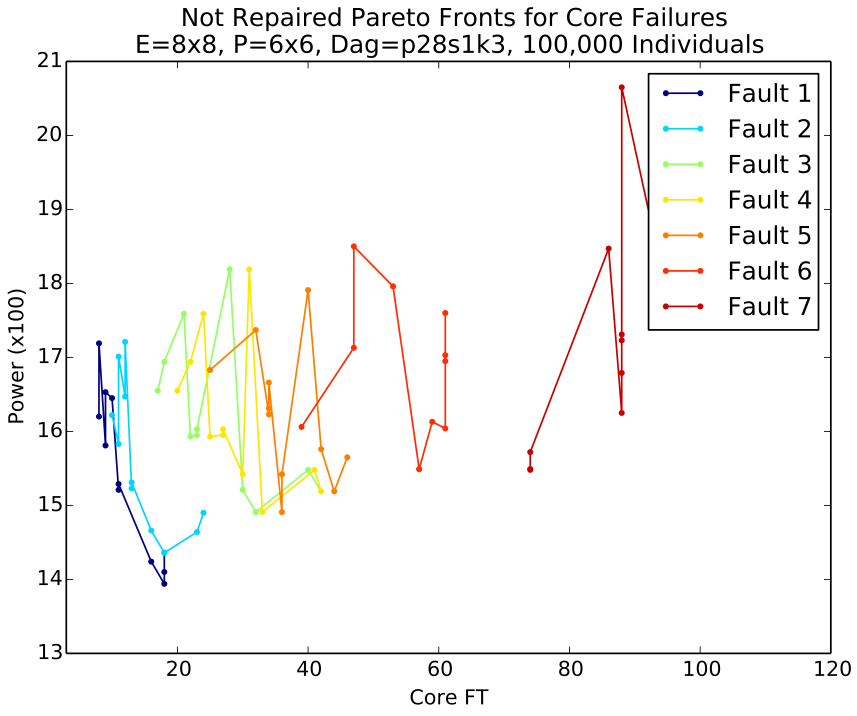


(b) Fault Added

same fault is very different. The mappings only need to differ in the use of the failed core, with one using it as an idle core and the other as a processing core, for the fault and repair mechanism to have a profound effect on one individuals while having little affect on the other individual.



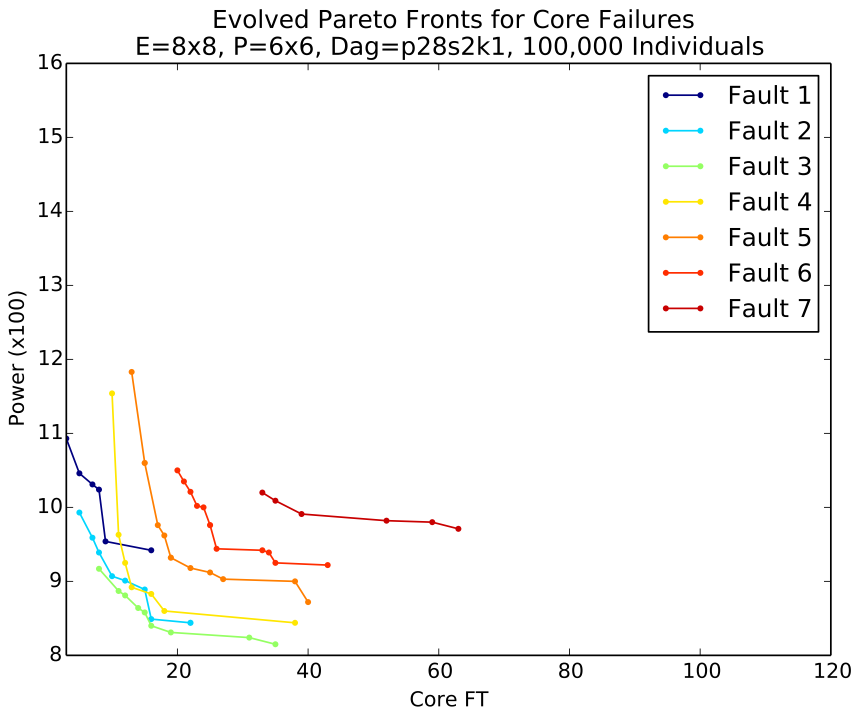
(c) Repaired



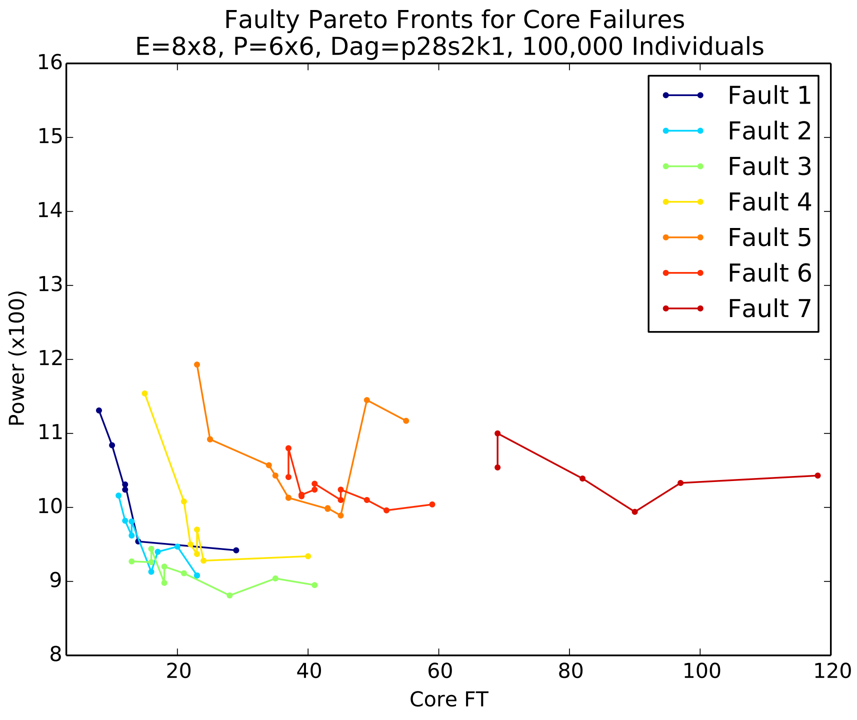
(d) Original Repaired

Figure 7.4 – Multi-objective, Core Fault Recovery, Core fault tolerance with Power, Densely Connected Graph

This is an example of where the crowding distance between individuals on the Pareto front is a poor guide to how similar or different the individuals are. Although the fitness values



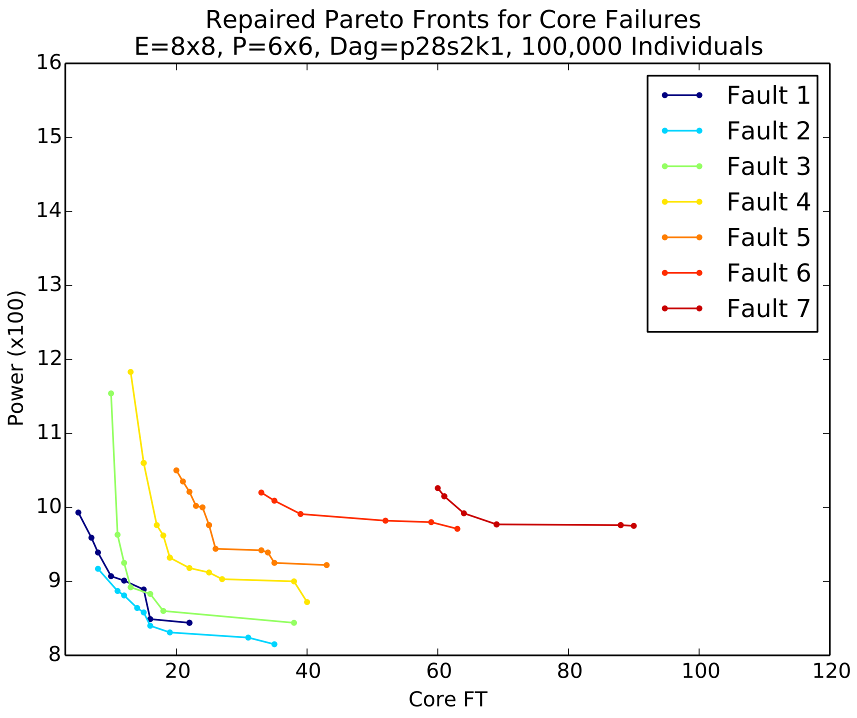
(a) Evolved



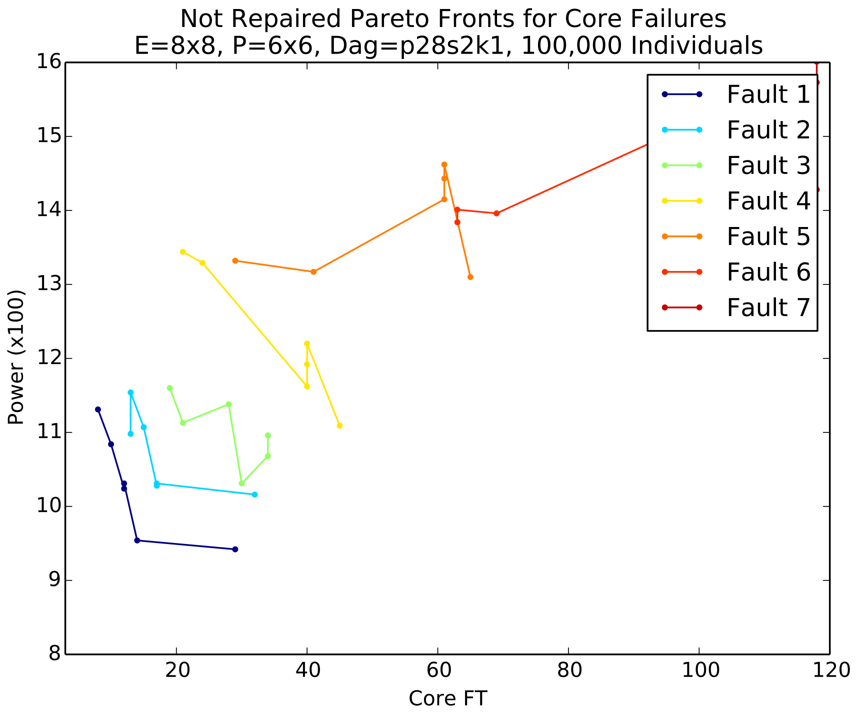
(b) Fault Added

are very similar, the mappings are sufficiently different that the qualitative difference of the individuals is large, which results in a very different response to the same fault condition.

By deciding not to use crowding distance for the selection of individuals from the Pareto front, the evolutionary algorithm retained individuals with similar fitnesses but very different



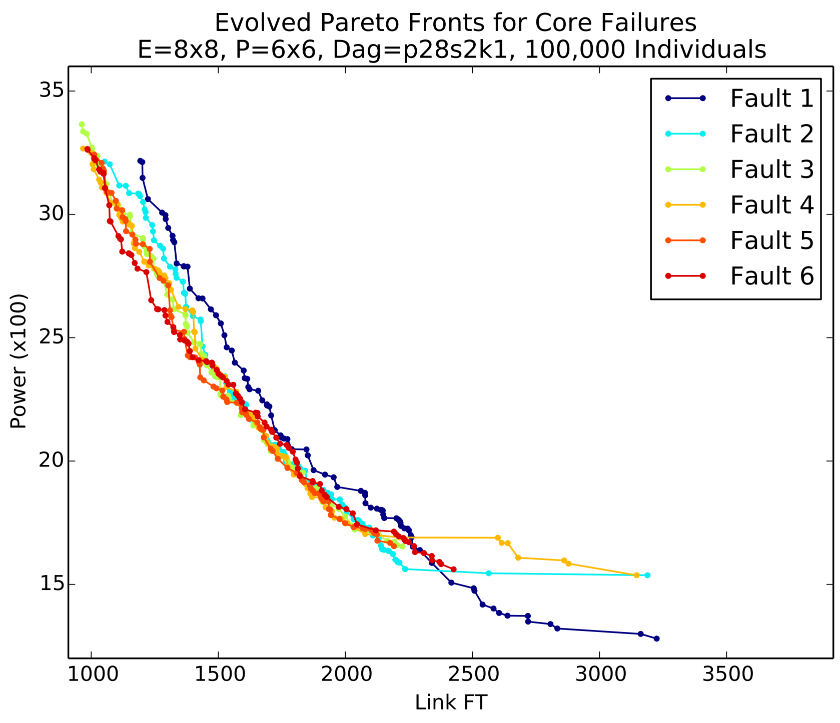
(c) Repaired



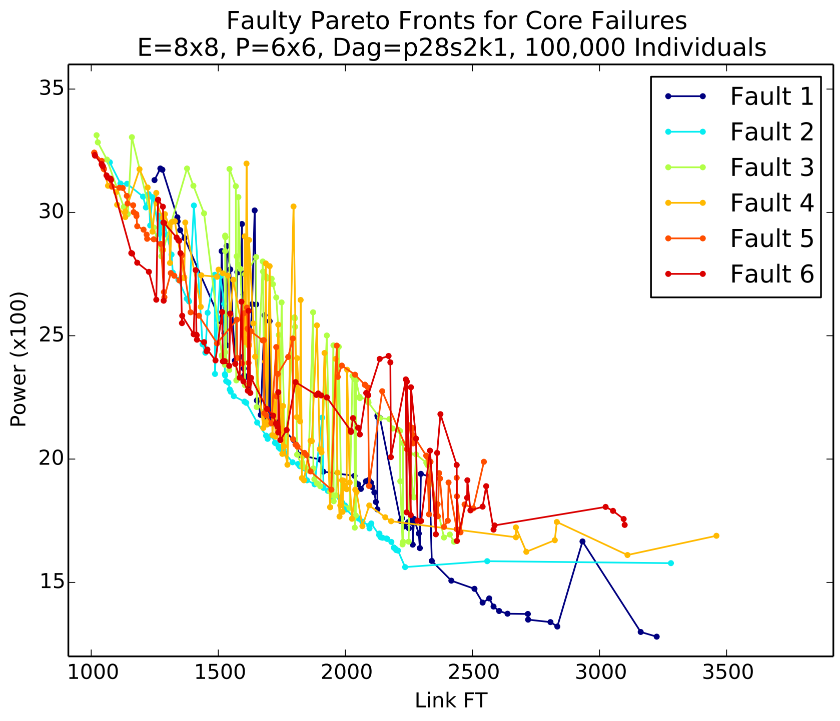
(d) Original Repaired

Figure 7.5 – Multi-objective, Core Fault Recovery, Core fault tolerance with Power, Sparsely Connected Graph

underlying properties which increases the genetic diversity of the individuals in the Pareto front.



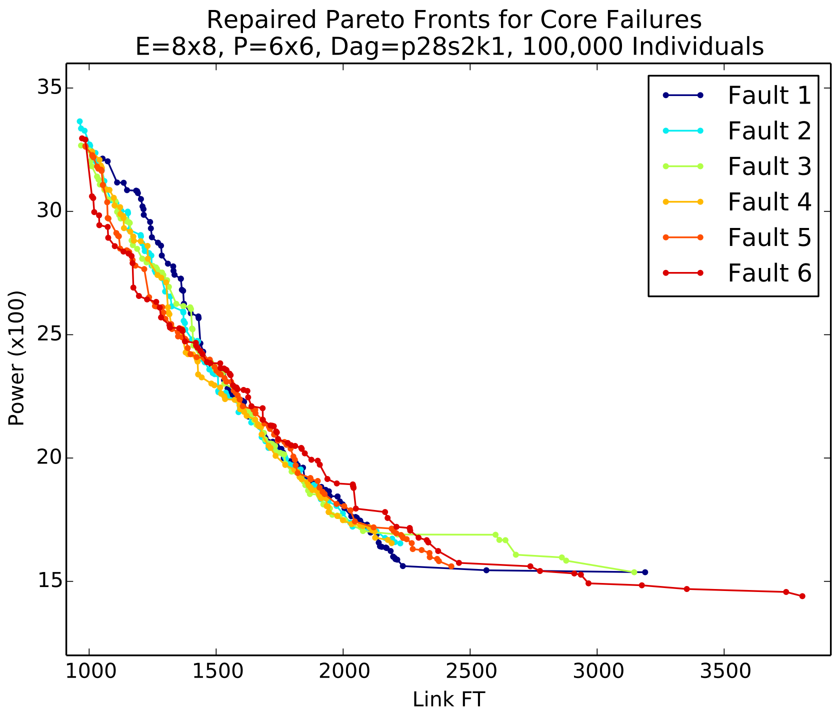
(a) Evolved



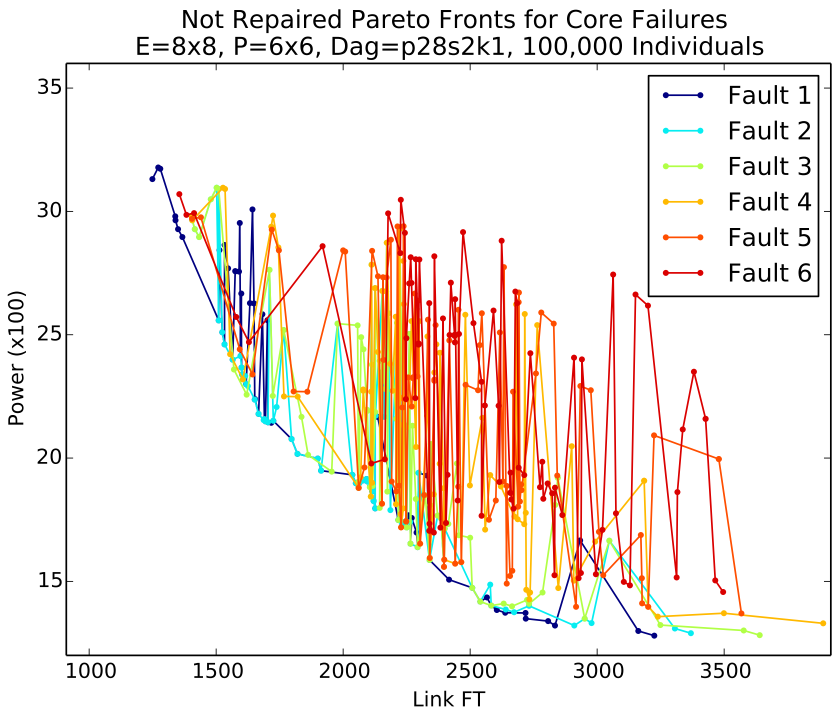
(b) Fault Added

The original points of the Pareto front show a particularity erratic picture of the effect of cumulative faults on the neighbouring points. It is evident that the evolution of new solutions is very beneficial to maintaining good quality mappings.

Link Faults applied to mappings for Link fault tolerance with Power - Figure 7.7



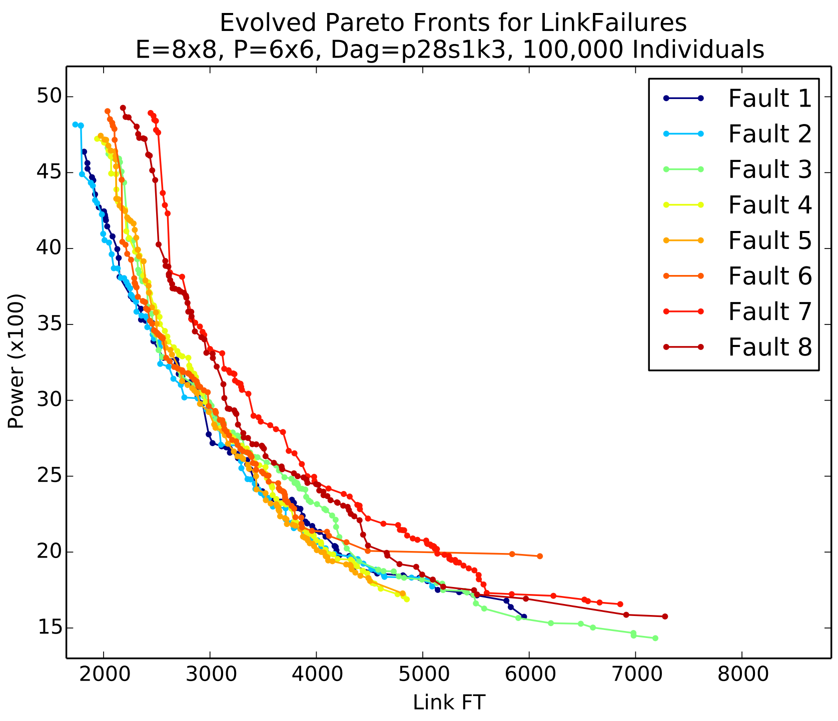
(c) Repaired



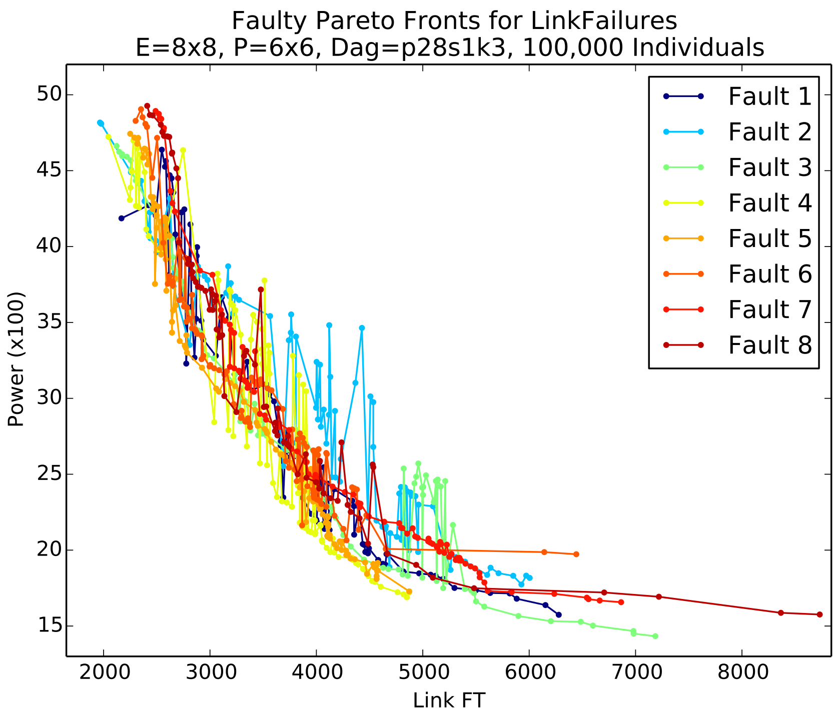
(d) Original Repaired

Figure 7.6 – Multi-objective, Core Fault Recovery, Link fault tolerance with Power, Sparsely Connected Graph

Figure 7.7 shows the results for a series of link faults impacting a many-core system using mappings optimized for the multiple objectives of link fault tolerance and network power.

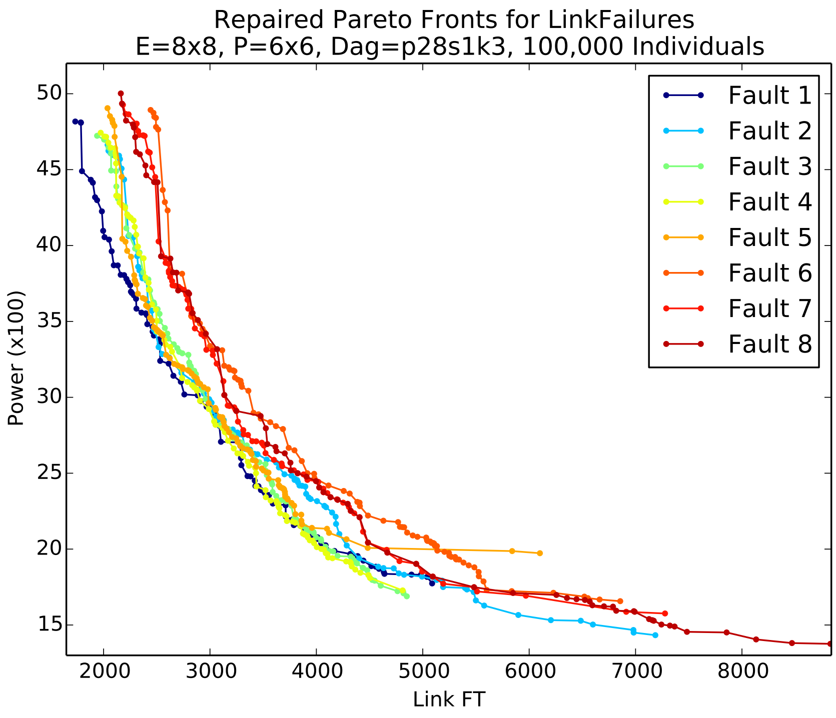


(a) Evolved

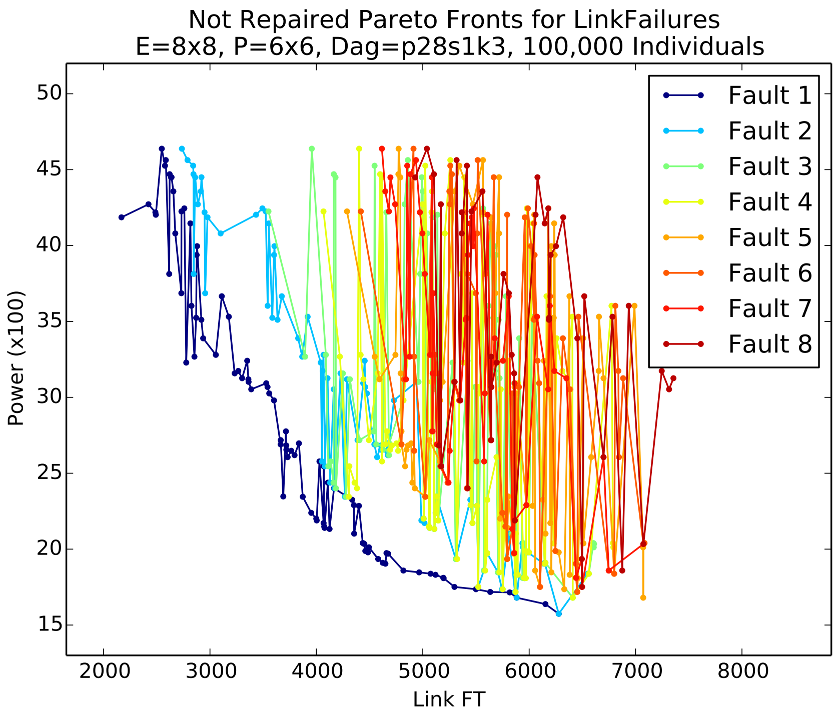


(b) Fault Added

This experiment ended after only nine link faults at which point the evolutionary algorithm was unable to find any viable mappings. These results look similar to the previous set of results, where the fault has very different effects on the network power value of neighbouring points on the Pareto front. A faulty link can have a severe effect of the length of paths



(c) Repaired



(d) Original Repaired

Figure 7.7 – Multi-objective, Link Fault Recovery, Link fault tolerance with Power, Densely Connected Graph

between a ComPair; if the link is a critical link for the ComPair then the fault will result in there being no minimal length paths for the ComPair making the mapping non-viable, if the link is a significant link for the ComPair then the fault can result in another significant link becoming a critical link for the ComPair increasing the value of the objective value.

The fitness values of the link fault tolerance have also been affected by the link faults. Some of the link fault tolerance values appear to have an increase in response to the link faults, although the effect is much less pronounced than for the power metric.

Link Faults applied to Link fault tolerance with Mean Traffic - Figure 7.8
Figure 7.8 shows the results for a series of link faults impacting a many-core system using mappings optimized for the multiple objectives of link fault tolerance and mean excess traffic.

These result resemble closely the previous set of results, although this experiments did not end until the thirteenth fault resulted in the evolutionary algorithm being unable to find any viable mappings.

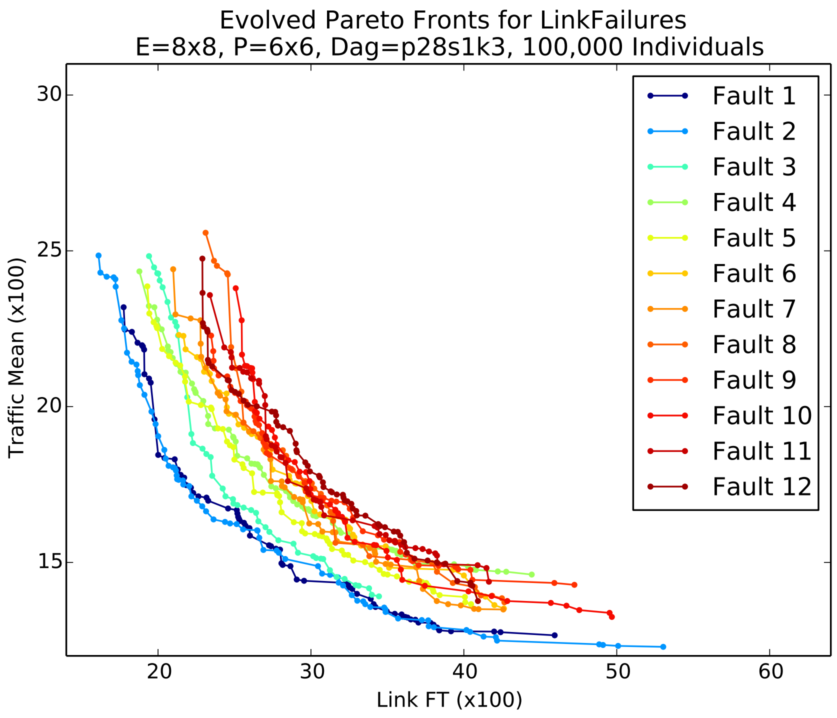
7.5.4 Summary

All the results of these experiments confirmed that the use of the fault-recovery cycle to implement graceful degradation and graceful amelioration is an effective strategy for maintaining mappings that minimize the objective values and that the search for alternative mappings prolong the operational life of a many-core system compared to a static mapping that would quickly become non-viable after a small number of faults.

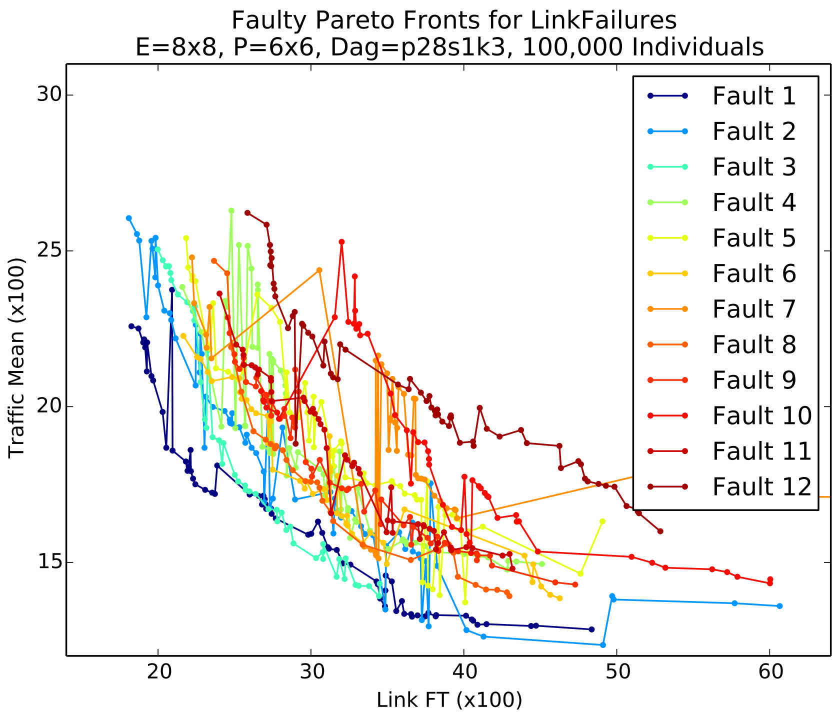
The response of the network power fitness to both core fault and link faults, demonstrates that points that are in close proximity on the Pareto front can have very different underlying properties even when the fitness values are almost identical. This confirms that a Pareto front crowding calculation is not always a good mechanism for determining if individuals are similar and can, therefore, be detrimental to diversity if used for selection.

7.6 Conclusion

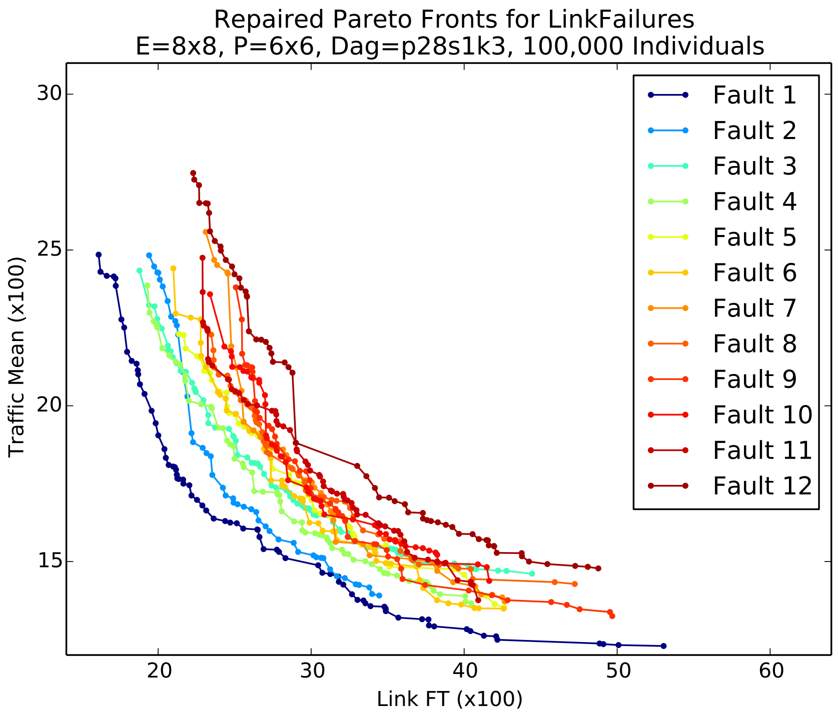
The results of this chapter illustrated how the fitness values of two points that are in close proximity on a Pareto front can react very differently to the same fault. This is a demonstration of how a crowding calculation is not necessarily a good indicator of how similar two solutions are. These results supports the idea that care should be taken when using crowding distance by understanding the relationship between the crowding distance and the underlying qualitative properties of the individuals.



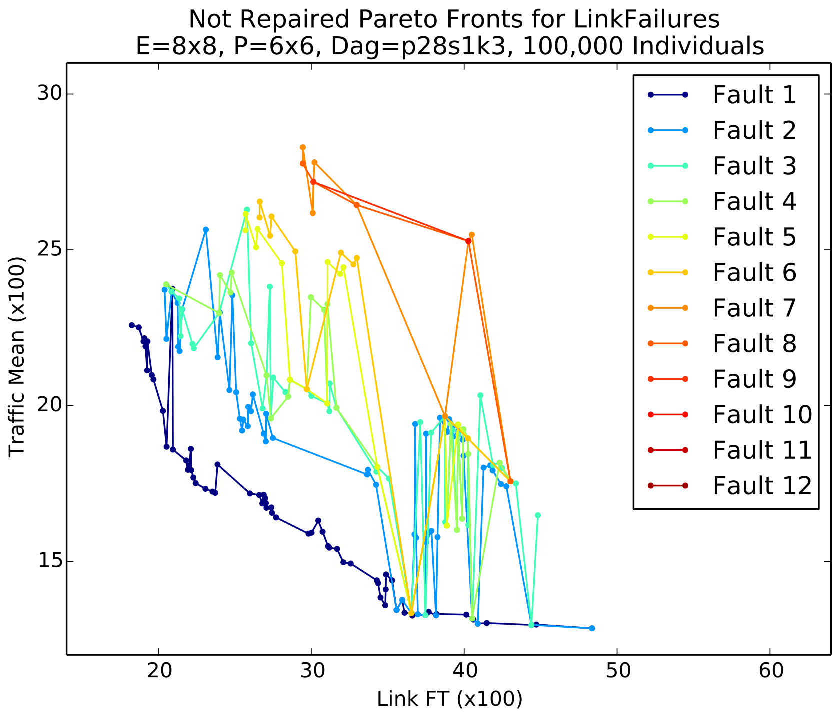
(a) Evolved



(b) Fault Added



(c) Repaired



(d) Original Repaired

Figure 7.8 – Multi-objective, Link Fault Recovery, Link fault tolerance with Mean Traffic, Densely Connected Graph

Chapter 8

Conclusions and Future Work

The every decreasing feature size of semiconductor devices provide the opportunity to design single chip NoC devices with hundreds or thousands of processing cores. The same reduction in feature size is accompanied by greater variability in the characteristics of transistors leading to lower reliability. In addition dark silicon is required for devices with a feature size of 22nm. The use of the run-time fault tolerance strategies presented in this thesis can take advantage of the large number of cores in a many-core system to protect against hardware faults and manage dark silicon.

To fully utilise the flexibility of being able to choose the core that hosts a process of a application process graph, this thesis has developed an implementation of an evolutionary algorithm that uses knowledge of the status of the many-core hardware to search for a set of optimal process mappings.

This work has developed four objectives that can be used effectively together in a multi-objective search of the solution space. These include two examples of fault tolerance (core fault tolerance and link fault tolerance) that are designed to produce mappings that are robust in the event of failure in a processing core or a communications link. Such mappings ensure that the effect of a failure on the functioning of the many-core system is minimized and that the system can recover to a fully functioning condition, possibly with some degradation of performance, with minimal reconfiguration.

The other two objectives (network power and excess traffic) are examples of properties that can be used to manage the run-time environment, illustrating the generality and extensibility of the approach. The network power objective searches for mappings that minimize the overall consumption of power used to route data between pairs of communicating cores while the excess traffic objective aims to reduce the overall amount of communication traffic within the many-core communications array.

The four objectives developed in this thesis are examples of objectives that can be used to evaluate mappings. Many other objectives are possible, for example the control of frequency and voltage to reduce the power consumption of individual cores.

The design philosophy of the evolutionary algorithm has been to produce a multi-objective search mechanism that can be used together with any collection of objectives that a designer judges to be appropriate to the specific problem being solved. Each objective requires code to calculate the underlying metric, which may require further manipulation to produce the objective value used to compare mappings. Once the code is available to evaluate the selected objectives, the evolutionary algorithm can use any collection of objectives to search the solution space for a set of optimal solutions.

The evolutionary algorithm proved successful in finding good solutions to multi-objective problems within a defined computational budget.

Having developed a suitable evolutionary algorithm a Monitor process was developed to implement graceful degradation and amelioration through a fault/recovery life cycle, consisting of the following steps:

- Normal operation
- Fault detection
- Graceful degradation
- Graceful Amelioration
- Return to normal operation

The experimental results show that the cycle is effective in returning the system to a level of operation with a performance that is close to the pre-fault performance and prolong the operational life of a many-core system compared to a static mapping that would quickly become non-viable after a small number of faults.

All the results of these experiments confirmed that the use of the fault-recovery cycle to implement graceful degradation and graceful amelioration is an effective strategy for maintaining mappings that minimize the objective values and that the search for alternative mappings prolong the operational life of a many-core system compared to a static mapping that would quickly become non-viable after a small number of faults.

The fault-recovery cycle running in real-time can continue until depletion of the hardware resources result in the system being unable to accommodate the application due to the lack of functional cores, or the failure of the evolutionary algorithm to discover any viable mappings, a behaviour that was observed in the experiment results. Failure to find viable mappings, is the result of there being too few remaining links in the communications array to allow communication between all ComPairs or there being so few viable solutions that they become extremely difficult to find.

The experiments also tracked the effect of repeated failures on the mappings in the original pre-fault solution set. The results show that the objective values of the original mappings quickly become significantly worse than those of the new mappings.

The results of the experiments confirm that the use of graceful degradation and amelioration, and a Monitor to manage the recovery cycle can maintain the performance at a level that would not otherwise be possible.

The results of Chapter 7 illustrated how the fitness values of two points that are in close proximity on a Pareto front can react very differently to the same fault. This is a demonstration of how a crowding calculation is not necessarily a good indicator of how similar two solutions are. These results supports the idea that care should be taken when using crowding distance by understanding the relationship between the crowding distance and the underlying qualitative properties of the individuals.

Graceful amelioration can be applied to conditions other than hardware faults, for example the detection of hot-spots and the effects of thermal ageing. The cores in a hot-spot can be registered in the hardware map as faulty and then the evolutionary algorithm run to find new mappings that will not use the hot-stop cores. Once the temperature of hot-spot cores have return to an acceptable level the cores can be reinstated by changing their status to idle in the hardware map. Similarly, graceful amelioration can be used to balance the use of the cores by migrating processes from heavily used cores to low usage cores. The thermal ageing effect on heavily used cores will shorten their life compared to low usage cores, so balancing core usage can avoid the early failure of heavily used cores resulting in an extended live of the many-core system as a whole.

8.1 Future Work

The experiments used to determine the level of correlation between objective pairs show that there were mappings which were optimal for core fault tolerance which were also optimal for at least one of the other three objectives. This led to the speculative idea that a multi-objective problem using all four objectives could be replaced by a single objective problem of finding a set of optimal mappings for core fault tolerance and then using these mappings, with the position of the idle cores fixed, for a multi-objective search using the remaining three objectives. This staged approach is worthy of investigation to determine how it compares with a four-objective problem in terms of quality of mappings and computational budget.

This research was carried out in the expectation that it would be implemented on the many-core array developed by the Graceful project, making this the next step for the work completed in this thesis.

The target system for the work in this thesis is a many-core system of hundreds of cores. To achieve this requires the integration of the software with a Monitor process and implementation of a suitable SoC or embedded hardware. Integration with a Monitor, that collects actual traffic data and updates the information contained in the application process graph, will increase the accuracy of the calculation of the metrics and objectives.

The Monitor developed in this thesis has been been designed to demonstrate the validity of the fault-recovery cycle. The Monitor needs to be extended to implement multiple regions of array cores, each with a dedicated Monitor. Monitor processes will be required to can exchange information regarding the status of hardware and processing cores forming a boundary between regions, communication of data between ComPairs that straddle multiple regions, and processes to mange the migration of tasks between regions.

The model can be extended in a number of ways to increase the overall flexibility and applicability of the model:

- Map multiple applications simultaneously, which can be achieved by including a set of graphs for multiple applications in the data structure used to represent the application process graph.
- Allow multiple tasks to be mapped to a single processing node, an approach used by the majority of contemporary research.
- Relax the requirement for using only application process graphs represented by a directed acyclic graph so that applications with bi-directional communication between processes can be included.

Graceful amelioration can also be applied to conditions other than hardware faults, for example the detection of hot-spots and the effects of thermal ageing. The cores in a hot-spot can be registered in the hardware map as faulty and then the evolutionary algorithm run to find new mappings that will not use the hot-stop cores. Once the temperature of hot-spot cores have returned to an acceptable level the cores can be reinstated by changing their status to idle in the hardware map. Similarly, graceful amelioration can be used to balance the use of the cores by migrating processes from heavily used cores to low usage cores. The thermal ageing effect on heavily used cores will shorten their life compared to low usage cores, so balancing core usage can avoid the early failure of heavily used cores resulting in an extended live of the many-core system as a whole.

Appendix A

Graph Generator

Experiments require a selection of representative application process graphs which have been generated using a simple graph generator that produces graphs with the desired characteristics[190].

The following parameters are used to control the behaviour of the graph generator:

Number of Nodes
The total number of nodes required in the graph.

Single Source and Sink Nodes
A graph with a single source and single sink node can be specified. Graphs with single source and sink nodes are used in the early single objective experiments. Graphs with multiple sources and sinks are used in later experiments. If a single source and single sink are not specified then the number of source and sink nodes is random which means the that graph generator does not guarantee a specific number of processing nodes.

Gaussian Distribution Parameters
The number of outbound edges from nodes and the length of outbound edges in terms of the number of ranks that they span is determined by a random Gaussian distribution to ensure that there are a variety of numbers of inbound edges, outbound edges and path lengths.

The graph generator also ensures that the nodes are numbered one rank at a time from left to right for the aesthetic quality of making the graph visually easier to read, but otherwise has no affect on the mapping process from the application process graph to the many-core array.

Following a review of available description languages for graphs, the DOT Graph Description Language was chosen for its simplicity and available software [191], [192]. The DOT language is supported by GraphViz which proved to be able to provide a clear, uncluttered visual representation of the graphs. Utilities called DOT and NEATO provide the means for producing a pdf, eps or png file of the visual representation of a graph using standard free tools. Additionally, the DOT language allows groups of nodes to be defined as having the same rank and provides functionality for controlling the size, shape, colour and text of nodes.

The output from the graph generator consists of both a ".gv" file in "dot" format and an xml file with tags designed specifically for the thesis. A unix shell file is also produced to convert the generated .gv into pdf format as well as a file containing LaTeX commands for inclusion of the pdf file into documents.

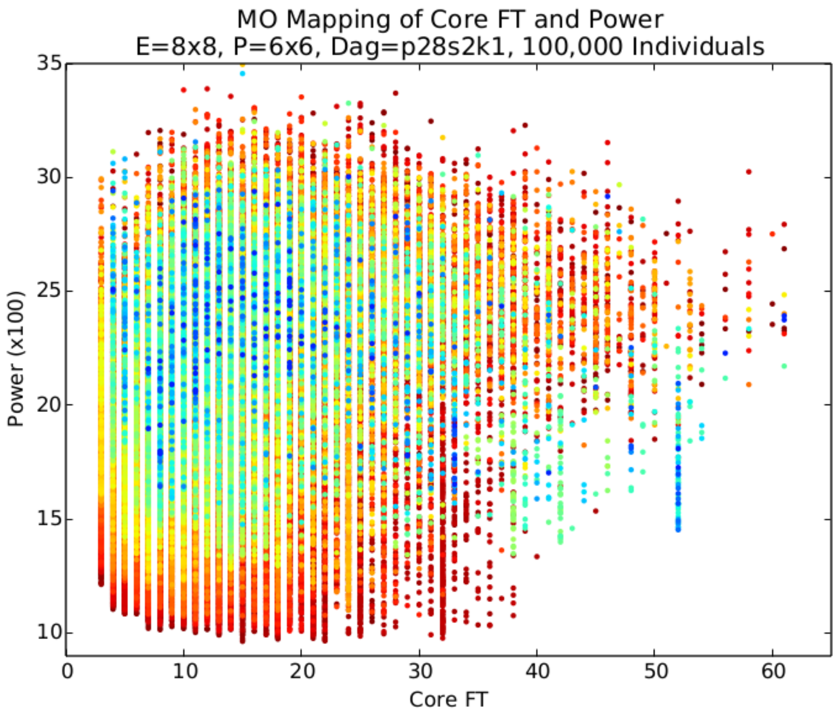
Appendix B

Multi-Objective Plots for Sparsely and Moderately Connect Graphs

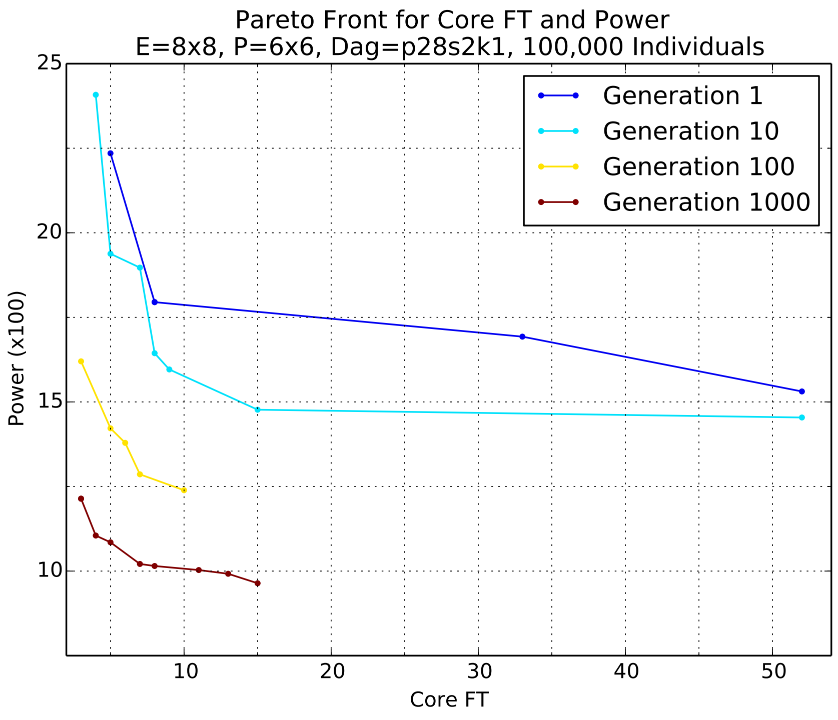
The sections of this appendix present the plots produced by running the experiments from Section 7.5 with the, sparsely connected graph p28s2k1, and the moderately connected graph p28s3k2.

Both of these graphs have fewer ComPairs which result in lower metrics values for link fault tolerance, network power and excess traffic metrics. This shows as a larger concentration of lower value points. Other than these lower value points the sparsely and moderately connected graphs share the same characteristics as the graphs in Section 7.5.

B.1 Results for a Sparsely Connected Graphs

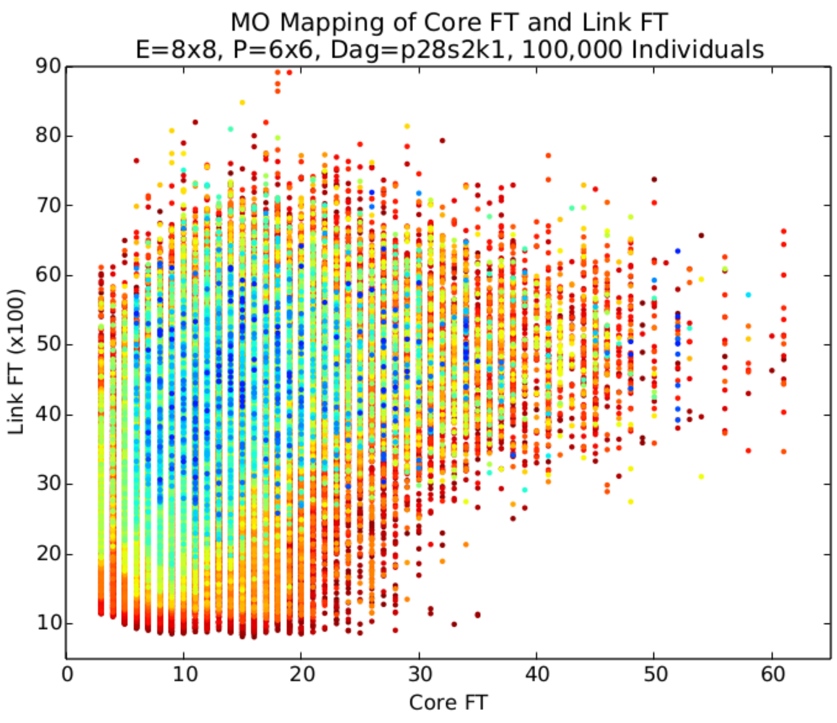


(a) Individuals

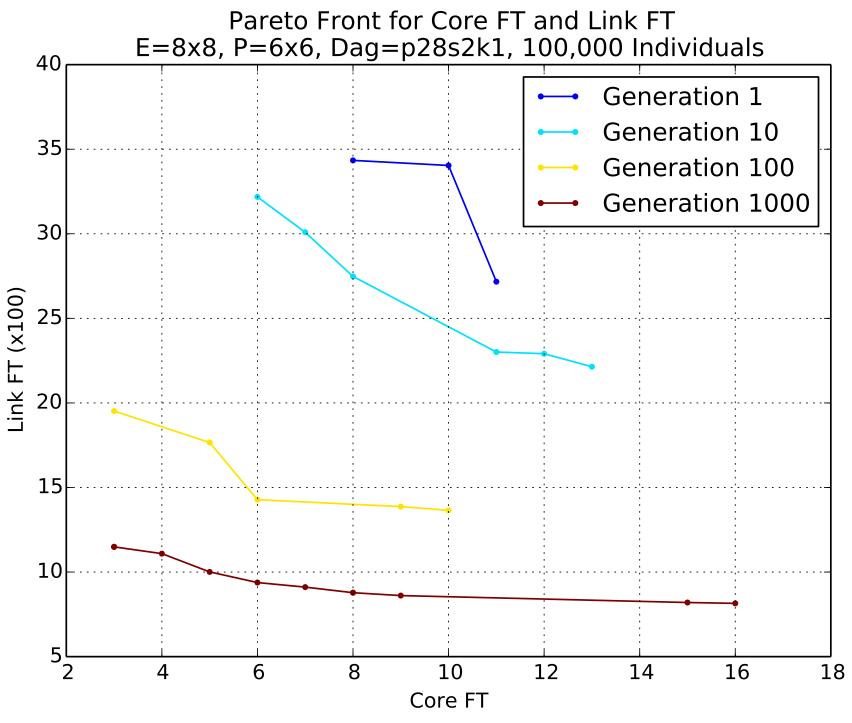


(b) Pareto Front Evolution

Figure B.1 – MO Core & Power for Sparsely Connected Graph

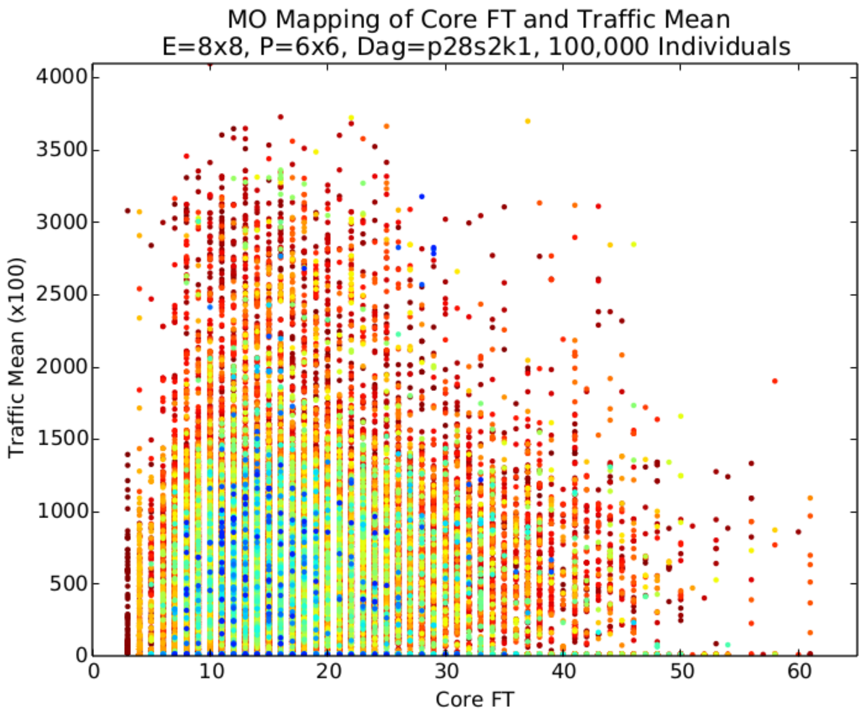


(a) Individuals

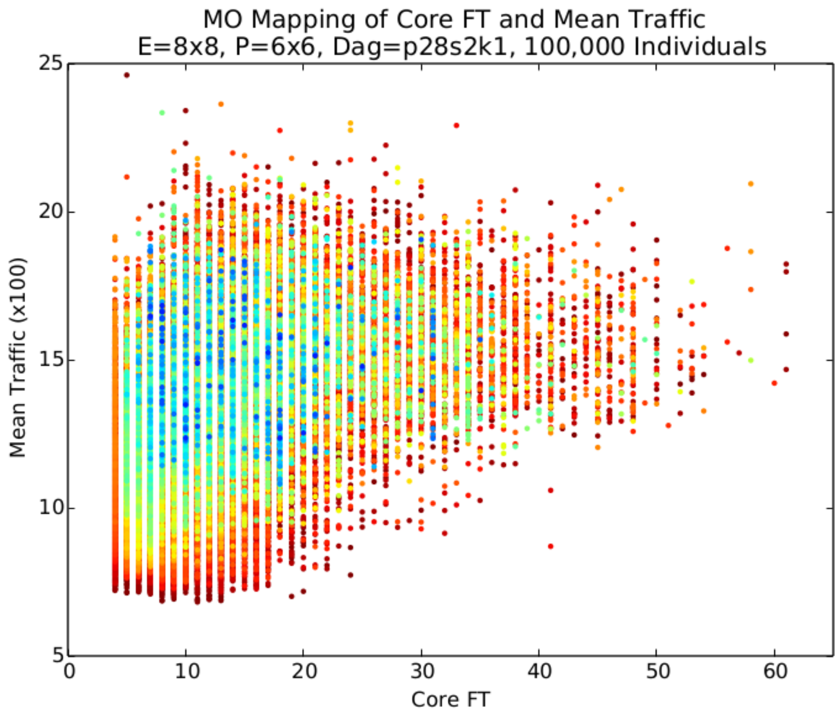


(b) Pareto Front Evolution

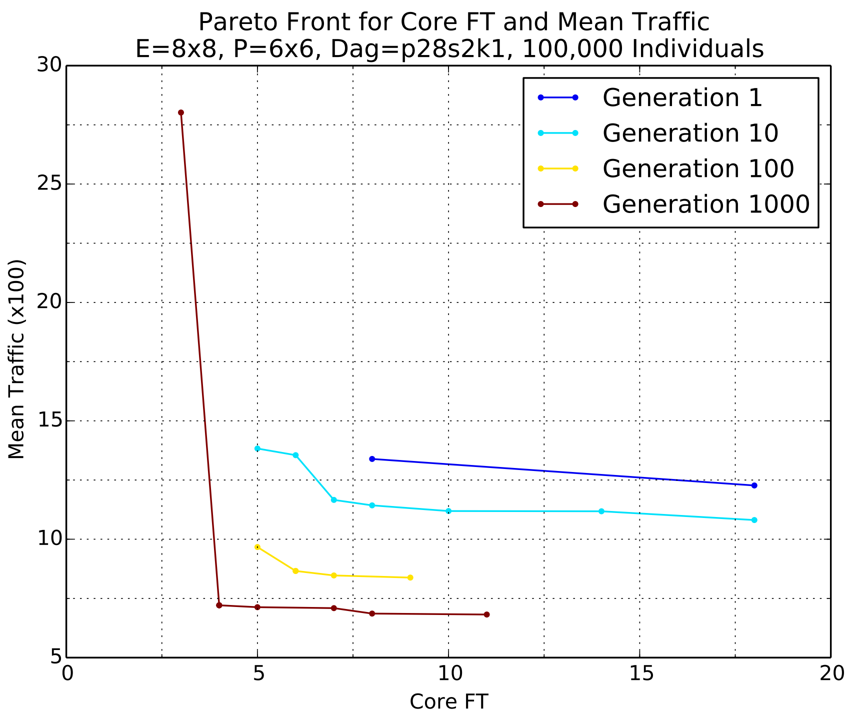
Figure B.2 – MO Core & Link for Sparsely Connected Graph



(a) Individuals

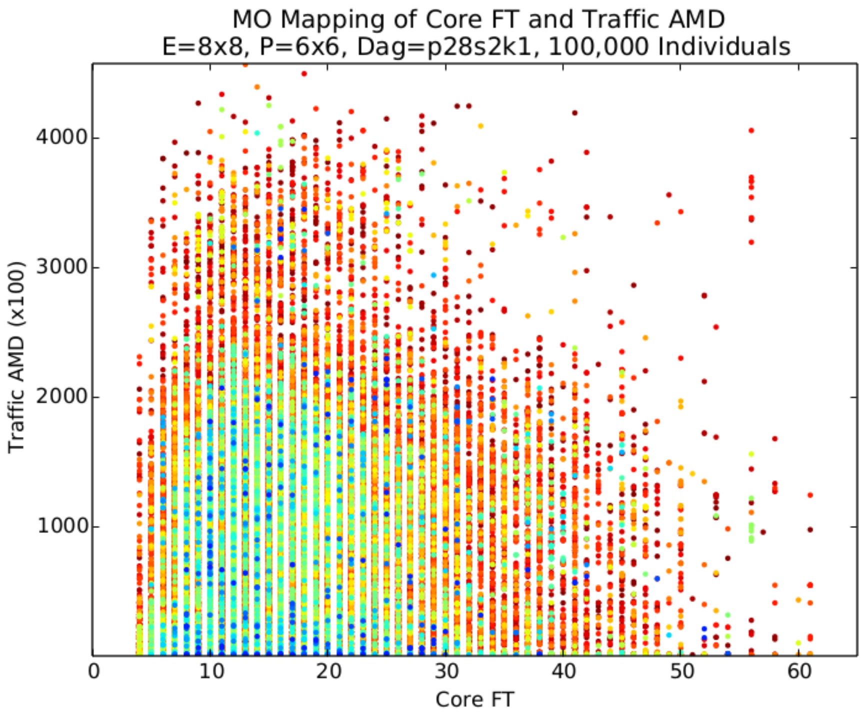


(b) Individuals

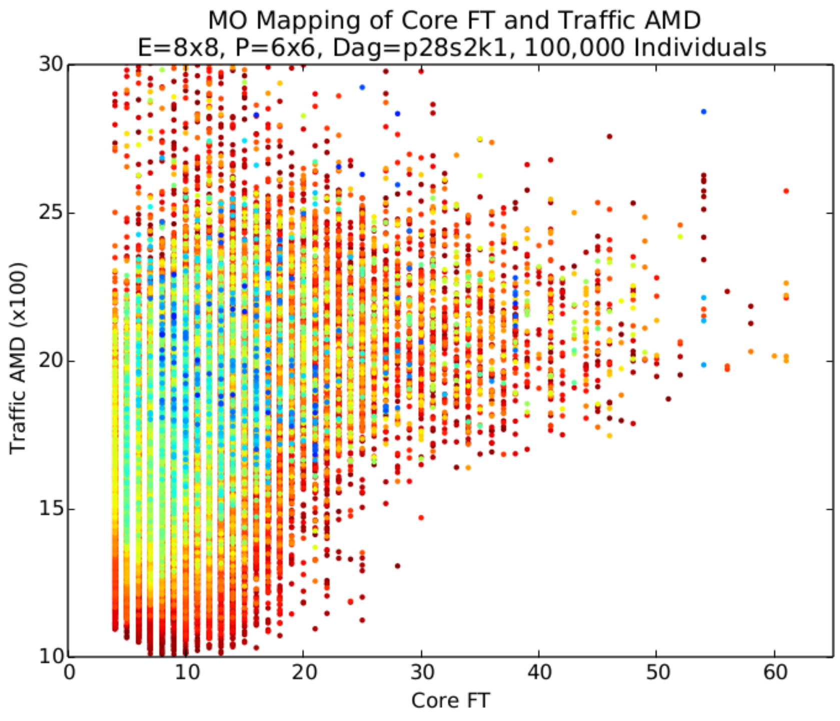


(c) Pareto Front Evolution

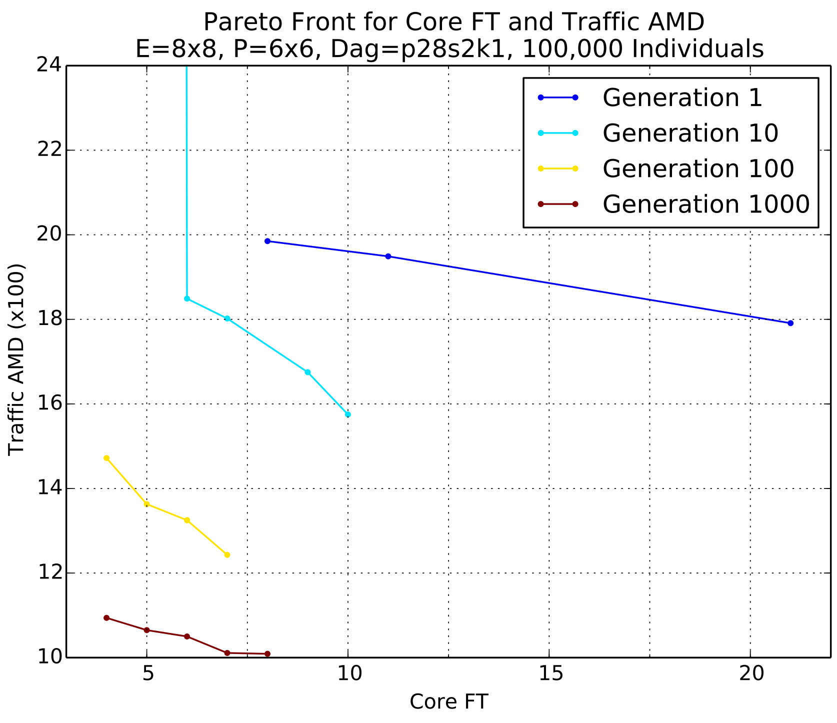
Figure B.3 – MO Core & Mean Traffic for Sparsely Connected Graph



(a) Individuals

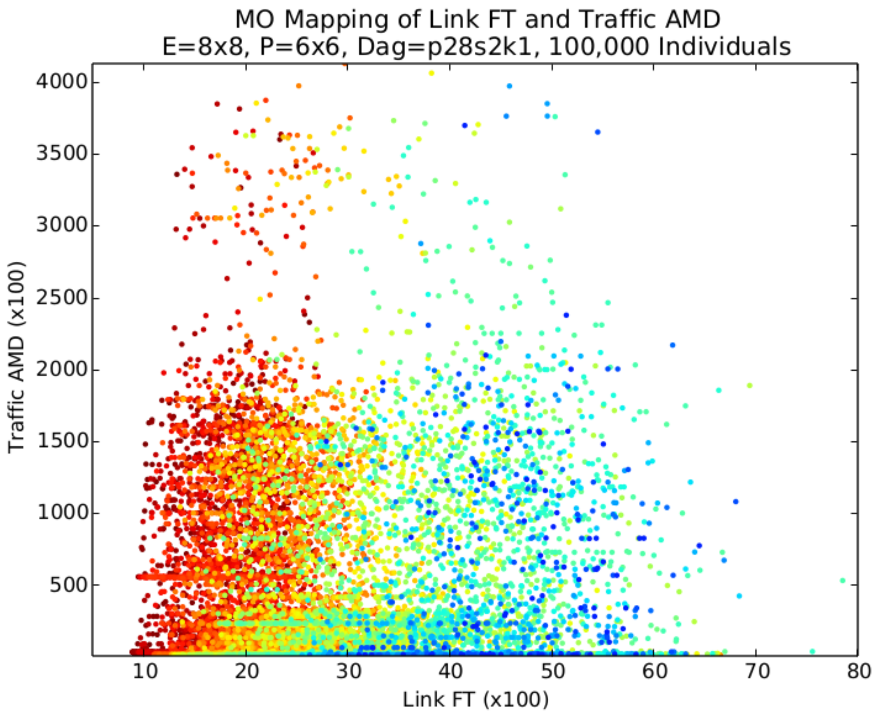


(b) Individuals

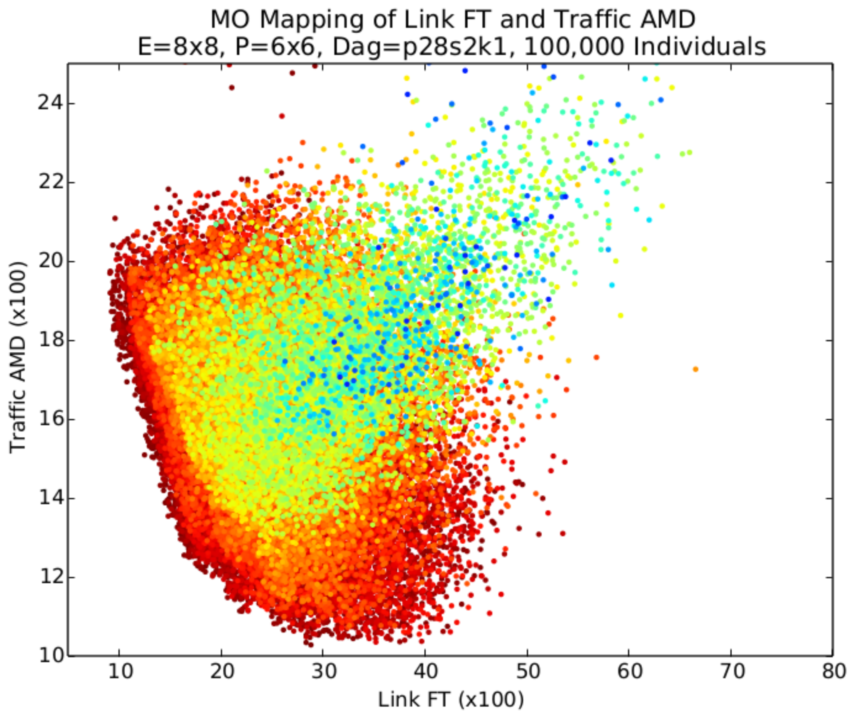


(c) Pareto Front Evolution

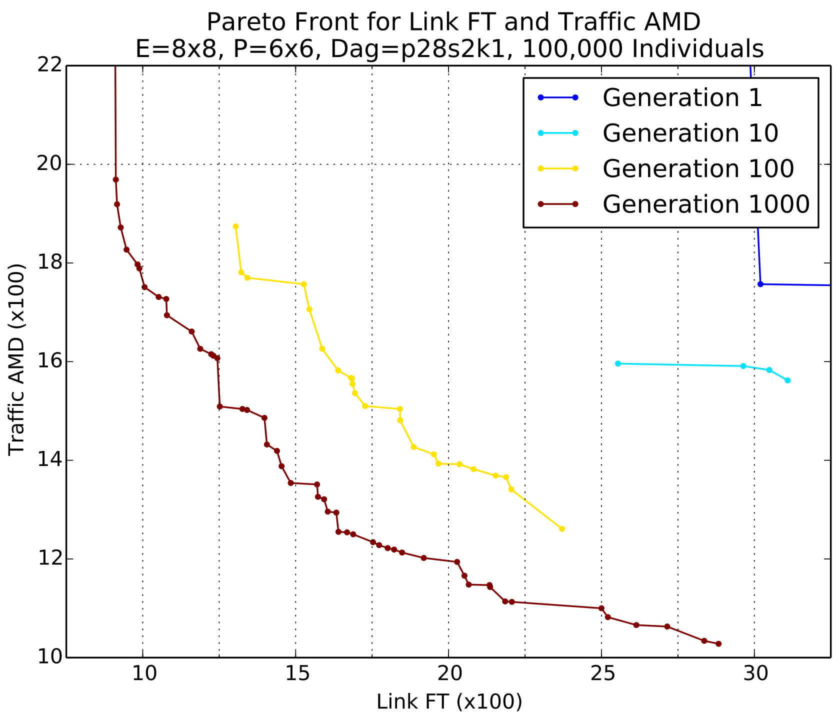
Figure B.4 – MO Core & Traffic AMD for Sparsely Connected Graph



(a) Individuals

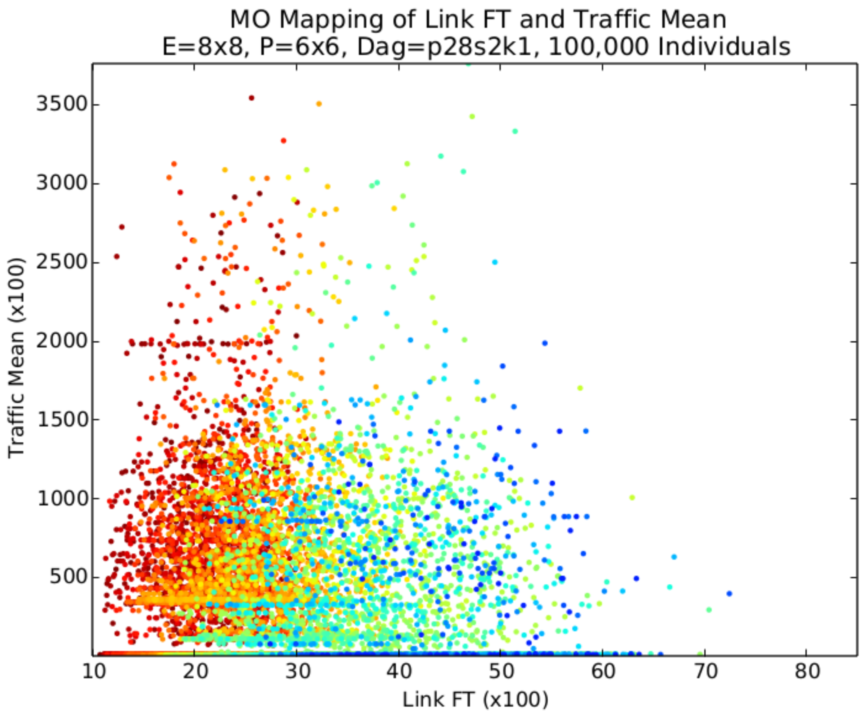


(b) Individuals

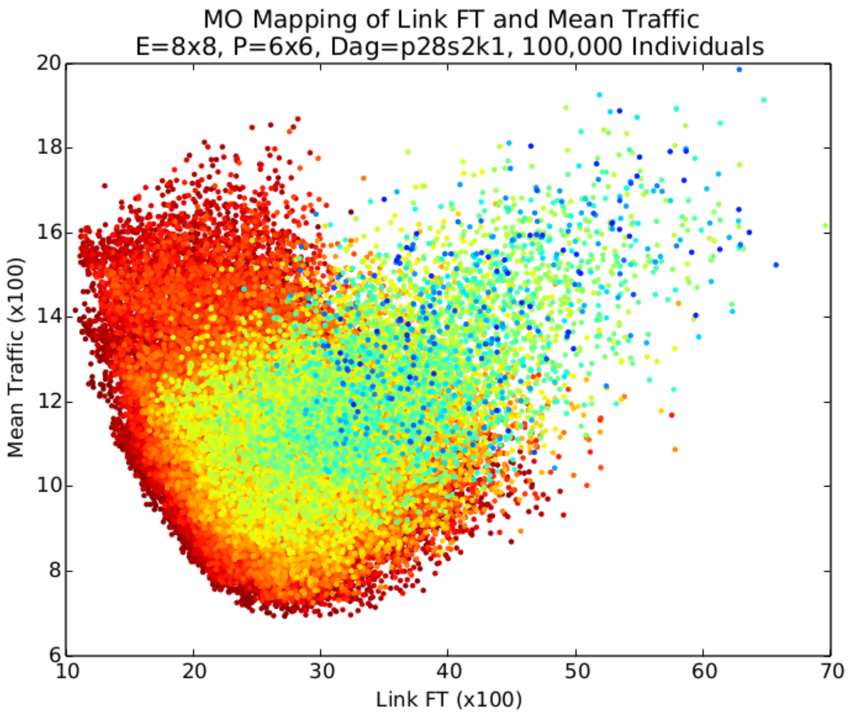


(c) Pareto Front Evolution

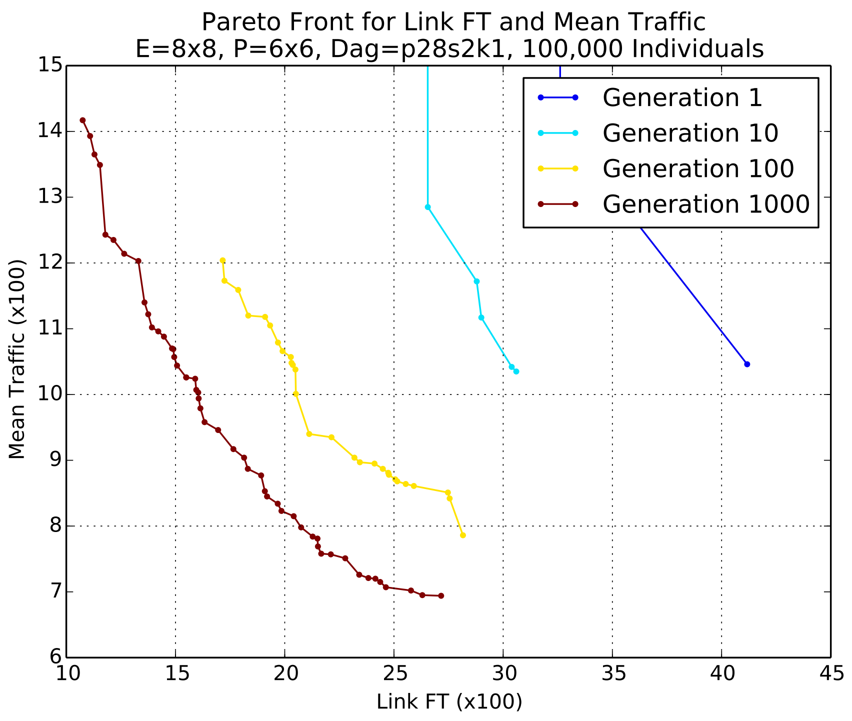
Figure B.5 – MO Link & Traffic AMD for Sparsely Connected Graph



(a) Individuals

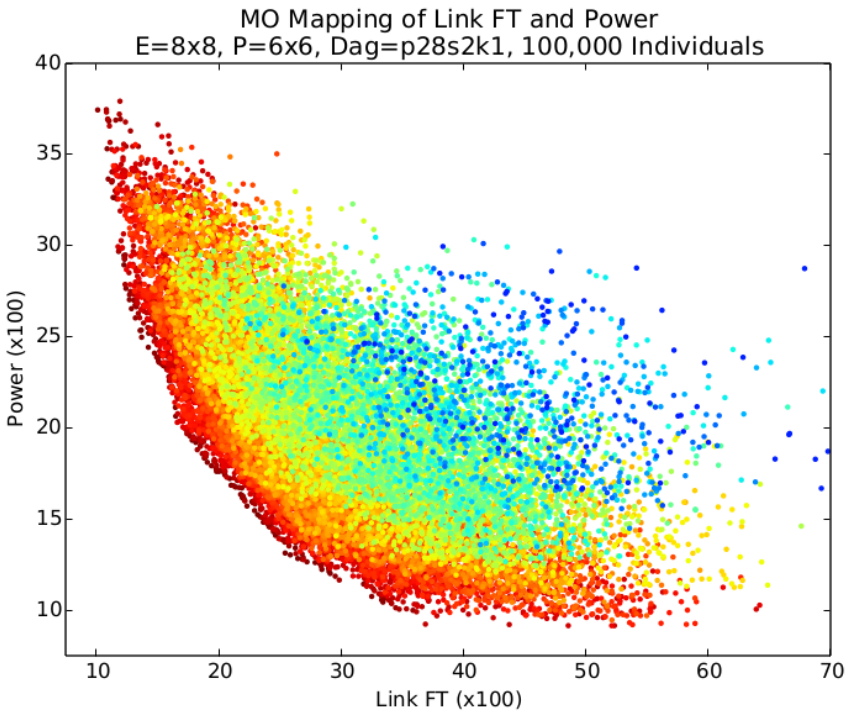


(b) Individuals

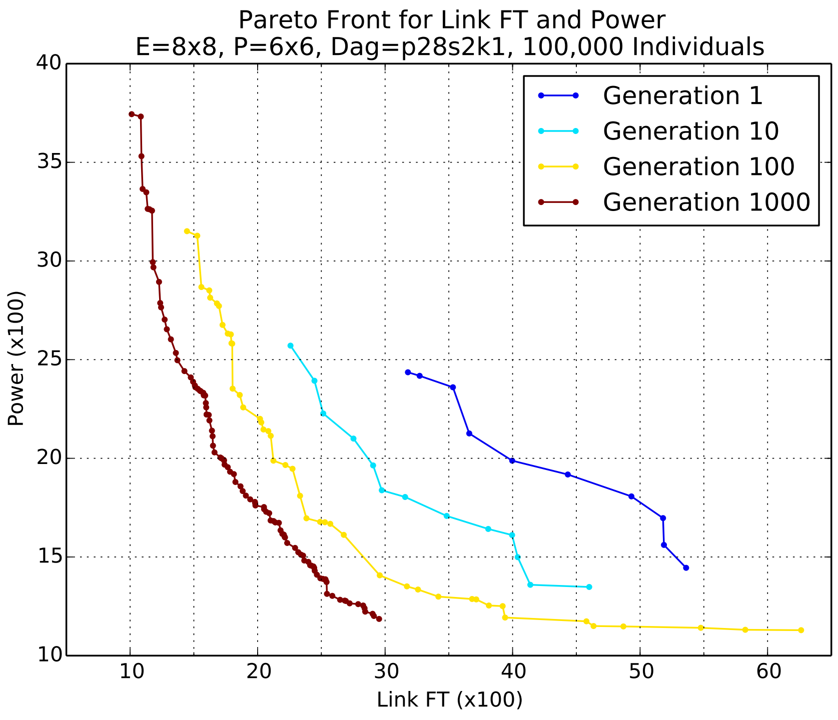


(c) Pareto Front Evolution

Figure B.6 – MO Link & Mean Traffic for Sparsely Connected Graph

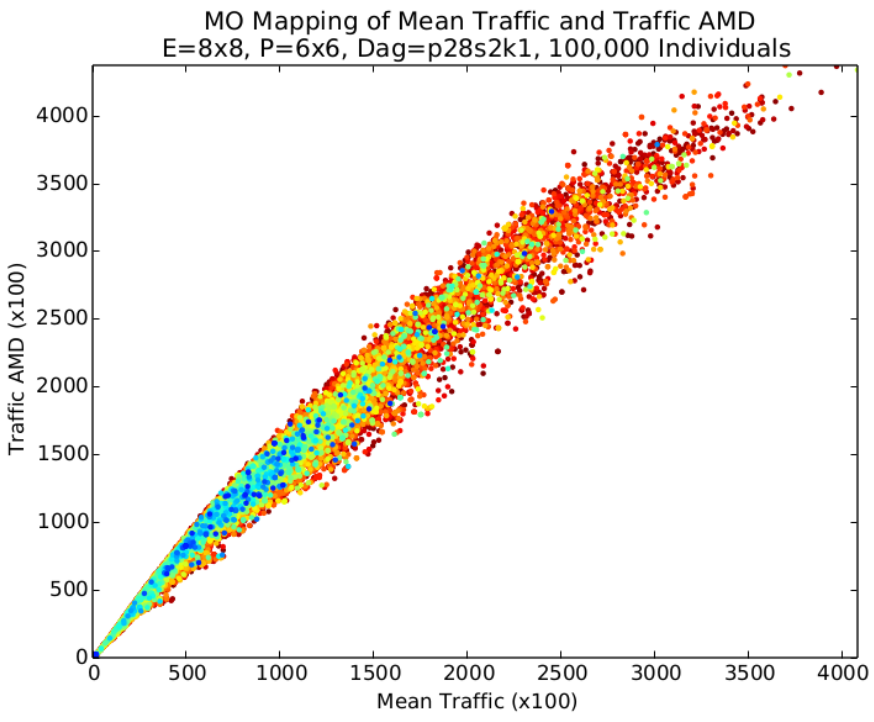


(a) Individuals

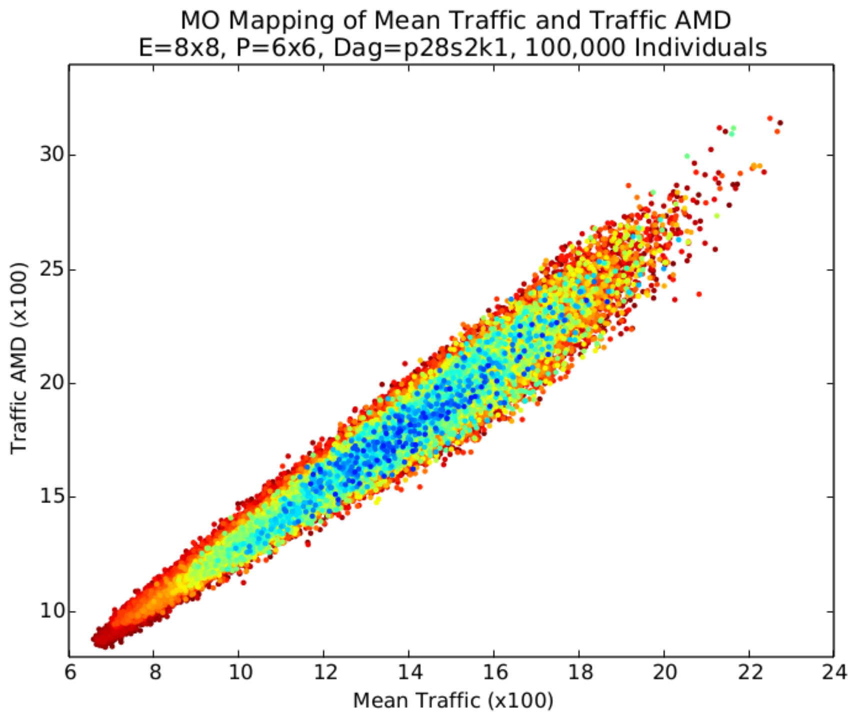


(b) Pareto Front Evolution

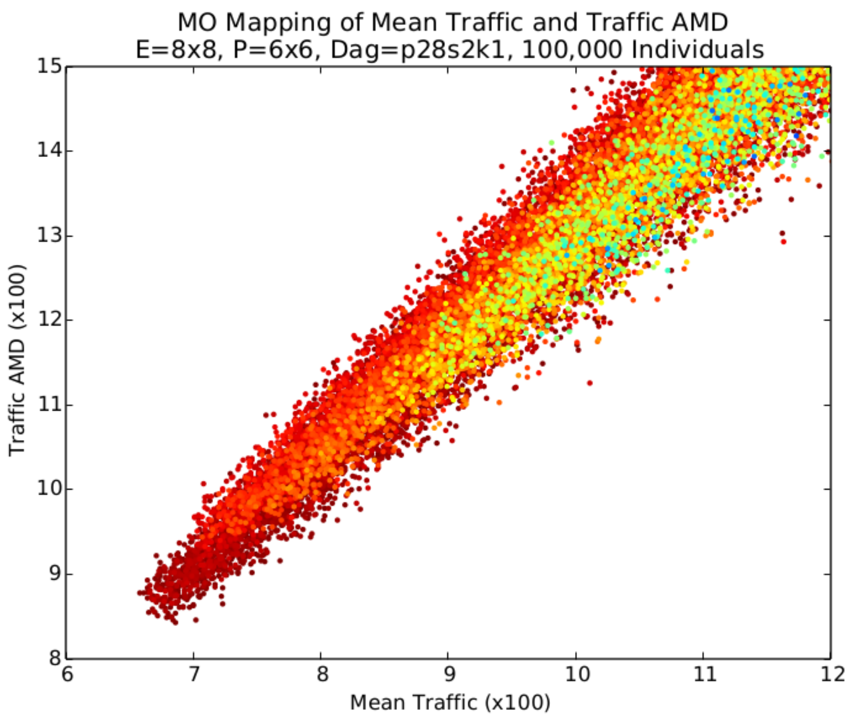
Figure B.7 – MO Link & Power for Sparsely Connected Graph



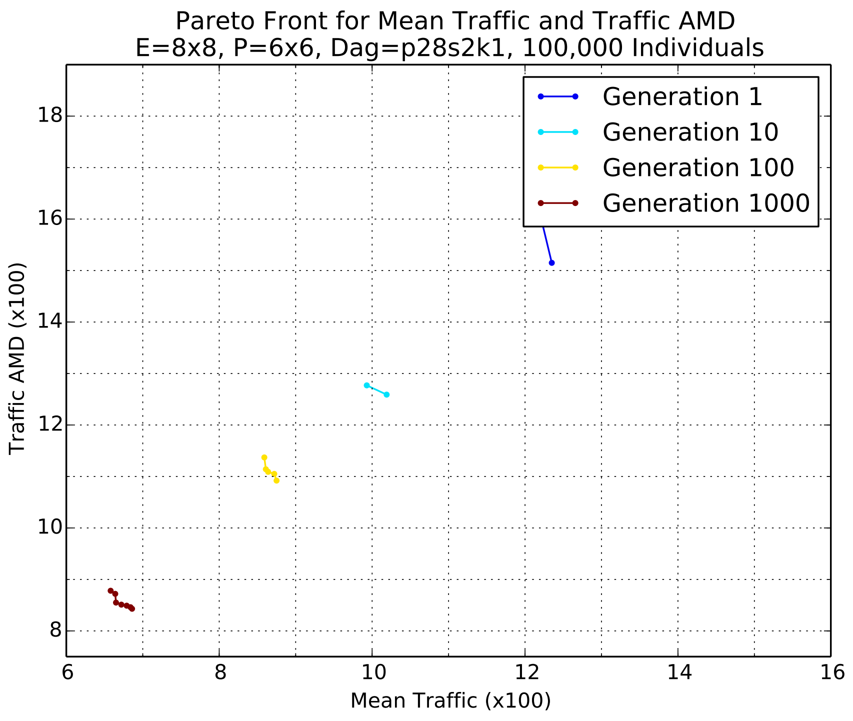
(a) Individuals



(b) Individuals

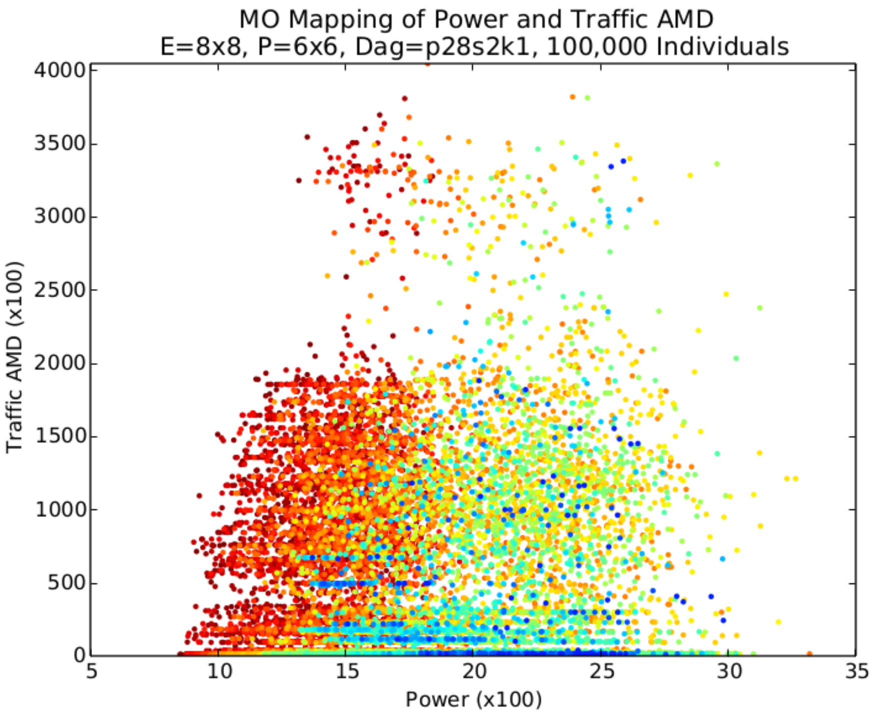


(c) Individuals

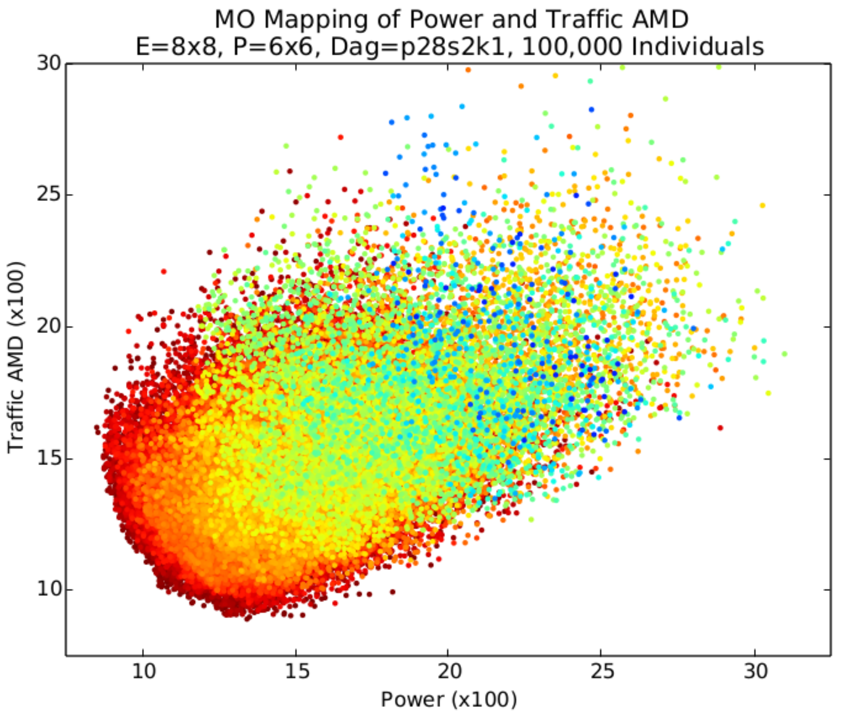


(d) Pareto Front Evolution

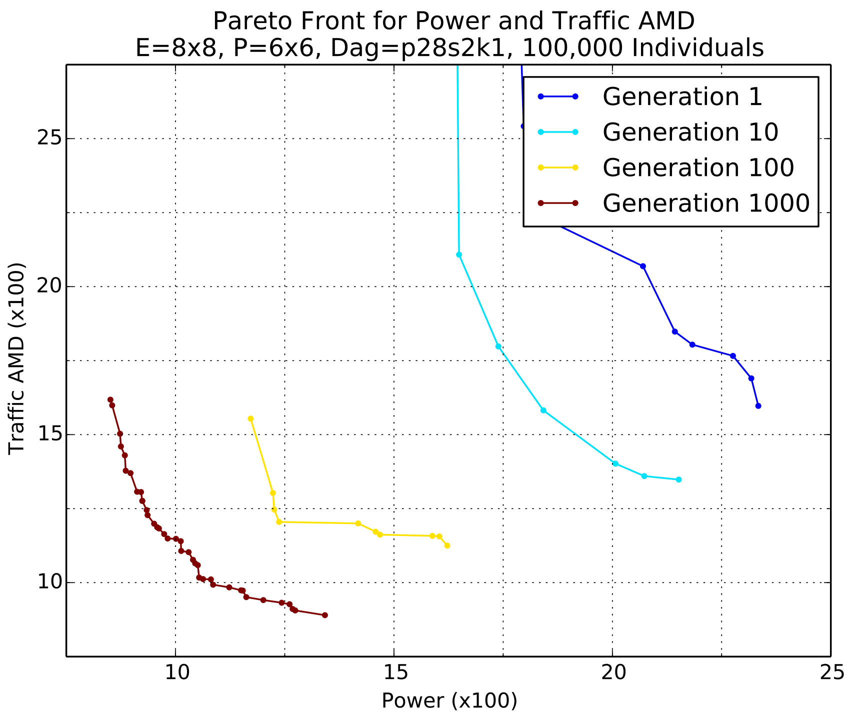
Figure B.8 – MO Mean & Traffic AMD for Sparsely Connected Graph



(a) Individuals

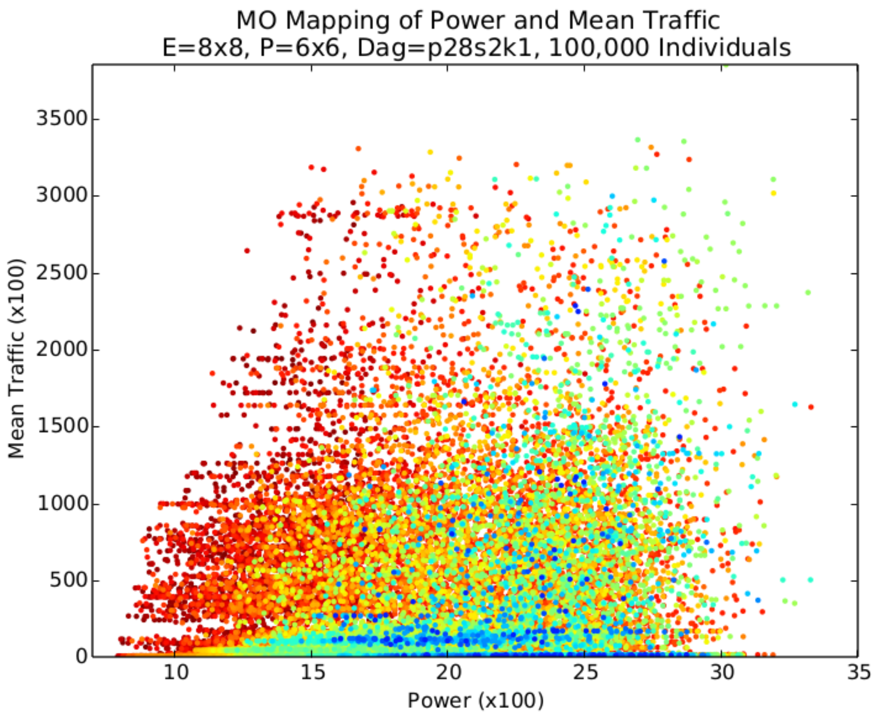


(b) Individuals

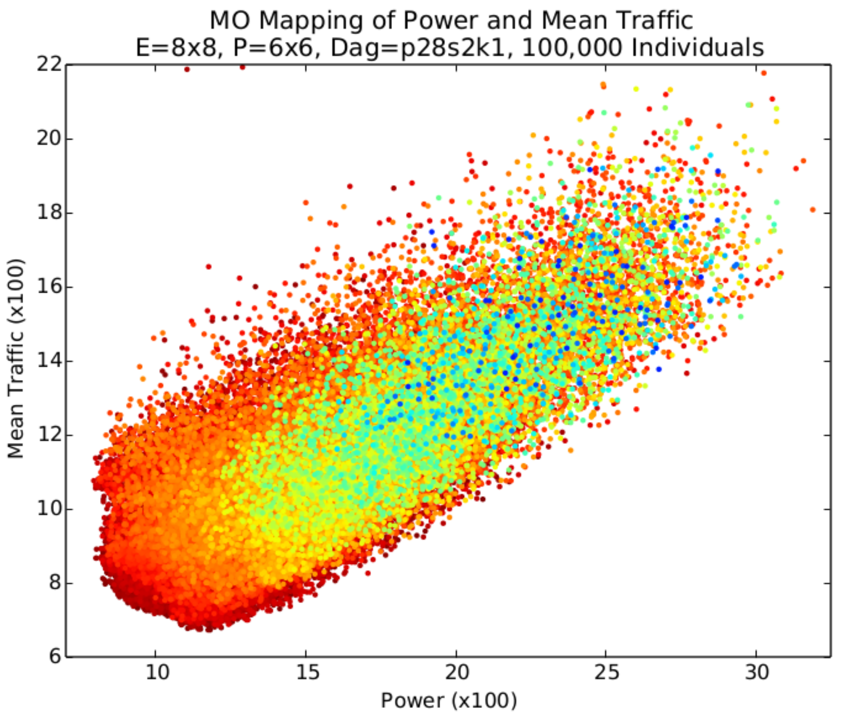


(c) Pareto Front Evolution

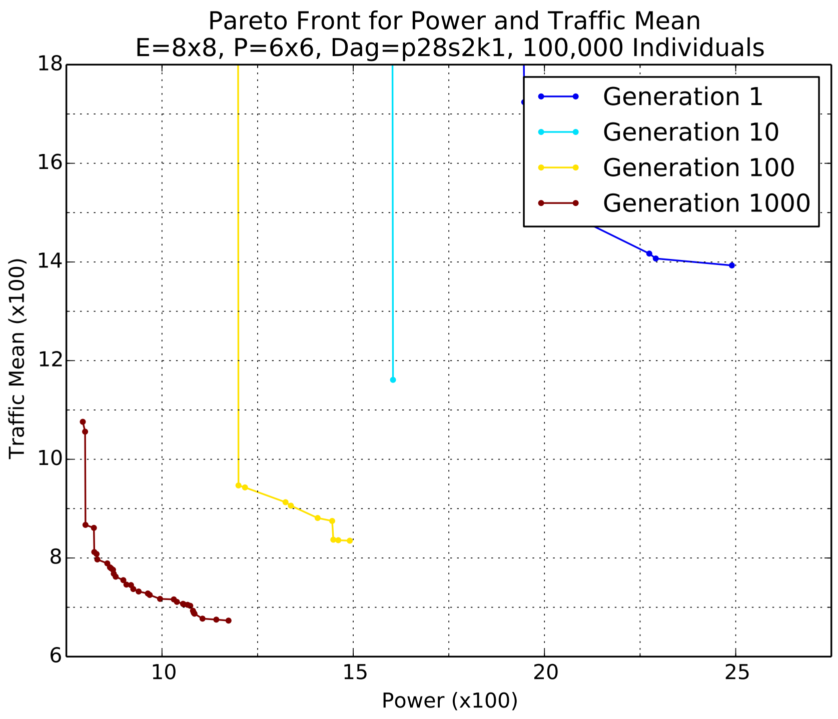
Figure B.9 – MO Power & Traffic AMD for Sparsely Connected Graph



(a) Individuals



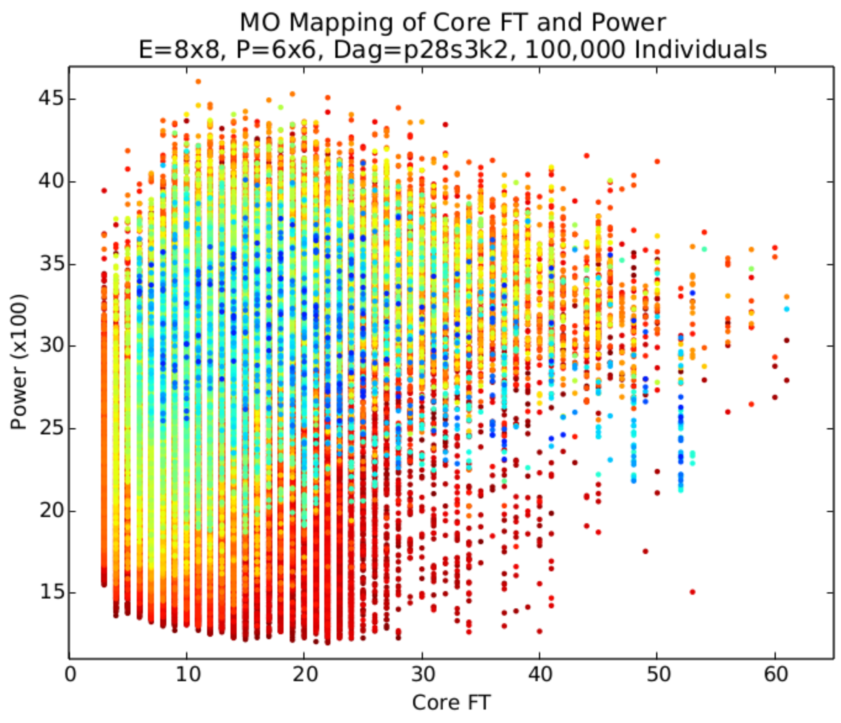
(b) Individuals



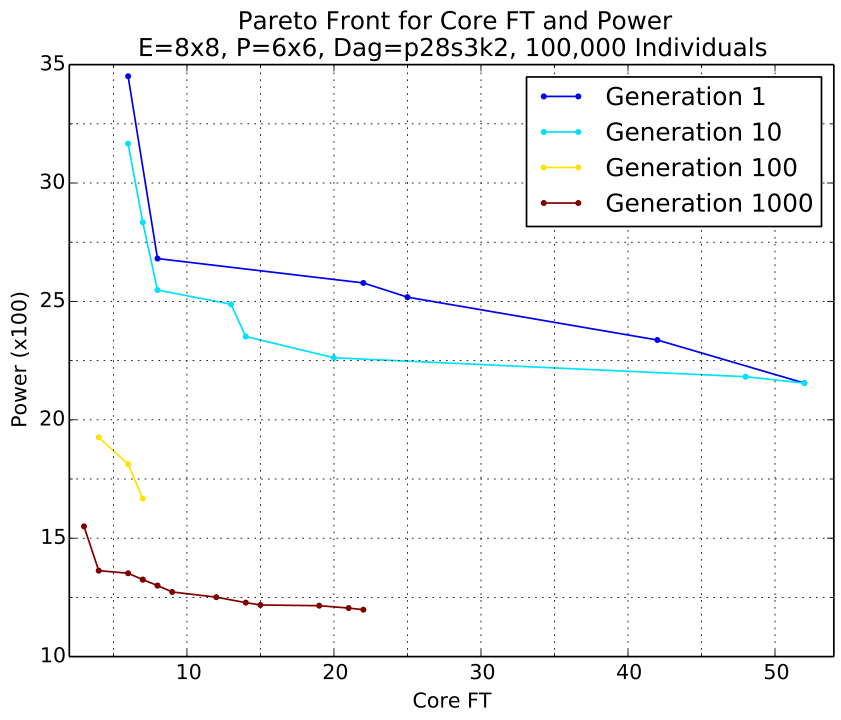
(c) Pareto Front Evolution

Figure B.10 – MO Power & Mean Traffic for Sparsely Connected Graph

B.2 Results for a Moderately Connected Graphs

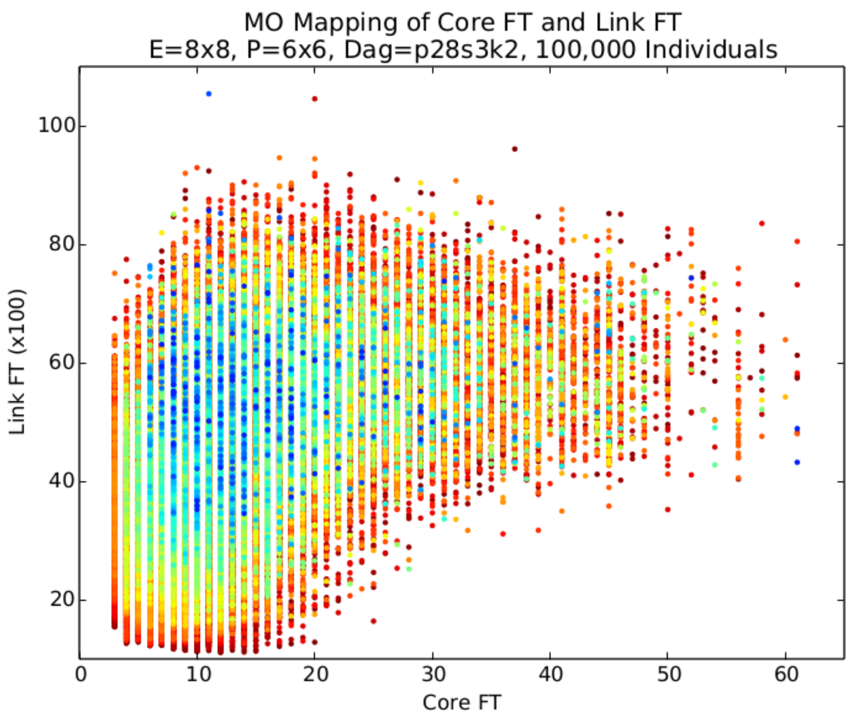


(a) Individuals

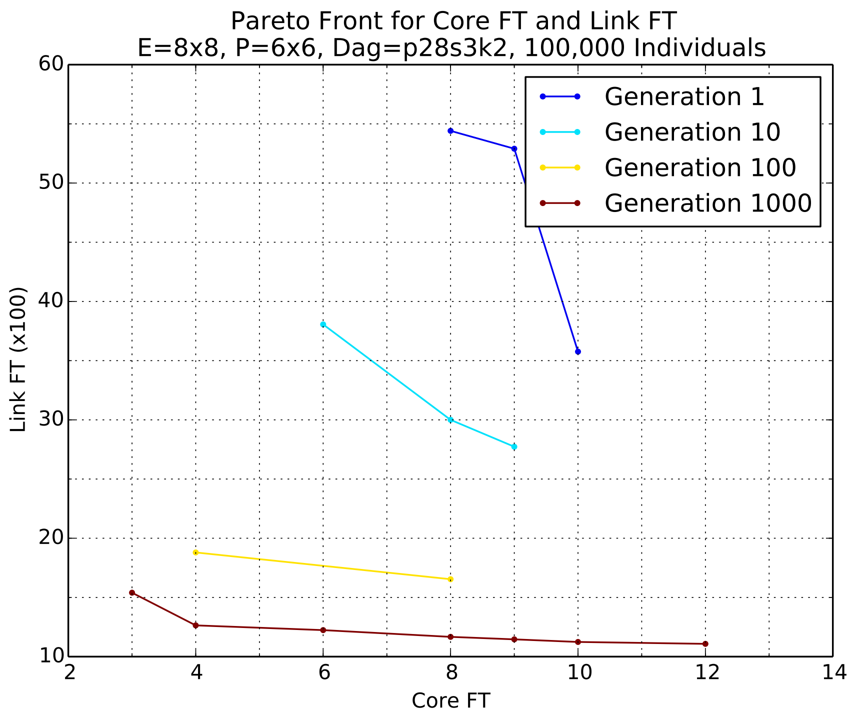


(b) Pareto Front Evolution

Figure B.11 – MO Core & Power for Moderately Connected Graph

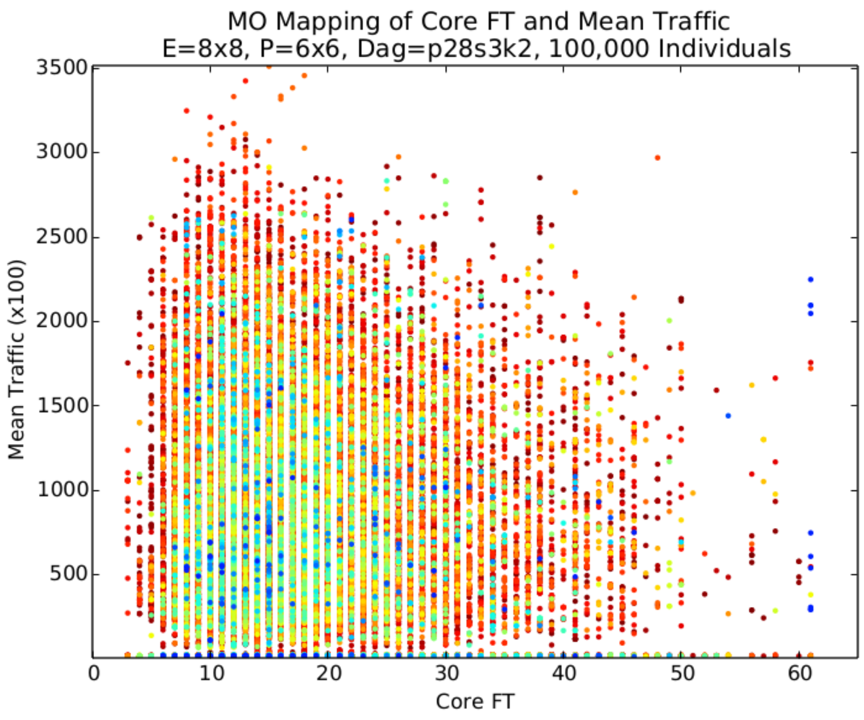


(a) Individuals

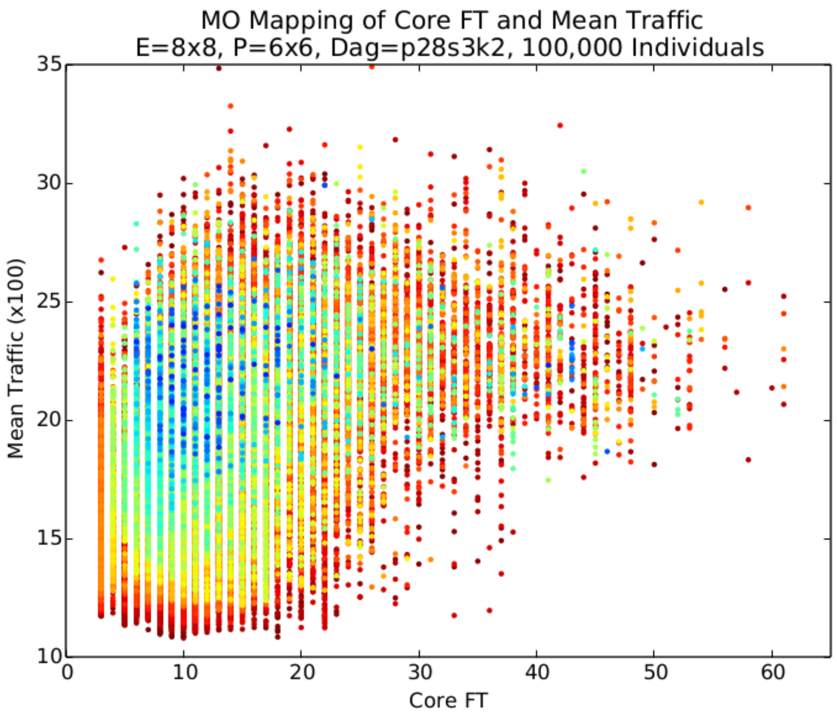


(b) Pareto Front Evolution

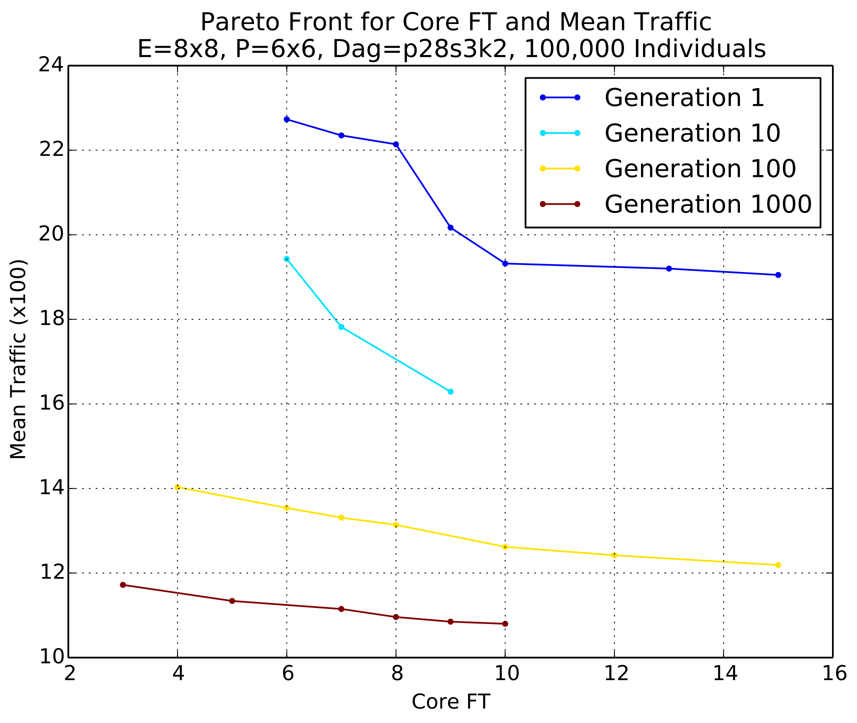
Figure B.12 – MO Core & Link for Moderately Connected Graph



(a) Individuals

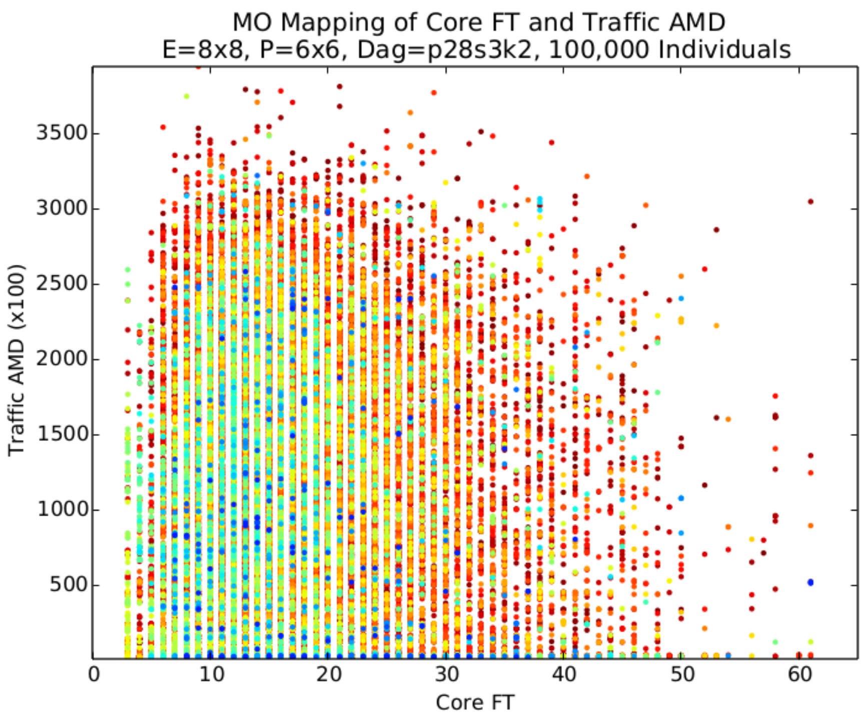


(b) Individuals

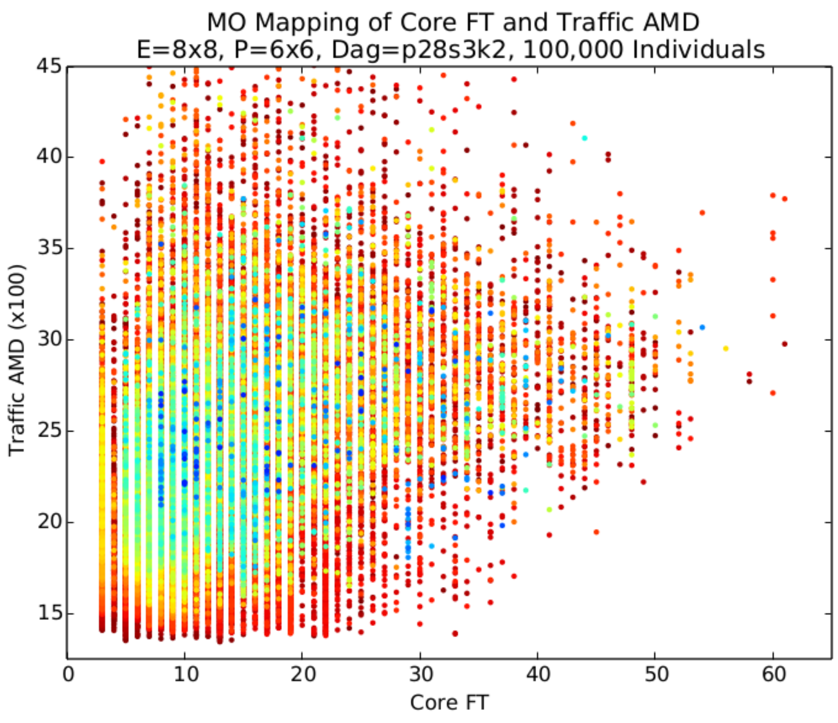


(c) Pareto Front Evolution

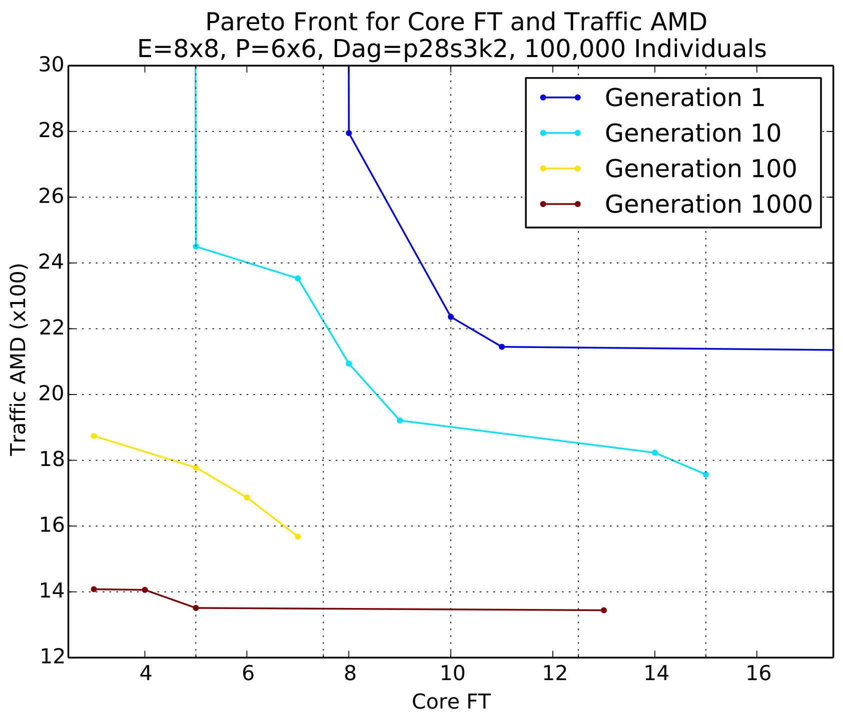
Figure B.13 – MO Core & Mean Traffic for Moderately Connected Graph



(a) Individuals

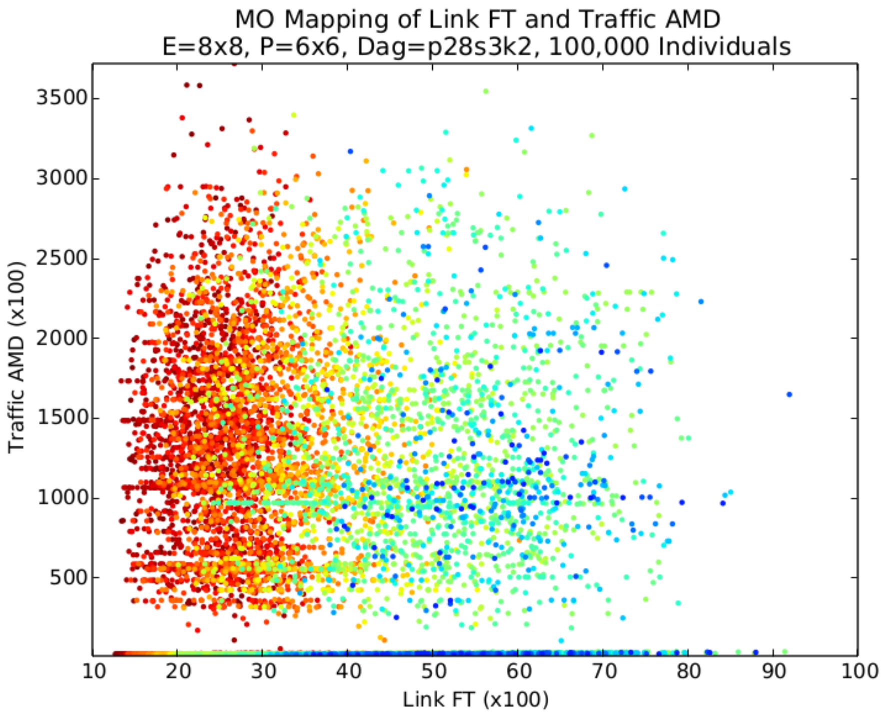


(b) Individuals

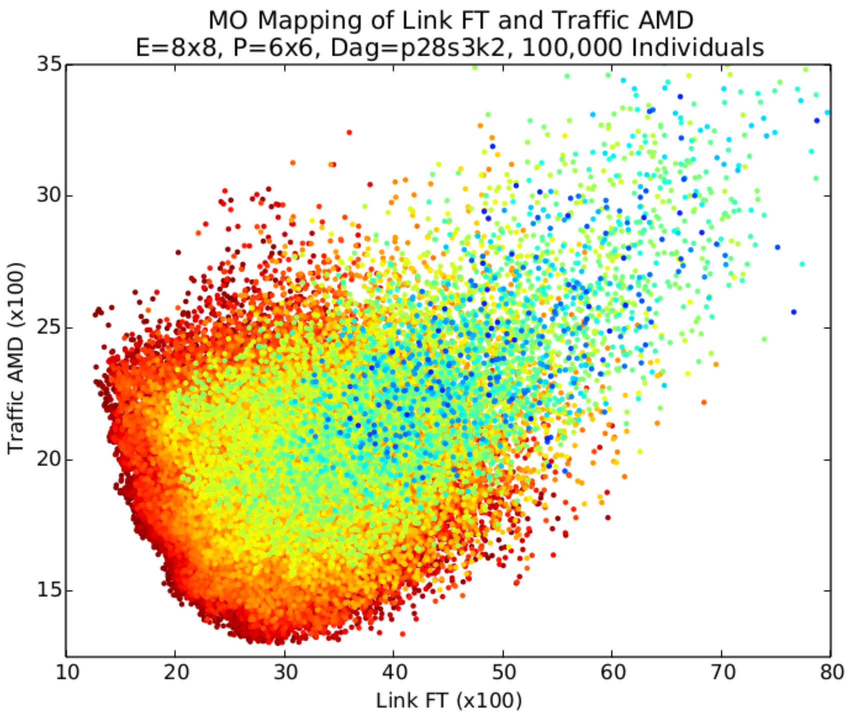


(c) Pareto Front Evolution

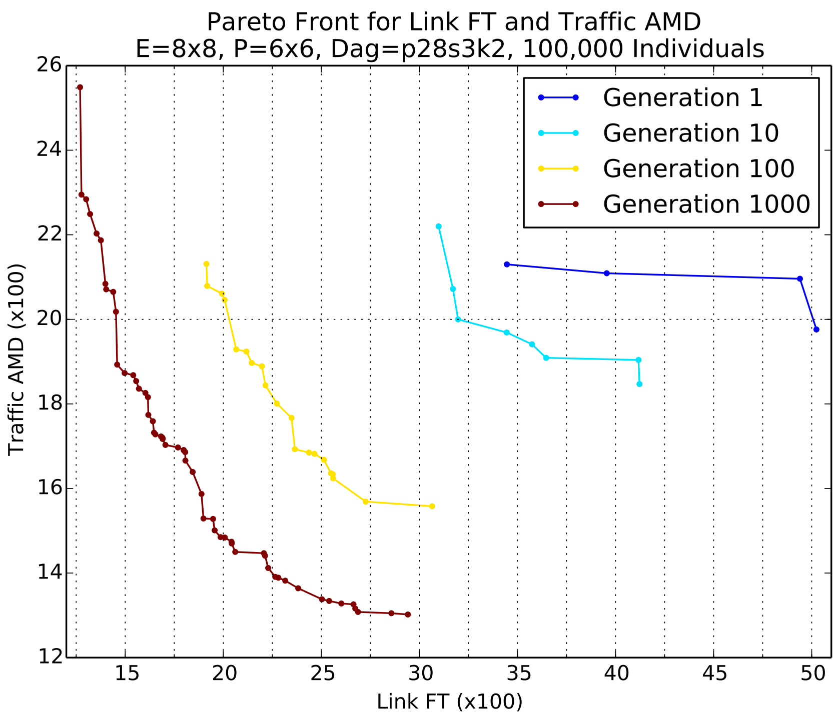
Figure B.14 – MO Core & Traffic AMD for Moderately Connected Graph



(a) Individuals

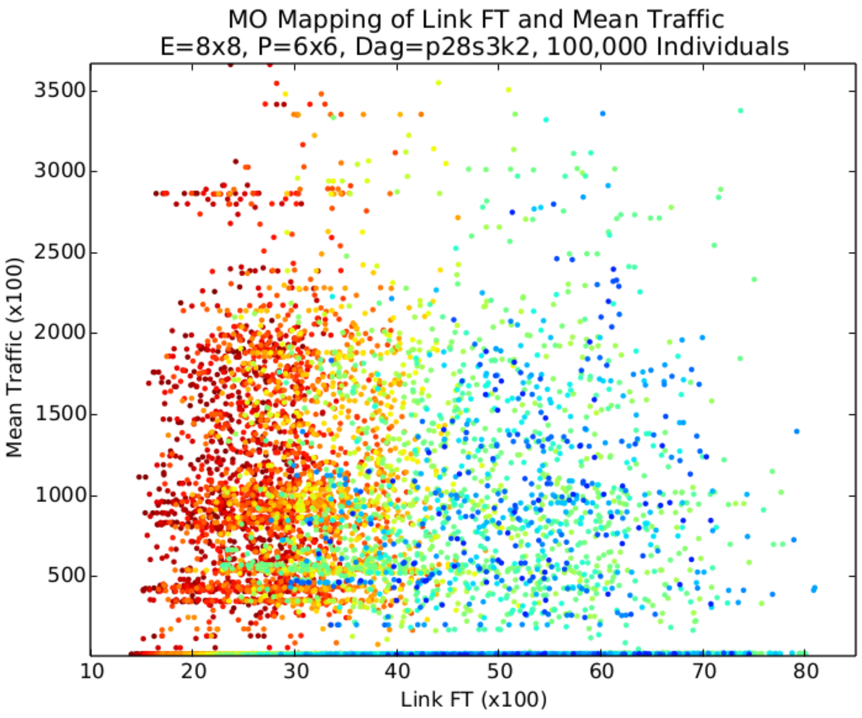


(b) Individuals

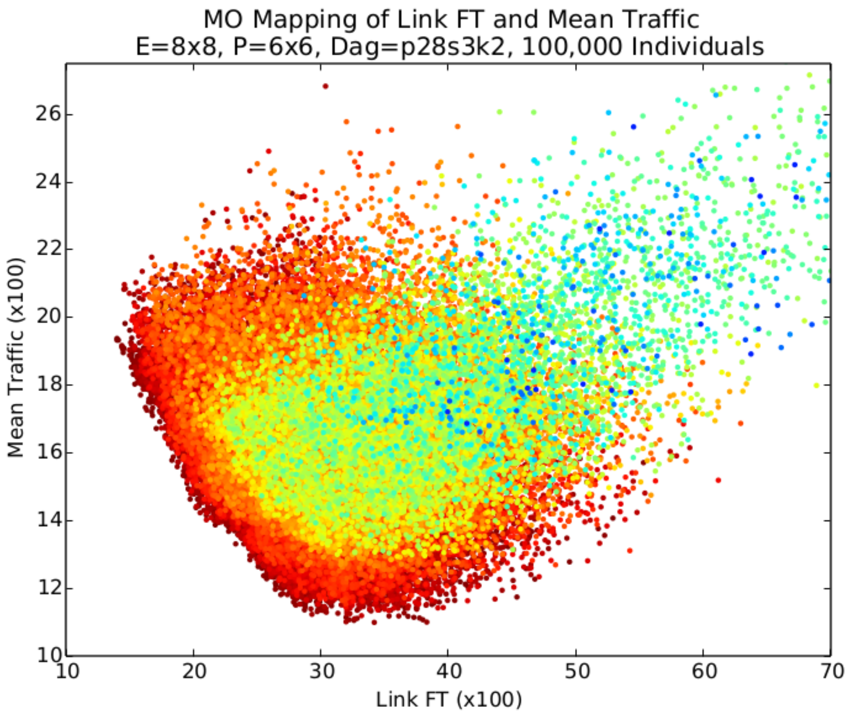


(c) Pareto Front Evolution

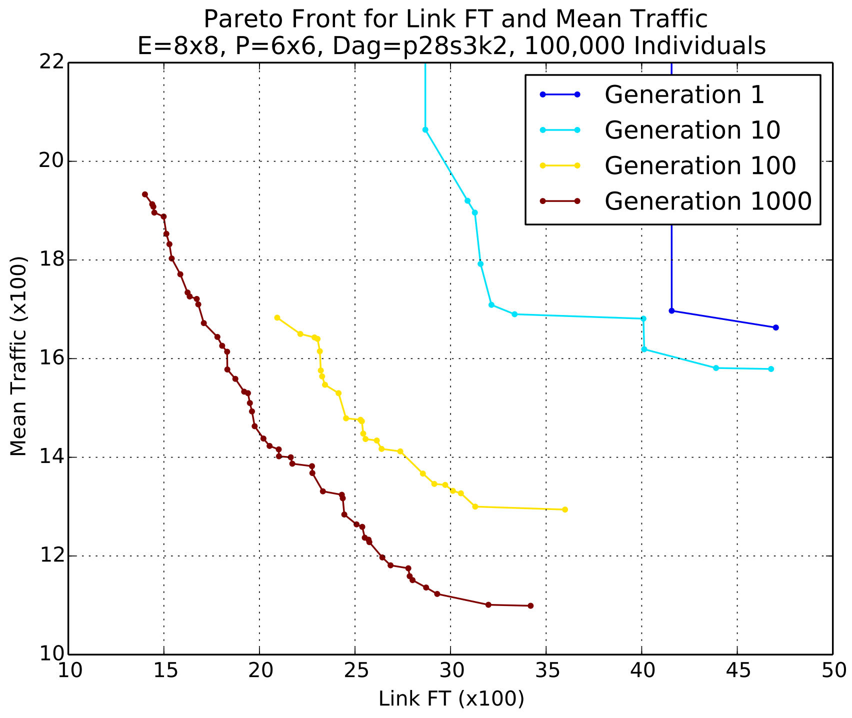
Figure B.15 – MO Link & Traffic AMD for Moderately Connected Graph



(a) Individuals

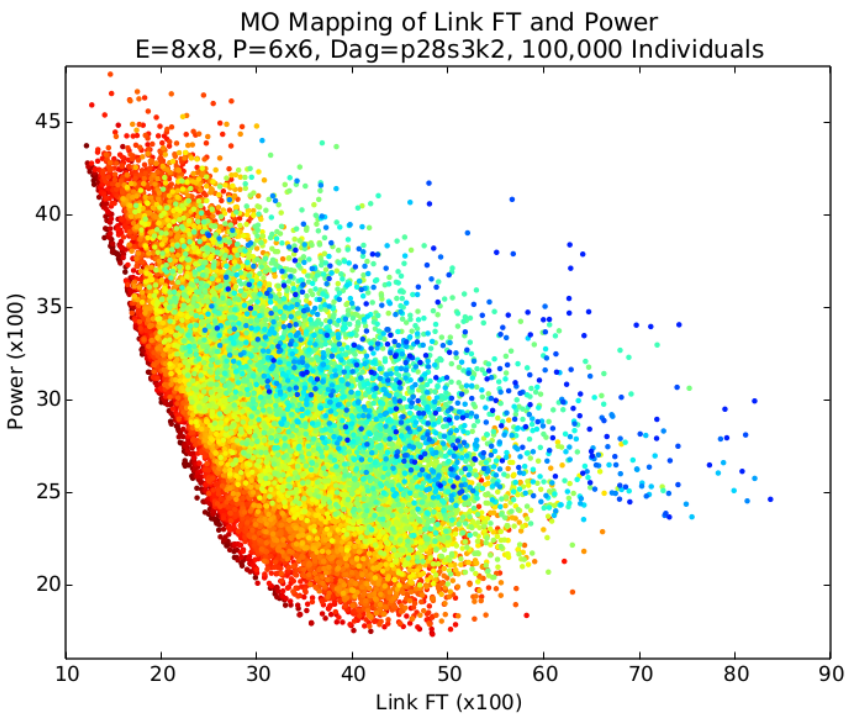


(b) Individuals

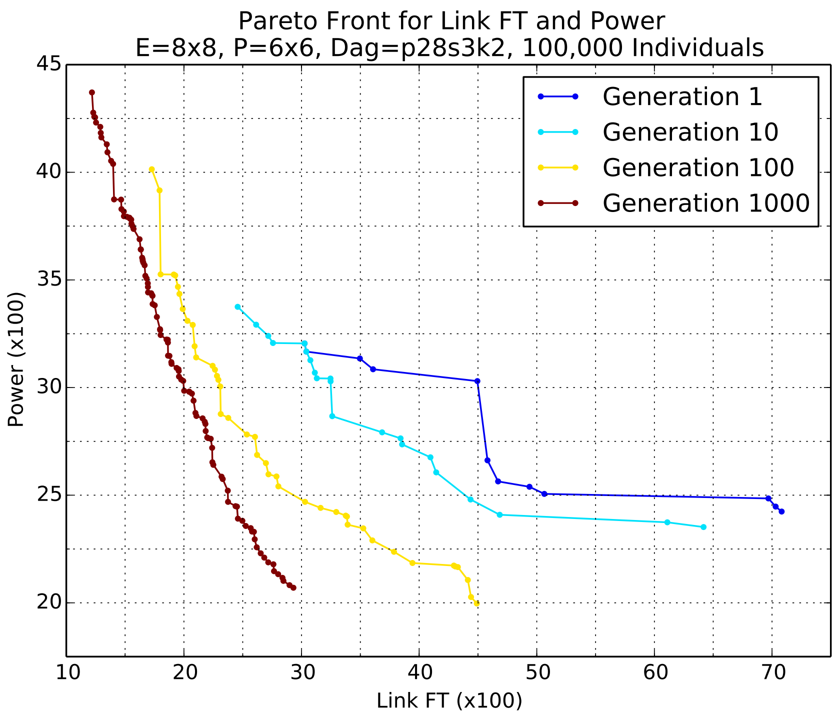


(c) Pareto Front Evolution

Figure B.16 – MO Link & Mean Traffic for Moderately Connected Graph

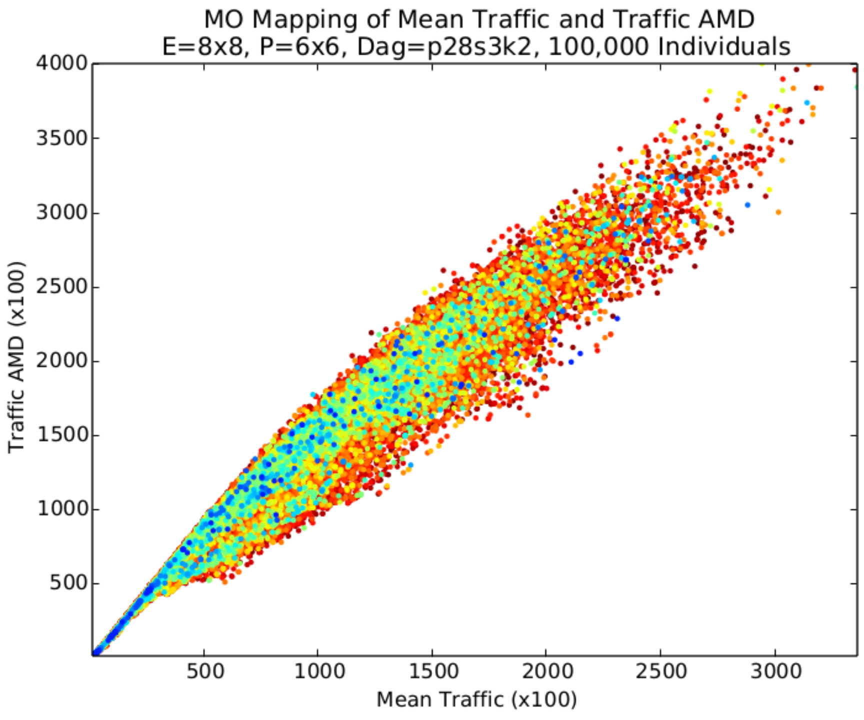


(a) Individuals

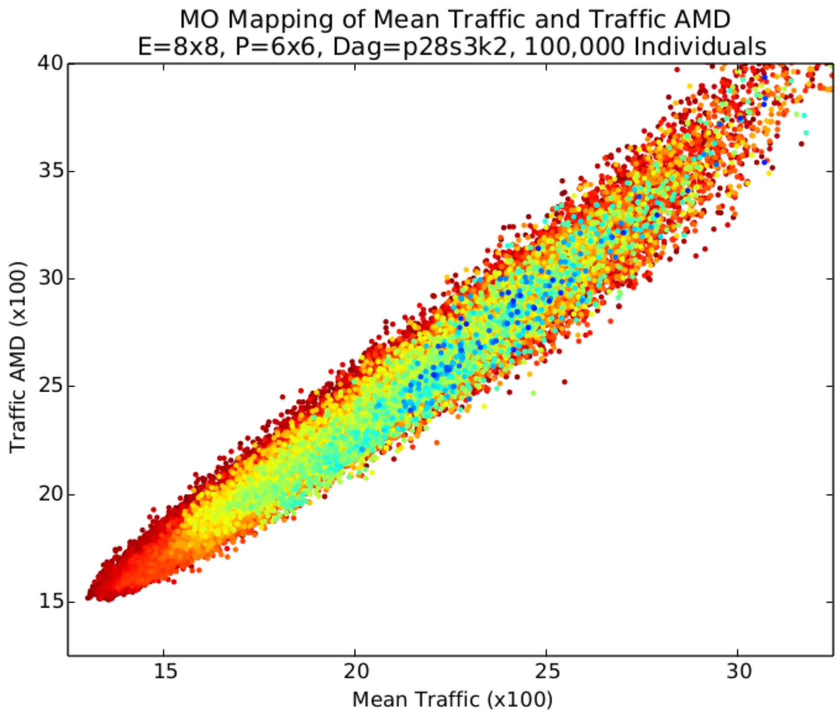


(b) Pareto Front Evolution

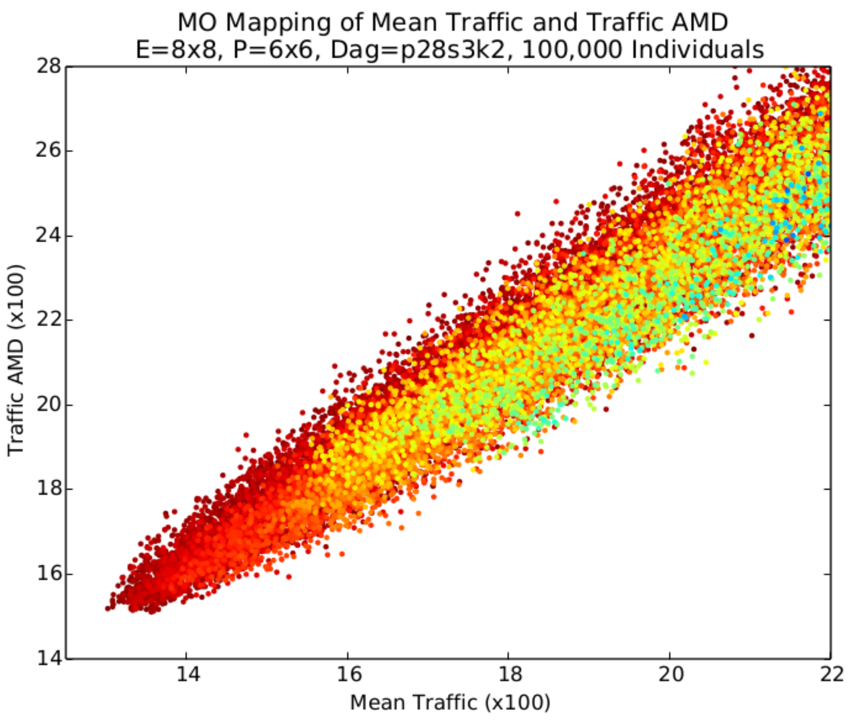
Figure B.17 – MO Link & Power for Moderately Connected Graph



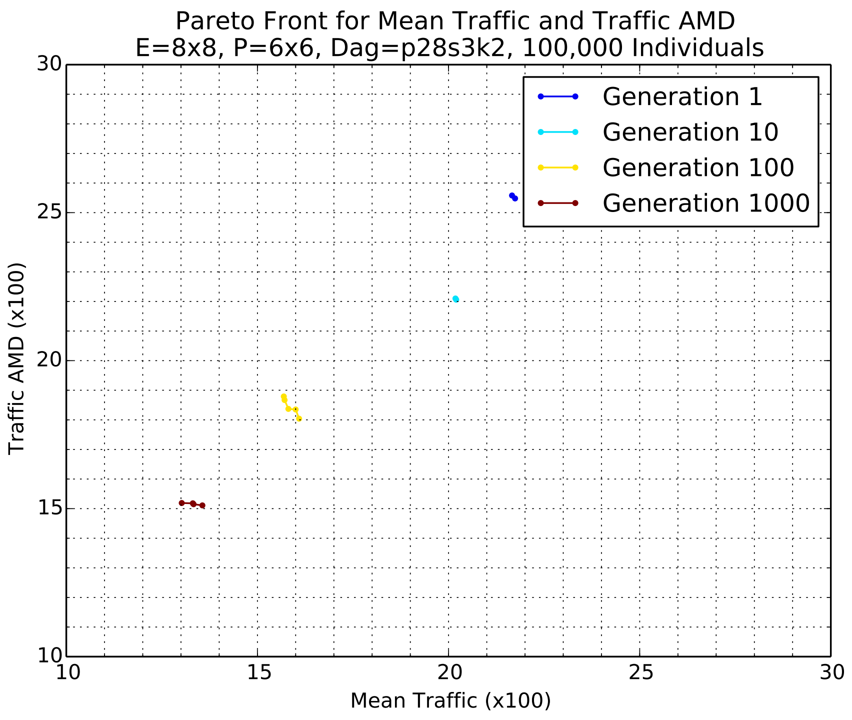
(a) Individuals



(b) Individuals

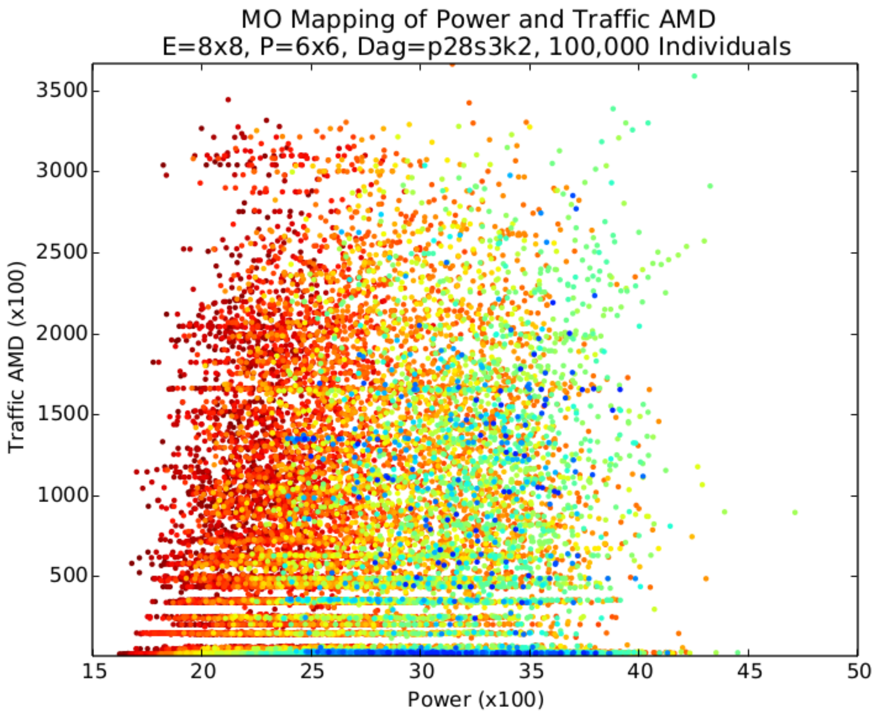


(c) Individuals

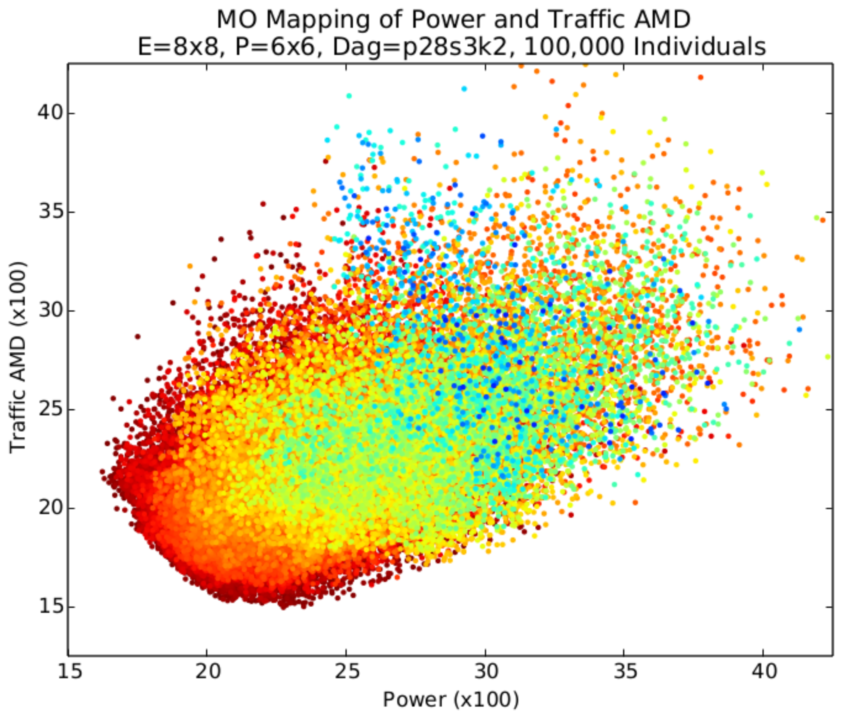


(d) Pareto Front Evolution

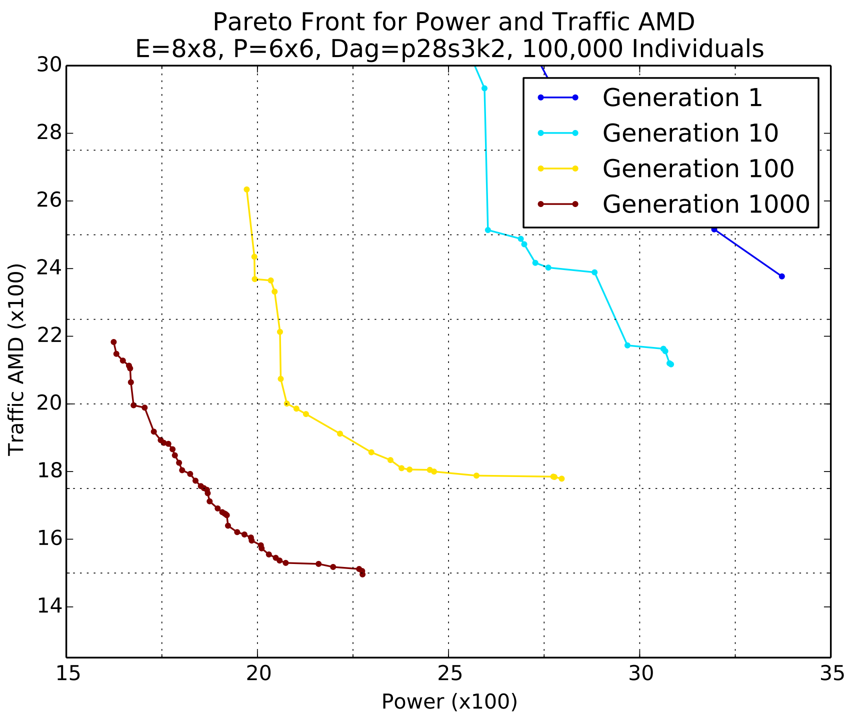
Figure B.18 – MO Mean & Traffic AMD for Moderately Connected Graph



(a) Individuals

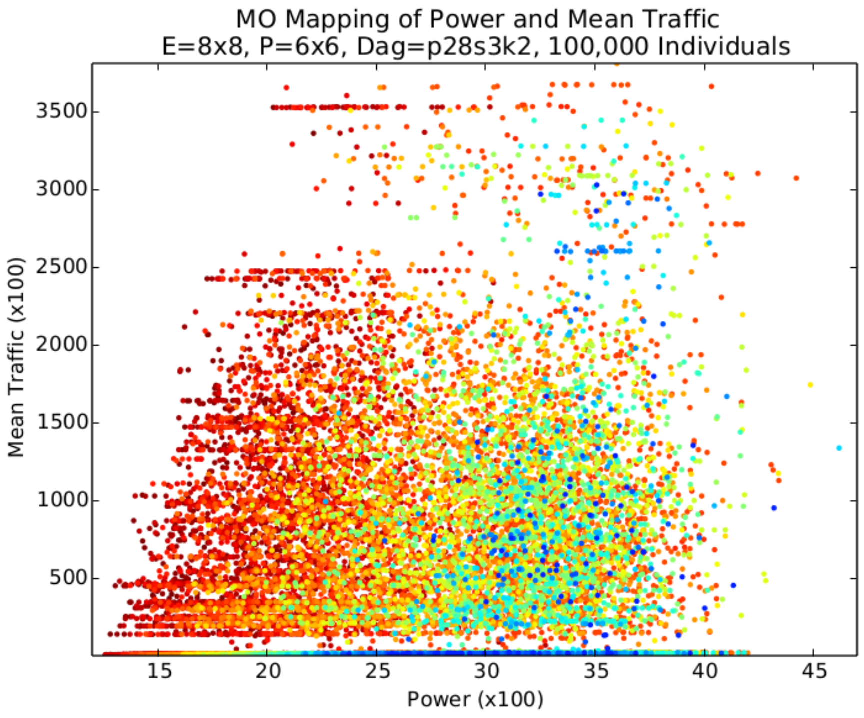


(b) Individuals

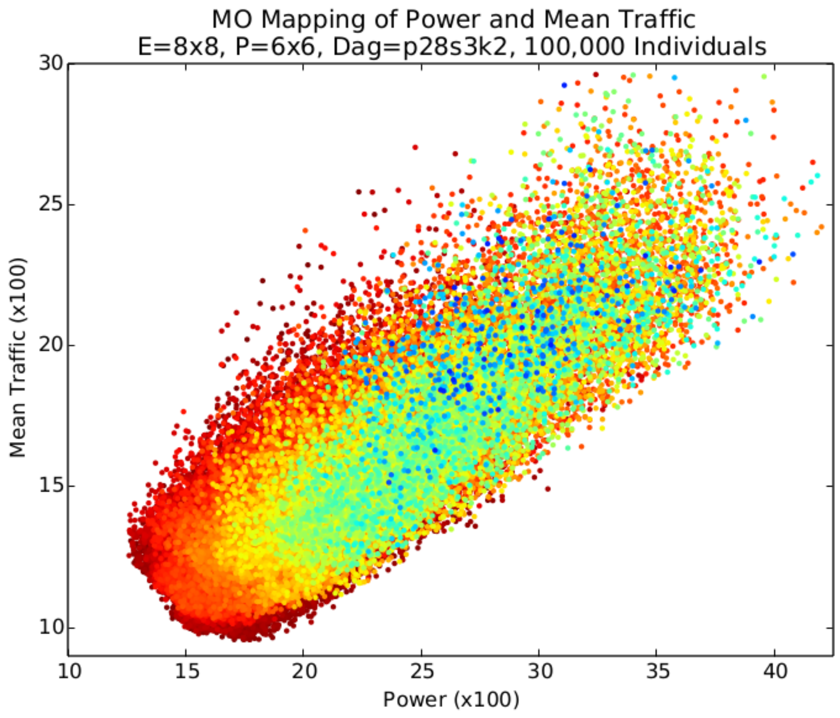


(c) Pareto Front Evolution

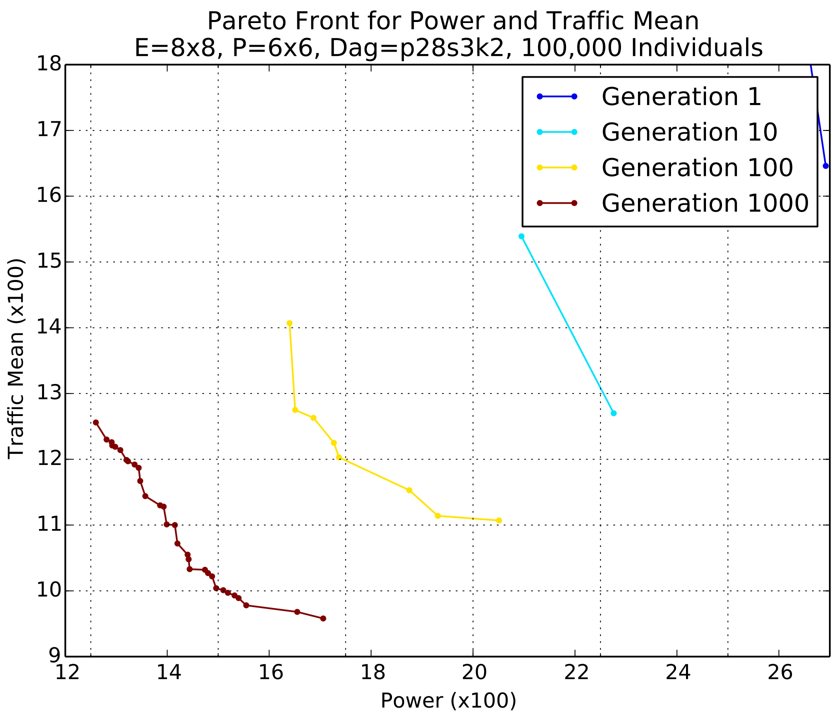
Figure B.19 – MO Power & Traffic AMD for Moderately Connected Graph



(a) Individuals



(b) Individuals



(c) Pareto Front Evolution

Figure B.20 – MO Power & Mean Traffic for Moderately Connected Graph

Appendix C

Evolution of Microprocessor Based Systems

This appendix presents a graphical picture of the evolution of Intel processes focusing on the transistor count in each processor family and relating the transistor count to the technological advances in processor design. The information used has been sources from a number of www.wikipedia.org pages, requiring verification from primary sources.

In 1965 Moore, while employed at Fairchild Semiconductor, wrote a paper in which he identified the historical increase in complexity of components as doubling every year and stating that there was no reason why this trend should not continue [2]. In 1975 Moore revised his prediction for the doubling of transistor counts to take place every two years [19]. Figure C.1 shows how Intel processor designs have kept up with Moore's Law over the 45 years from 1970 to 2015 during which time the transistor count has increased from 2.5×10^3 to 5×10^9, an increase by a factor of 2×10^6.

The vast number of additional transistors have allowed designers to evolve the design of the single chip microprocessor in a number of directions.

This examination focuses on four areas of design illustrated in: Figure C.2 which shows the evolution of the processors word size, Figure C.3 showing the evolution of pipeline stages, Figure C.5 showing the evolution to the number of cores in multi-core processors and Figure C.4 showing the addition of on chip cache memory, also referred to as Level 1 or L1 cache.

The first observation we can make it that evolution of the design focused on one aspect of design at a time. The word length evolved to a size of 32 bits before pipelining evolved. When the number of pipeline stages reached 20 the emphasis moved to the number of cores. Small L1 cache made an appearance at the same time as pipelines and then remained static until multi-core processes made an appearance which were accompanied

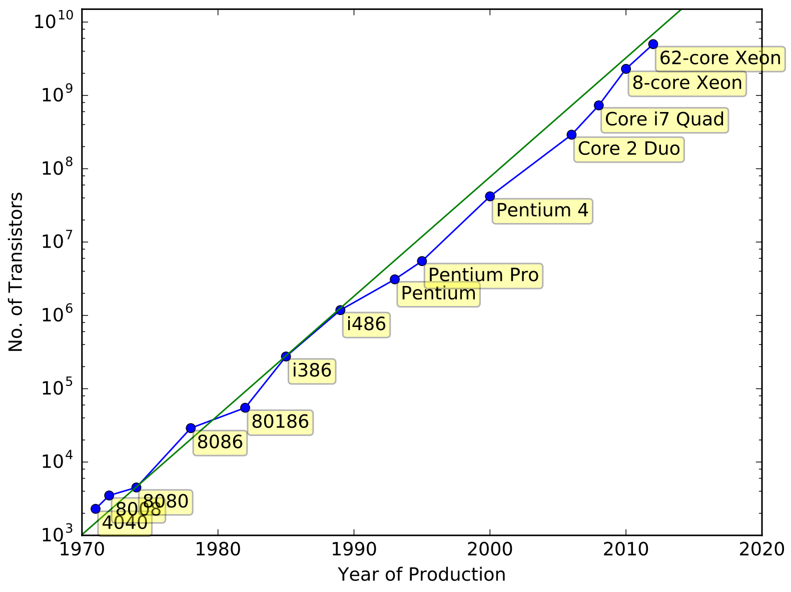


Figure C.1 – Processor Transistor Count

by larger caches.

This staged evolution of one design element at a time can be explained by assuming that designers focused on the design element which offered the best performance gain for the available transistors.

C.1 Word Length

The 8 bit 8080 processor uses an instruction set whose length is 1, 2 or 3 bytes, while the 16 bit 8086 uses an instruction set with instruction lengths varying from 1 to 6 bytes depending on the addressing mode. In both cases the majority of instructions require the fetching and decoding of multiple words, with the fetching and decoding of each word occupying a clock cycle. This reduces the effective speed of the processor when compared to the ideal of each instruction occupying a single word of the machine's architecture.

With the introduction of the i386 processor with a word length of 32 bits, the opcode, addressing mode information and register addresses can be contained within one word with additional words only required for addresses and immediate data, thus reducing the average cycles per instruction when compared to a 16 bit word length.

The benefits of a 32 bit work length compared to a 16 bit word length are considerable,

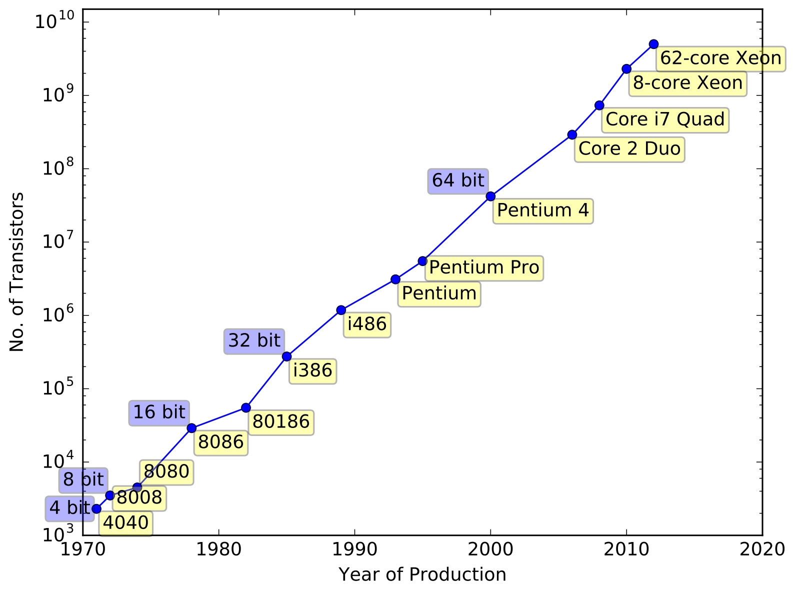


Figure C.2 – **Processor Word Length**

especially when it is recognised that the longer word length also benefits the access of data as well as instructions, thus making this an obvious priority in the design evolution. Word lengths of greater than 64 offer the option of encoding more than one operation in a single word, increasing the number of operations fetched per cycle to more than one.

C.2 Pipeline

Having evolved the word length to 32 bits the emphasis switched to the development of instruction pipelines. The motivation for pipelining is to increase the number of instructions that are executed per second by increasing the clock frequency.

Pipelining arises from the realisation that each instruction passes through a number of serialised stages and that it is not necessary for one instruction to have completed all stages before the next instruction enters the first stage. For example the Intel i486 breaks the instruction execution down into 5 stages [20]:

- FI: Fetch the instruction from cache
- D1: Main instruction decode
- D2: Secondary instruction decode, and memory address computation
- EX: First execution clock

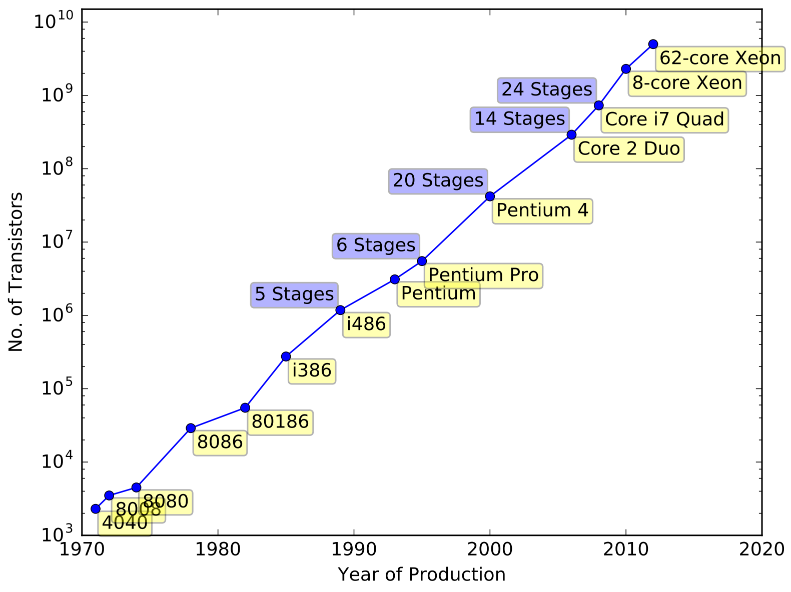


Figure C.3 – **Processor Pipeline Stages**

- WB: Write results into the register file

Each stage takes place in a single clock cycle. Without pipelining an instruction that requires all five stages to be executed requires a long clock cycle to complete. With pipelining the i486 can execute five instructions simultaneously reducing the average execution time for some instructions to one faster clock cycle. Pipelining achieves this because each stage is relatively simple and so uses fewer logic gates, reducing the amount of time required for the logic to complete its task and therefore allowing the clock frequency to be increased [21].

The benefit of the i486 five stage pipeline can be illustrated by comparing the number of clock cycles required to complete execution of a selection of instructions to the equivalent i386 instructions [20]:

Instruction	Clock Cycles	
Type	i386	i486
LOAD	4	1
STORE	2	1
ALU	2	1
JUMP taken/not	9/3	3/1
CALL	9	3

In the Pentium 4 design the number of pipeline stages increased to 20 which, Intel reported, increased the relative frequency of the Pentium 4 when compared of the to the i486 by a factor of 2.5 [21].

A balance needs to be achieved between the number of logic gates in a pipeline stage and therefore the increase in frequency and the number of pipeline stages which increases the total number of clock cycles required to complete the execution of an instruction. The optimum, based on Intel processor designs, seems to be in the region of 20 stages.

A further evolution of pipeline technology is *superscalar* which is a design that has multiple pipelines working in parallel to increase the number of instructions executing simultaneously.

C.3 On Chip Cache

Memory is a relatively transistor hungry resource, using 50M transistors per megabyte of memory based on a design that uses 6 transistors per memory cell, so was lower down the list of processor features to take advantage of the increase in transistor numbers. As can be seen in Figure C.4, L1 cache first appears in the i486 at the same time as pipelining was introduced into the the processor design. Although the 8KB cache of the i486 seems small, especially when compared to the later multi-megabyte caches, it accounts for roughly 4% of the transistor count of the i486.

In single core designs the L1 cache was on-chip and stayed below 1MB while L2 was generally, but not exclusively off-chip. With the advent of dual core processors, most designs allocated a quantity of L1 cache for the exclusive use of each core and added on-chip L2 cache which is shared between multiple processors. As a result we see a step change in the size of on-chip caches from kilobytes to megabytes that coincides with the introduction of multi-core designs.

C.4 Processing Cores

The continuing growth of the number of transistors available to designs next found a use is multi-core designs. Multi-core designs provide two or more identical and sophisticated processing cores on a single die. The sophistication and power of the cores is similar to the that of single processor designs. The processing cores are independent while sharing resources such as memory and communication buses.

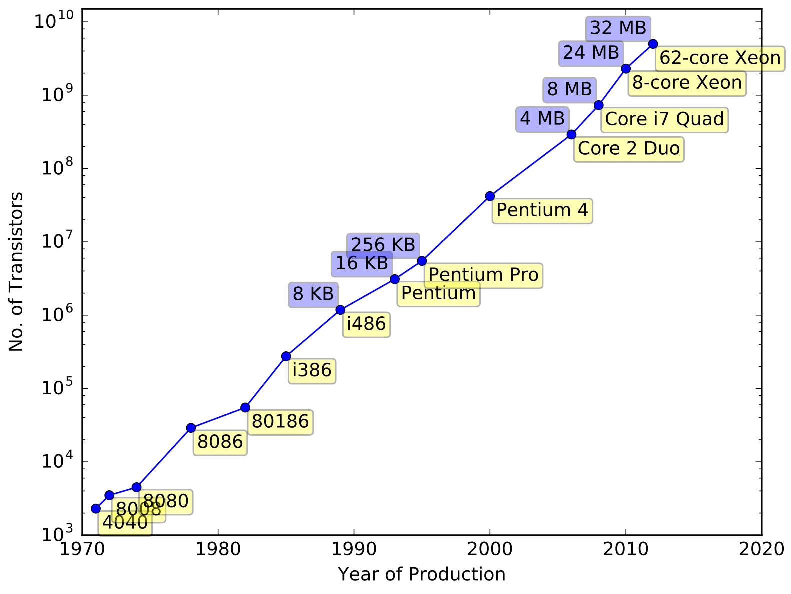


Figure C.4 – **Processor Cache Size**

- 2005 Intel released their first dual core processor, the Intel Pentium Processor Extreme Edition 840
- 2007 Intel released a Core 2 Quad Processor
- 2014 Intel unveiled its first 8 core desktop processor, the Intel Core i7-5960X processor Extreme Edition
- 2016 Intel released the 64 core Intel Xeon Phi Processor 7210
- 2016 Intel released the 72 core Intel Xeon Phi Processor 7290

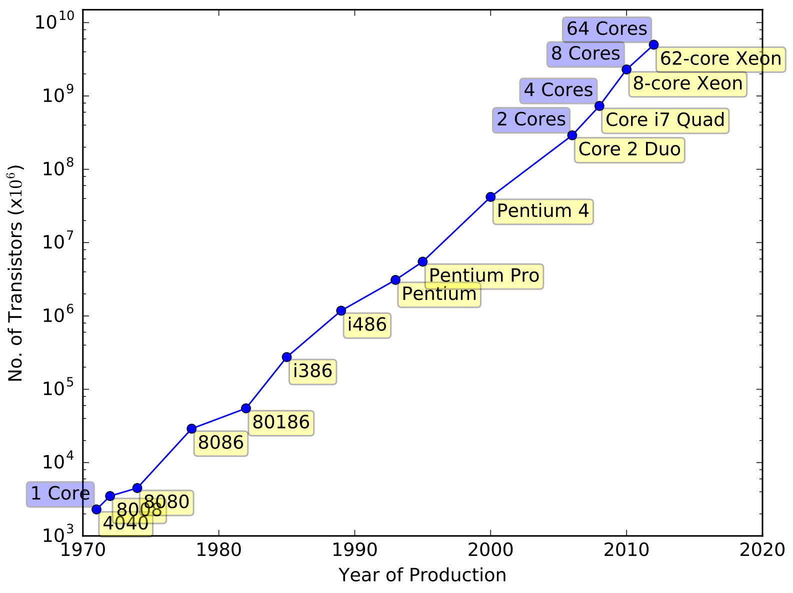


Figure C.5 – Processor Number of Cores

Appendix D

Many-Core Implementations

D.1 The Graceful Project

This appendix is taken from the proposal for the EPSRC funded project EP/L000563/1 **Continuous on-line adaptation in many-core systems: From graceful degradation to graceful amelioration**, known informally as the *Graceful* project which is a joint project between The University of York, The University Manchester and University of Southampton.

Project Outline Imagine a many-core system with thousands or millions of processing nodes that gets better and better with time at executing an application, "gracefully" providing optimal power usage while maximizing performance levels and tolerating component failures. Applications running on this system would be able to autonomously vary the number of nodes in use. The proposed project aims at investigating how such mechanisms can represent crucial enabling technologies for many-core systems.

Specifically, the project focuses on how to overcome three critical issues related to the implementation of many-core systems, as identified in the Many-Core Architectures and Concurrency in Distributed and Embedded Systems workshop organized by the EPSRC in April 2012: reliability, energy efficiency, and on-line optimisation. The need for reliability is an accepted challenge for many-core systems, considering the large number of components and the increasing likelihood of faults of next-generation technologies, as is the requirement to reduce the heat dissipation related to energy consumption. On-line optimisation, on the other hand, is a mechanism that could be vital to enable the implementation of these properties in systems where several parameters (e.g. number of available cores, power profile, substrate defects and on-line faults) cannot be known at compile time and cannot be managed centrally due to the vast number of cores involved. The 2011 ITRS roadmap strongly supports these considerations.

The proposed approach is centred around two basic processes: *Graceful degradation* implies that the system will be able to cope with faults (permanent or temporary) or potentially damaging power consumption peaks by lowering its performance. *Graceful amelioration* implies that the system will constantly seek for alternative implementations that represent an improvement from the perspective of some user-defined parameter (e.g. execution speed, power consumption).

Approach The project will combine novel fault tolerance and optimisation methods (which can be optimisation of functional performance but also of power consumption, area, etc.) using a set of background mechanisms that run concurrently with the applications, making use of spare cores (of which in a many-core system there will be significant numbers) to monitor and fix/optimize the operational cores. Thus, an application that runs rarely will be largely ignored but one that runs often will slowly become more and more optimized.

The basic scenario we will consider is that of an application running on a system consisting of many, possibly heterogeneous, processor cores. While acknowledging the fundamental importance of software compilation techniques, the project will focus on the hardware aspects of a many-core system, assuming that the application has been pre-parallelized by the user and starting the investigation from the output stage of the compiler. Each core will then handle a specific subprogram, or task, within the application.

In the project, standard benchmarks (e.g., SPEC2000 and/or PARSEC) will be used for initial investigations, but we will quickly progress to a number of real-life applications, selected in consultation with our industrial contacts. We will consider in priority "real-world" applications (e.g. audio/video encoding and decoding such as MPEG, encryption/decryption protocols, etc.) with a particular focus on streaming applications, where the data flow can be mapped onto a directed acyclic graph of task nodes, and applications that can tolerate temporarily incorrect results (e.g. dropped frames in a video stream). An Industrial Advisory Board (see Impact section) will help identify commercially-relevant applications.

On the implementation side, we will deploy our system on two platforms. To demonstrate the "immediate" advantages of the approach, we will implement a software-based solution to the SpiNNaker system [9], probably the machine that most closely approximates a many-core system currently available in the UK (see Resources section for details). Using conventional ARM cores, this implementation will illustrate how the features of the proposed approach can impact current technology.

To more fully investigate the advantages of our system and to analyse the fundamental trade-off between homogeneous and heterogeneous systems (a crucial issue in many-core research), the second platform will consist of a set of custom FPGA-based boards. While the development of a custom IC would allow greater scope for our investigations (and remains an option for a continuation project in the future), the exploratory nature of this project is better suited to a programmable logic device, which will allow us freedom to investigate alternative node architectures. The design of custom boards will allow us to introduce complex power measurement and management components that are not available in off-the-shelf solutions.

Implementation strategy The proposed implementation approach relies on the observation that, in a many-core system, the abundance of resources implies that not all cores will be used for the execution of applications. The overall operation of the system (Figure 1) will rely on four different kinds of units (in addition to the empty nodes, currently not used but at the disposal of the application):

- Application Units (AU) represent the "conventional" processing nodes running the target application. These could be conventional processors (e.g. ARM cores in SpiNNaker) or dedicated units containing programmable logic. If the latter, the Optimization Units will explore hardware/software trade-offs to seek improved implementations.

- Monitoring Units (MU) will monitor and profile the execution of the AUs and, if necessary, take steps to resolve issues. Monitoring will be carried out according to user-specified parameters: in this project, we will focus specifically on fault tolerance and power consumption, with a lesser emphasis on execution speed, but the general approach could easily be expanded to handle different parameters. They will operate in the background, without affecting the execution of the application unless an issue is detected or an improved implementation is found. The optimal density of MUs (and hence their radius, i.e., the number of monitored AUs) is a parameter that depends on several factors and will be attentively examined in the project. To avoid single points of failure, MUs will also instantiate reciprocal monitoring.

- Optimisation Units apply search algorithms to identify alternative implementations of application nodes. They will operate on intermediate code (e.g. compilation trees) and apply functionally-neutral code transformations and/or hardware/software trade-offs to generate and evaluate Candidate Units. Previous research ([2,3,12]) has shown that Evolutionary Algorithms (EAs) can be an efficient approach for this kind of search, particularly considering that the optimisation will be implemented as a background process that requires neither a single optimal solution nor a fast reaction time.

- Candidate Units, generated by the Optimisation Units, represent alternative implementations of AUs. They are evaluated according to the desired parameters (power consumption, performance, etc.) and, if they represent an improvement over the original nodes, will eventually replace them. The replacement will be handled by the MUs and will be non-disruptive (e.g., will occur after the internal states of the nodes have been aligned and at specific and predictable points in the code).

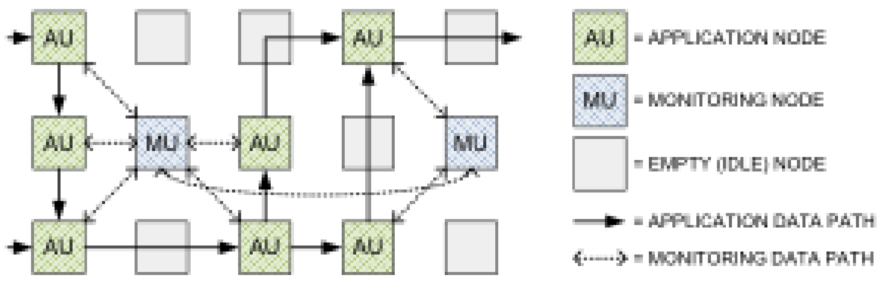


Figure D.1 – Many-core scenario - Monitoring with radius = 1

To ensure scalability, all units will be physically implemented by the same processing nodes (in other words, the same kind of hardware processors) and all interactions will be local, with no centralized control. To more closely model a probable scenario in future many-core systems, the general structure of nodes will be based on a processor with a "base" datapath (Core Processing Unit & Local Memory) plus an area of programmable logic, which can be used to optimize performance, by exploiting hardware acceleration and effectively implement heterogeneous systems, and for fault tolerance, by providing alternative computational elements. However, in order to increase the potential impact of the approach, all the mechanisms developed in the project will have a general application to many-core systems that do not include programmable logic, a versatility that will be demonstrated through the dual implementation on the SpiNNaker architecture and on custom processing boards.

Operation The following example scenarios illustrate the projected operation of the system. These scenarios, by no means exclusive, should illustrate the general operation of the system to be developed in the project. While these seem rather well defined and worked out, their practical implementation on many-core systems is far from worked out or trivial. Achieving the desired properties will involve the development of novel mechanisms at several levels of the system design cycle, from the output of the compilation process through to the hardware architecture of the nodes and the interconnection network. Achieving these properties in an appropriate way and developing the mechanisms to make them functional is the core of this proposal.

Secnario I *(graceful amelioration for performance):* A Monitoring Unit [A] detects that one of the Application Units is a bottleneck in the performance of the system (e.g., in [2] this was detected by monitoring I/O FIFOs). If execution speed is a critical factor in the system, the MU can implement a rapid solution by instantiating an identical copy of the slow Application Unit to share the load [2]. At the same time [B], it might generate an Optimisation Unit that will run a search algorithm which will seek alternative implementations for the task. These are tested by running them in parallel to the existing node in one or more Candidate Units. The CUs receive the same input data as the original AU, and their performance is compared. Non-optimal implementations are discarded. If the OU identifies that the candidate node outperforms existing node [C], the original node is discarded, returned to the pool of empty nodes, and replaced by the new one, which contains the same state and can therefore be seamlessly substituted.

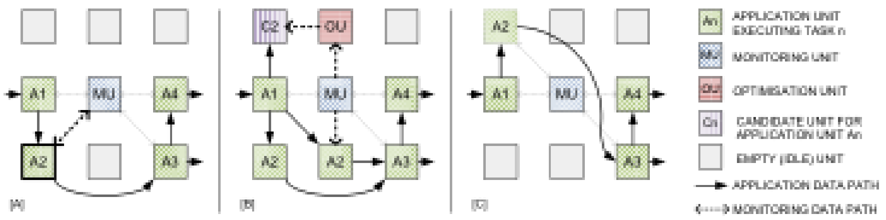


Figure D.2 – **Graceful amelioration for performance**

Secnario II *(graceful amelioration and degradation for power consumption):* A scenario for power consumption could be somewhat similar to the scenario for performance (indeed, the objective of the project is to define general mechanisms that can be applied for the optimization of any parameter). A Monitoring Unit [A] detects that one of the Application Units is exceeding acceptable thresholds of power consumption (spikes due to possible faults or generally high computational load), which could potentially lead to failure. Again, the MU could act in two steps. First, it could act directly on the AU, for example by reducing clock frequency and/or lowering voltage (node A2* in [B]), which could lead to a (graceful) degradation in performance but avoid the risk of catastrophic failures (if the degraded performance is not adequate, Scenario I will then "kick in", illustrating the seamless interfacing between optimisation scenarios). At the same time, once again an Optimization Unit could be created to seek for alternate implementations of the application task with better power efficiency, using a mechanism identical to the one described for Scenario I, but with a different target for optimisation ([B] and [C]).

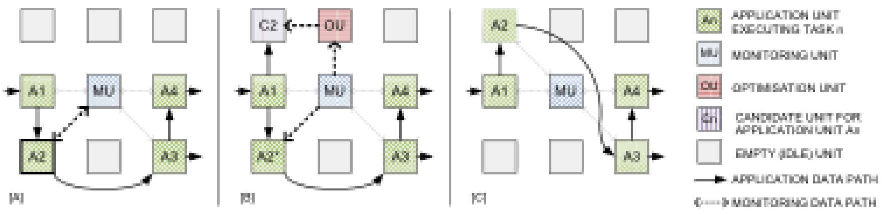


Figure D.3 – Graceful degradation for fault tolerance

Secnario III *graceful degradation for fault tolerance):* This scenario presents a certain number of differences compared to the previous two, in that the presence of a fault could potentially load to immediate catastrophic failure of a node. Once a fault in one of the monitored has been detected by the MU (A2x in the figure [A]), several different options are possible. In this project, our main focus will be on streaming DSP applications, where the loss of information, if temporary, can deemed acceptable A first reaction of the MU [B] could then be to act on frequency and voltage levels of the node (A2*) to determine whether the fault can be (temporarily) masked and allow computation to continue, albeit at a lower speed (graceful degradation). At the same time, the MU will activate one of the idle units to replace the faulty one, attempt, where possible, to copy and transfer the state of the faulty AU, and, once the process is complete, disable the original node and bring on-line its replacement [C]. At this stage, the original node can be thoroughly tested and, if deemed fault-free (for example, in case of a non-permanent fault), returned to the pool of idle nodes.

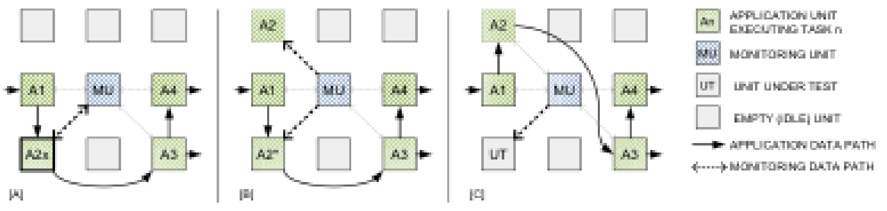


Figure D.4 – Graceful amelioration for performance

D.2 Paralella

D.2.1 Adapteva

Adapteva Inc was formed in 2008 and has been successful in fabricating a 16 core chip called Epiphany with plans to produce a 64 core chip. Adapteva produce credit card sized development boards which incorporate the 16 core Epiphany chip alongside a Xilinx 7010/7020 Zynq processor.The Epiphany chips are also available as individual components which can be incorporated into end user designs.

D.2.2 Epiphany

The Epiphany chip has been designed to be scalable either by fabricating more cores on a single chip or by connecting a number of chips together. There are two varieties of the Epiphany chip, one with 16 cores and another with 64 cores. Whilst the 16-core chip is available in commercial quantities the availability of the 64-core chip has proved to be limited. Both the 16-core and 64-core chips are designed within an overall framework which supports architectures of up to 4096 cores. Chips can be directly connected to other chips in a mesh pattern.

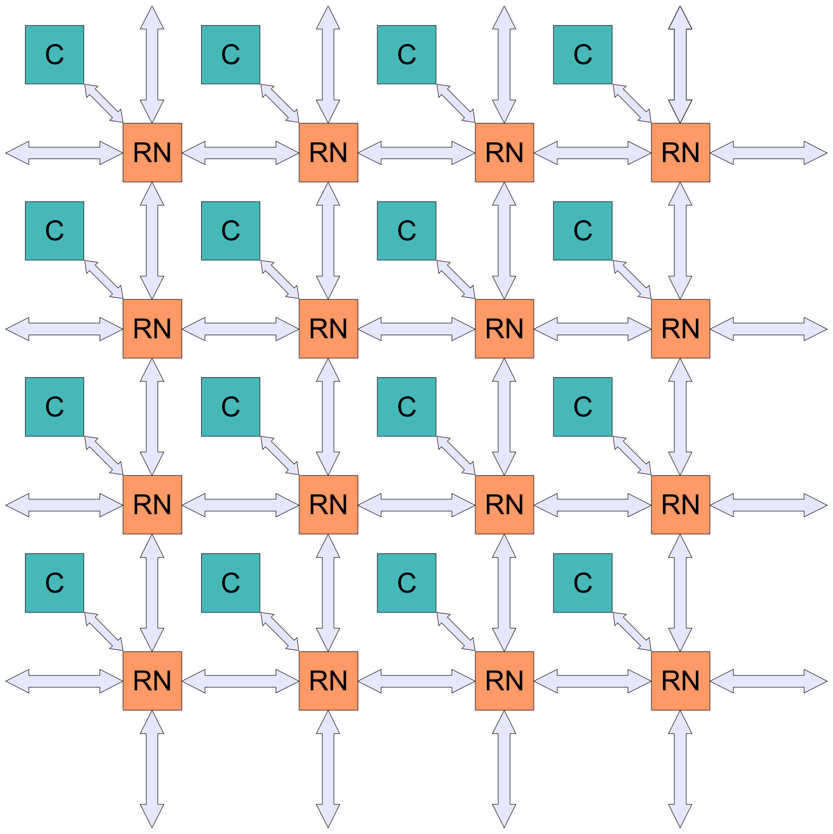


Figure D.5 – Epiphany Mesh Architecture

Figure D.5 illustrates the basic architecture of the Epiphany mesh consisting of mesh nodes within a routing network. Each Mesh Node consists of a CPU, DMA Engine, Memory and Network Interface. The routing network consists of three separate communication channels, one for on-chip data writes, one for off-chip data writes and one for all reads.

D.2.3 Memory

Each Epiphany core has 1Mb of local memory, giving a total of 4Gb across the maximum available 4096 cores, requiring 32 bit addressing. Local memory of each core is accessible by all the other cores in the mesh. The lower 20 bits address memory locations in the local memory, while the high end 12 bits are used to specify one of the 4096 organized in a 64x64 mesh. Cores are given an (x,y) coordinate. Taking the top left core as core (0,0), bits (31-26) of the 32 bit address gives the x coordinate and Bits (25-20) gives the y coordinate.

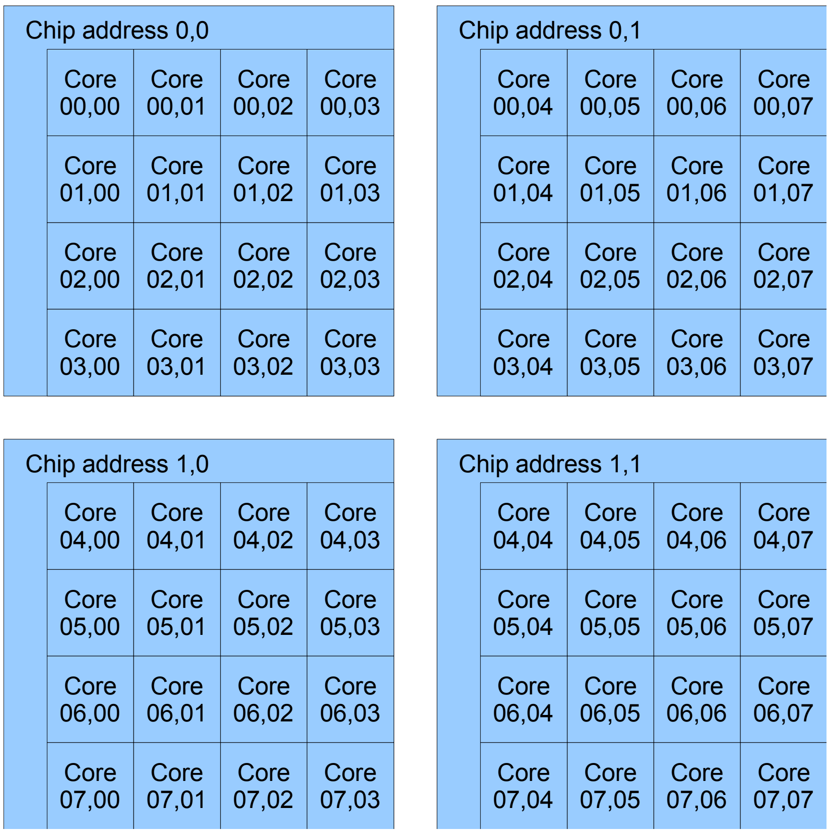


Figure D.6 – **Epiphany Core Addressing**

Cores are arranged in a grid mesh with a maximum dimension of 64x64, which an be achieved using a 16x16 array of 16-core Epiphany chips. Figure D.6 illustrates how cores are addressed using four 16-core Epiphany chips. All addresses are given in decimal. Each chip is given a row address from 0-15 (decimal) and column address from 0-15 (decimal). Within each chip, each core is given a row address from 0-3 (decimal) and column address from 0-3 (decimal). Combining the chip address with the core address gives a unique address for each core in the mesh.

The routing protocol requires the chip addresses to coincide exactly with the physical positions of the chips in the mesh. The chip address is set by applying signals to 8 address pins and applying a reset signal, four pins representing the row address of the chip in binary and four pins representing the column address in binary. Addresses need to be applied to each chip during the system boot-up.

According to the documentation only 32Kb of the memory is available for use by the core with the majority being 'reserved for future use'. This limits the six e of programs and data that can reside in local memory.

D.2.4 Routing

Routing of data packets between cores is controlled by the routing protocol designed into the chip fabric. Each packet sent through the mesh is routed first east-west until the packet arrives at the target column and then north-south until the packet reaches the the target core.

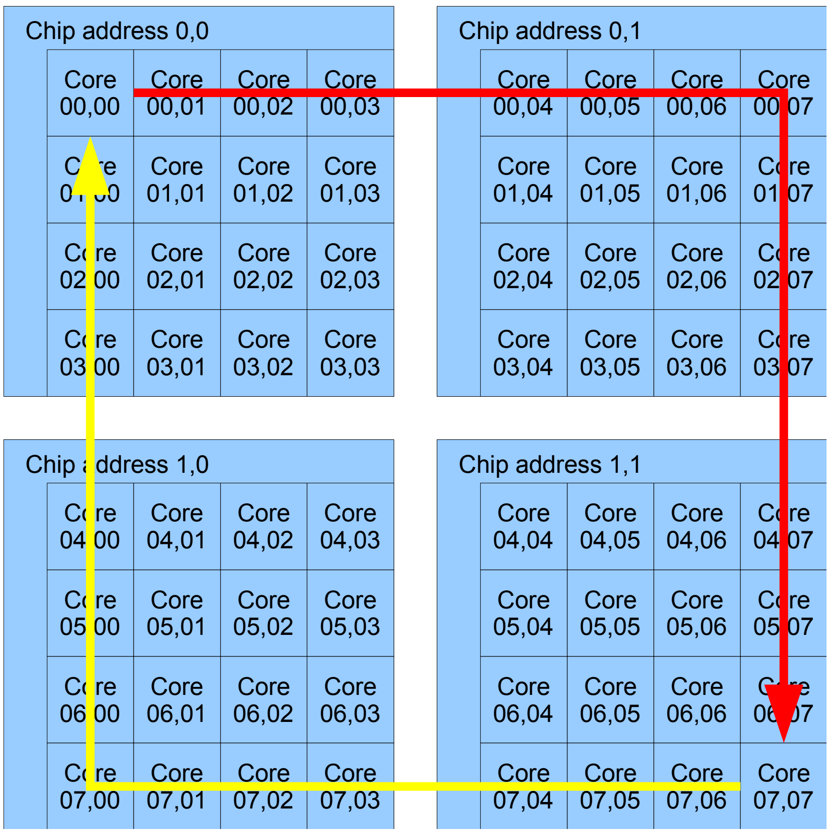


Figure D.7 – Epiphany Packet Routing

Figure D.6 illustrates how a packet would be routed between cores (0,0) and (7,7). A packet travelling from core (0,0) to core (7,7), represented by the dark arrow would first be routed east through the cores on row 0. When the packet reaches column 7 the packet is routed south down column 7 until it reaches the core (7,7). The return packet from core (7,7) to core (0,0), represented by the light arrow, first travels west along row 7 and then north up column 0. Thus, due to the protocol using an east-west route followed by a north-south route the outward and inward packets travel along different routes.

When connecting chips together, the routing protocol demands the North side of a chip is connected to the South side of another and the East side of a chip is connected to the West side of another.

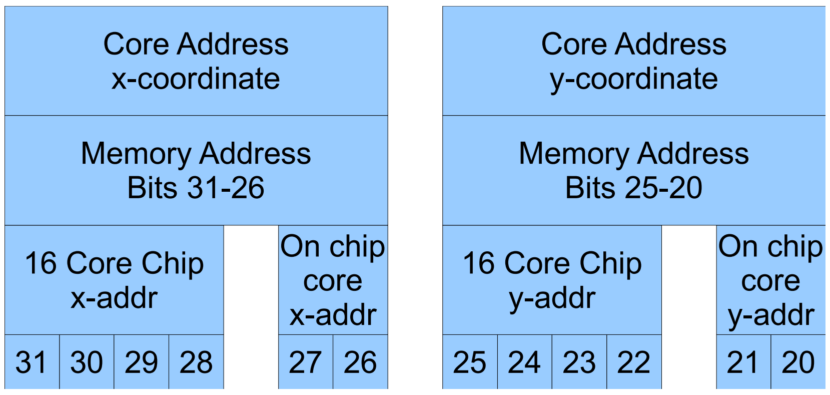


Figure D.8 – **Epiphany Core Addressing**

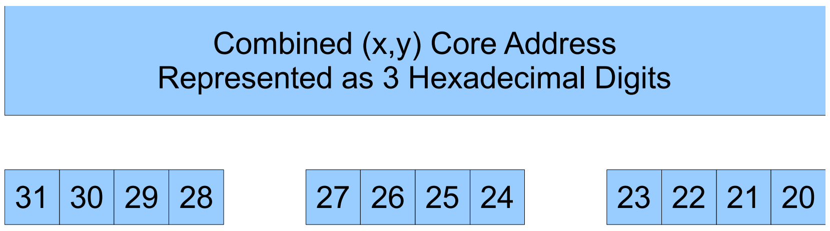

Combined (x,y) Core Address
Represented as 3 Hexadecimal Digits
31
30
29
28
27
26
25
24
23
22
21
20

Figure D.9 – **Epiphany Core Hex Address**

Figures D.8 and D.9 illustrate how the 12 high end address bits are used to identify a core and how these bits are represented as the three high end hexadecimal digits of the address.

D.2.5 Hardware Platforms

The hardware platform will be based on utilising the Adapteva 16 Core Epiphany chip. The hardware platform will evolve with the research in three stages.

This research plans to use three different configurations:

- Stage 1 - Parallella Board with a single 16 core Epiphany chip
- Stage 2 - Four Parallella boards connected to produce 4x16 = 64 core array
- Stage 3 [Optional Extension] - Custom board with 16 Epiphany chips producing a 16x16 = 256 core array

D.2.6 16 Core - Parallella Board

Parallella is a development board that contains a single 16-core Epiphany chip along side a Xilinx Zynq 7010 or 7020 processor and a collection of communication ports.

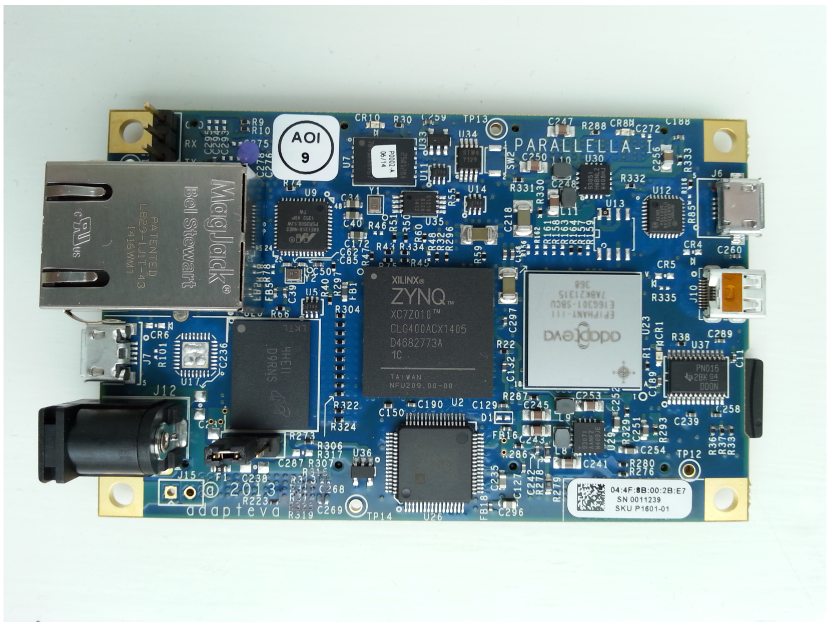


Figure D.10 – Parallella Board

The Xilinx Zynq®-7000 platform is a family of SoCs that combine Xlinx's FPGA programmable logic with a Dual ARM®Cortex^{TM}-A9 MPCore^{TM}. The ARM processor can boot Ubuntu from a mini SD card which can then be used to program the Ephiphany cores and communicate with them.

D.2.7 64 Core - 4 x Parallella Board

The Parallella development board provides connectors to the North and South sides of the Epiphany chip. The East side of the chip is connected to the Zync SoC and the West side is unconnected. A number of Parallella boards can be connected together linearly by connecting the South of one board to the North of another.

A motherboard will be designed that allows the connection of four Parallella boards, giving a total of 64 cores in a 4 x 16 mesh.

The Parallella board has very limited access to pins on the FPGA and the Epiphany Chip. In addition to linking four parallel boards together the motherboard will provide breakout the FPGA, JTAG and the Epiphany address configuration pins.

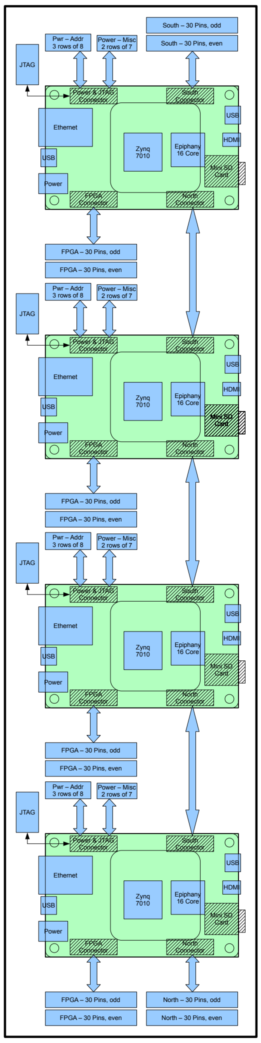


Figure D.11 – Motherboard for connection 4 Parallella boards

D.3 SpiNNaker chip

SpiNNaker is an architecture designed for the construction of massively parallel machines. Inspired by the human brain a primary goal is to provide a computing platform that can be used for research into the workings of the mind by simulating the behaviour of neurons.

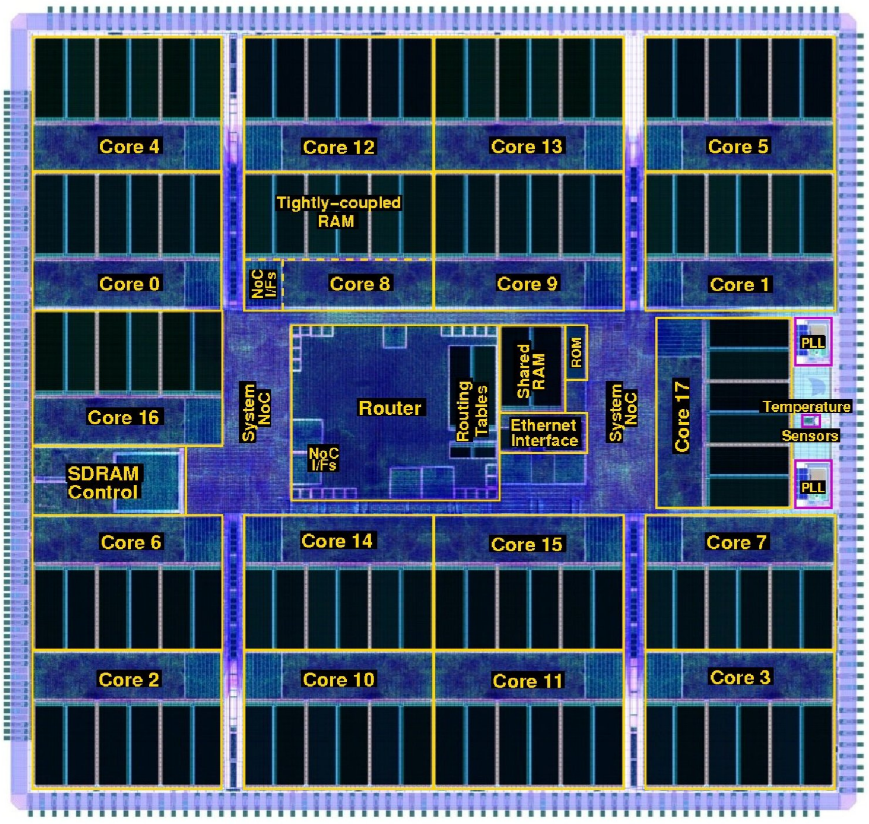


Figure D.12 – SpiNNaker chip Architecture

The core of the architecture is the SpiNNaker chip, which is described as a multicore System-on-Chip, with 18 ARM968 Globally Asynchronous Locally Synchronous (GALS) processing cores. Each SpiNNaker chip package has a 128 MB SDRAM mounted on top. SpiNNaker chips are mounted on boards of various sizes, the largest of which has 48 SpiNNaker chips, 24 of which can be mounted in a frame of a 19" rack which can accommodate 5 frames. Each 48 chip board has six 3.1Gbps high-speed serial interfaces, three of which are used to interconnect the boards into a toroidal mesh.

Of the 18 cores on a SpiNNaker chips one core is used as a monitor node and one is left spare to provide fault tolerance, leaving 16 cores for processing. A fully populated 19" rack has a total of 103,680 cores, 92,160 of which are available for processing, 5,760 are monitoring nodes and 5,760 are spares nodes.

D.4 Intel® Many Integrated Core (Intel® MIC) Architecture

In 2012 Intel launched the Intel® Xeon Phi™ Coprocessor. Mounted on a PCIe 3.0 card, is designed to be used as a slave to a conventional processor such as the Intel Core i7 CPU or Intel Xeon CPU. The top-of-the-range has 61 cores, each of which can run 4 threads concurrently giving a possible 244 concurrent threads. Multiple coprocessor boards can be installed on a single PC motherboard using the PCIe peer-to-peer interconnect to communicate without any intervention.

Each core has its own dedicated 32KB L1 instruction cache, a 32KB L1 data cache and 512 KB L2 cache which uses a global-distributed tag directory to maintain coherency. The cores are connected via a very high bandwidth ring interconnect, interleaved with memory controllers, as shown in Figure D.13.

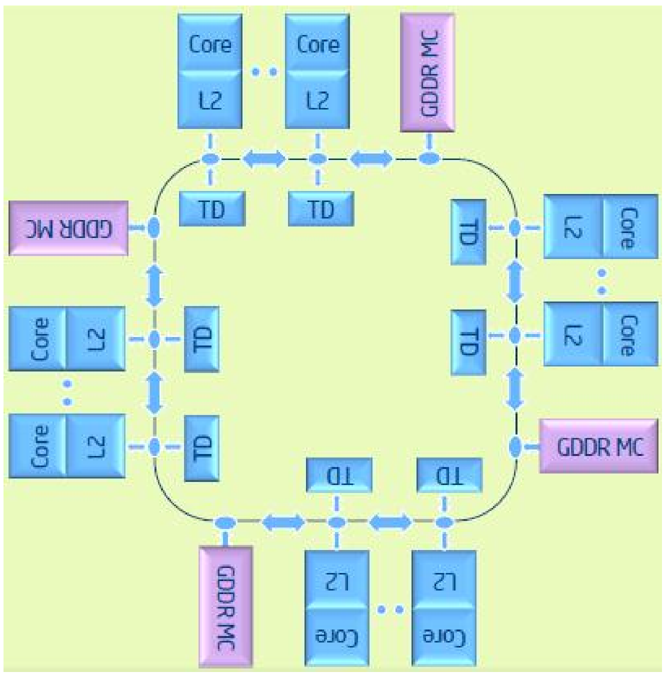


Figure D.13 – **Intel® Xeon Phi™ Architecture**

Each core also includes a vector processing unit cable of performing 16 single precision or 8 double precision operations each cycle. The Xeon Phi™ is well suited for vector and array processing but unlike GPUs the cores have dedicate cache and can be individually programmed, allowing the use of the coprocessor to be extended into areas inaccessible to GPUs. Programming for the cores is through C++ and Open MP.

Appendix E

The Many-Core Model

For ease of reference, the individual components of the many-core model described in Chapter 4 *Core Fault Tolerance*, Chapter 5 *Network Power* and Chapter 6 *Link Fault Tolerance and Network Traffic* are presented here.

E.1 The Many-Core Array

The many-core architecture is a square lattice arrangement of R rows and C columns of homogeneous computing nodes. Each *node* consists of a *router* that communicates with its nearest orthogonal neighbours and a *core* attached to the router. Communication between two nodes is via a *link*. Links are required to show the connections between nodes, however the details of the communication links between nodes is not important when considering the placement of spare cores. Communication links will be considered in more detail in the next chapter.

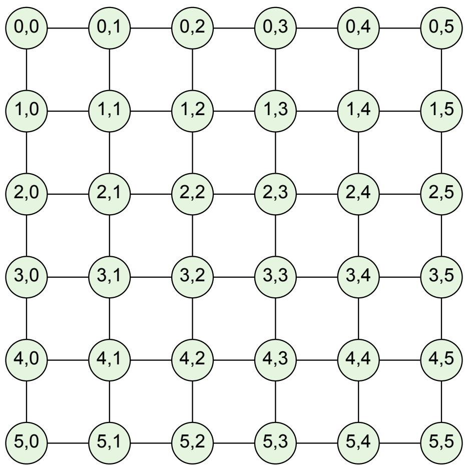


Figure E.1 – **A 6×6 many-core array showing the coordinates of each node.**

The many-core array $\mathcal{A}$ is represented as a tuple of the set $\mathcal{V}_a$ of nodes and the set $\mathcal{E}_a$ of links:

$$\mathcal{A} \coloneqq (\mathcal{V}_a, \mathcal{E}_a) \tag{E.1}$$

Given that there are R_a rows and C_a columns, the number of nodes and links are defined as V_a and E_a:

$$\begin{aligned} |\mathcal{V}_a| &= V_a \\ &= R_a C_a \end{aligned} \tag{E.2}$$

$$\begin{aligned} |\mathcal{E}_a| &= E_a \\ &= R_a(C_a - 1) + (R_a - 1)C_a \\ &= 2R_a C_a - R_a - C_a \end{aligned} \tag{E.3}$$

The set of nodes is defined as:

$$\mathcal{V}_a = \{v_1, \ldots, v_m \mid m = V_a\} \tag{E.4}$$

With a node defined as an ordered tuple consisting of the row and column coordinates of the node within the many-core array, which are also referred to as its location:

$$\begin{aligned} v_n &= (r, c) \\ &= loc \end{aligned} \tag{E.5}$$

Where:

$$loc = (r, c) \tag{E.6}$$

and:

r	Is the row coordinate of node $v_n \mid 0 \leq r < R_a$.
c	Is the column coordinate of node $v_n \mid 0 \leq c < C_a$.

Row and column coordinates are defined above as beginning at 0 and location $(0, 0)$ is arbitrarily assigned to the top-left hand core of the many-core array and location $(R_a - 1, C_a - 1)$ refers to the bottom-right hand core of the many-core array.

The set of links is defined as:

$$\mathcal{E}_a = \{e_1, \ldots, e_m \mid m = E_a\} \tag{E.7}$$

A link is defined as a set of ordered tuples consisting of a source node and a target node:

$$e_n = (s, t) \tag{E.8}$$

Where:

e_n	is a link from the set $\mathcal{E}_a \mid 1 \leq n \leq E_a$.
s	is the source node of link e_n from the set $\mathcal{V}_a \mid 1 \leq s \leq V_a$.
t	is the target node of link e_n from the set $\mathcal{V}_a \mid 1 \leq t \leq V_a$.

Since the definition of the node includes its location, the source and target nodes of a link also specify the locations of each end of the link. Figure E.1 illustrates a 6×6 many-core array showing the coordinates of the location of each node and the links between nodes.

E.2 Application Process Graph (APG)

This section extends the application process graph model given in Section 4.3 by adding source and sink nodes which model interfaces to resources external to the circuitry of the many-core system. Examples of external resources are memory systems, sensors, controllers or neighbouring many-core array regions. As before, each process or interface to an external resource, is represented as a node with data transfers between nodes represented as edges. The nodes of the revised application process graph can be one of the following types:

- a *source node*, an interface to an external source of data received by an APG process, is a node that has no inbound edges
- a *process node*, has both inbound and outbound edges
- a *sink node*, an interface that is the recipient of data from an APG process, is a node that has no outbound edges

As in Section 4.3 the graph $\mathcal{G}$ is represented as a tuple of the set of nodes, $\mathcal{V}_g$, and the set of edges, $\mathcal{E}_g$:

$$\mathcal{G} := (\mathcal{V}_g, \mathcal{E}_g) \tag{E.9}$$

The set of nodes $\mathcal{V}_g$ is now the union of three distinct sets of nodes within the application process graph: the set of source nodes $\mathcal{S}_g$, the set of sink nodes $\mathcal{K}_g$, the set of process nodes $\mathcal{P}_g$:

Each node in the APG is identified by a label: process nodes by P$n \mid 1 \leq n \leq P_g$ where P_g is the number of process nodes in the graph, sources nodes by S$n \mid 1 \leq n \leq S_g$ where S_g is the number of source nodes in the graph and sink nodes by K$n \mid 1 \leq n \leq K_g$ where K_g is the number of sink nodes in the graph.

The total number of nodes is denoted by V_g, where:

$$V = |\mathcal{V}_g| \ = S + K + P \tag{E.10}$$

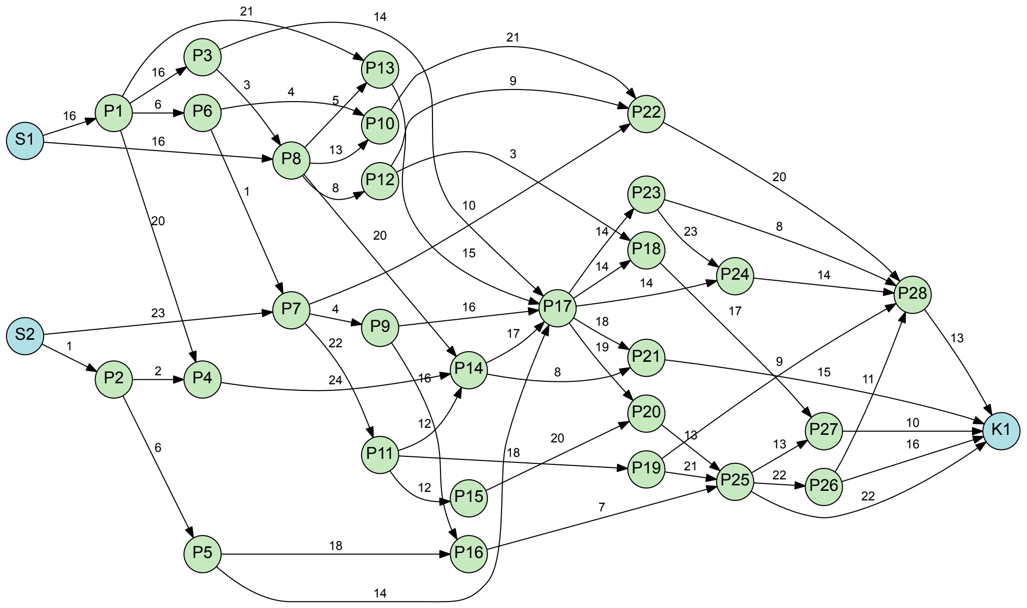


Figure E.2 – Sparsely connected graph with 2 source nodes, 28 processing nodes and 1 sink node.

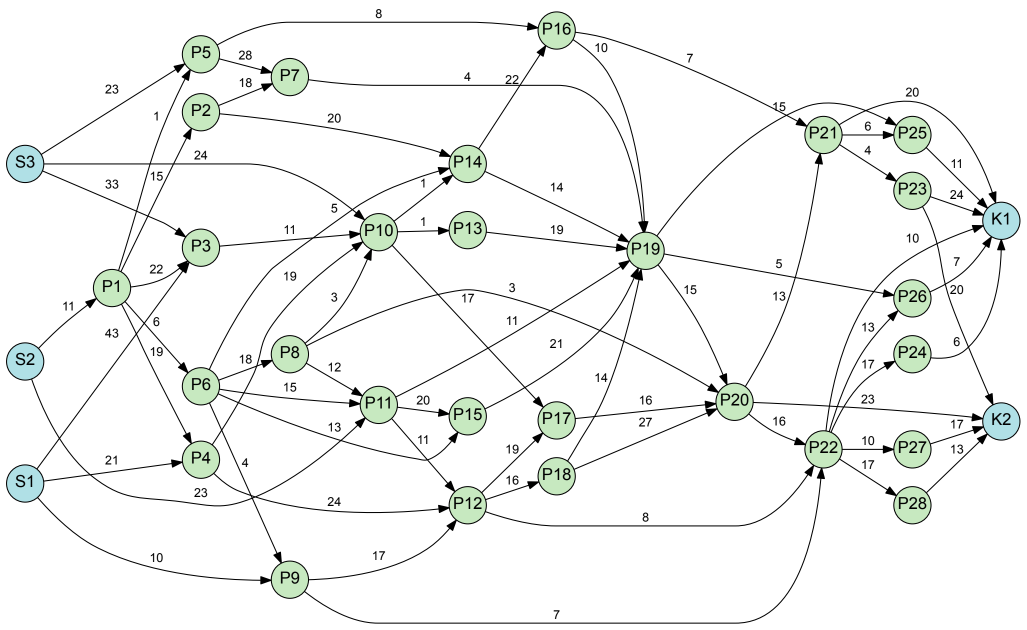


Figure E.3 – Moderately connected graph with 3 source nodes ,28 processing nodes and 2 sink nodes.

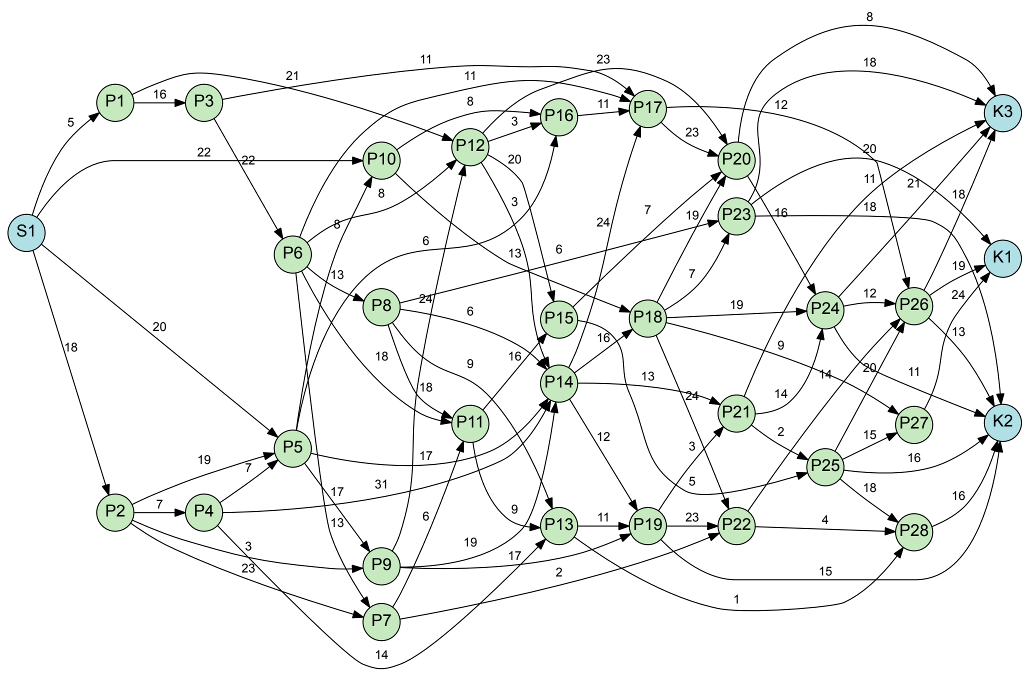


Figure E.4 – Densely connected graph with 1 source node, 28 processing nodes and 3 sink nodes.

The number of edges is:

$$E = |\mathcal{E}_g| \tag{E.11}$$

The node sets are defined as:

$$\mathcal{S}_g = \{s_1, \ldots, s_n \mid 1 \leq n \leq S\} \tag{E.12}$$

$$\mathcal{K}_g = \{k_1, \ldots, k_n \mid 1 \leq n \leq K\} \tag{E.13}$$

$$\mathcal{P}_g = \{p_1, \ldots, p_n \mid 1 \leq n \leq P\} \tag{E.14}$$

$$\mathcal{V}_g = \mathcal{S}_g \cup \mathcal{K}_g \cup \mathcal{P}_g \tag{E.15}$$

$$\mathcal{E}_g = \{e_1, \ldots, e_n \mid 1 \leq n \leq E_g\} \tag{E.16}$$

An APG can be characterised by the three numbers representing the number of process nodes, source nodes and sinks nodes; for example the graph in fig E.4 can be characterised by the three numbers $s = 1, p = 28, k = 3$ or, more compactly, by $V(1 : 28 : 3)$.

More densely connected graphs will, by definition, have more edges, which implies a greater total volume of traffic, which will affect metrics based on traffic volumes. To quantify the effect of graph density on the metrics, we categorized graphs as *sparsely connected*, *moderately connected* and *densely connected* depending on the density of edges in the graph. A sparsely connected graph is defined as a graph where there are less than two edges per node, a moderately connected graph is a graph where the number of edges is more than or equal to 2 edges per node and less than 2.5 edges per node, and a densely connected graph is a graph where the number of edges is more than or equal to 2.5 edges per node. These are arbitrary values which were found to be useful in this work.

The connection density of graphs is defined by Equation E.17

$$C_g \begin{cases} sparse, & \text{if } \mathcal{E} < 2 \cdot \mathcal{V} \\ moderate, & \text{if } (\mathcal{E} \geq 2 \cdot \mathcal{V}) \wedge (\mathcal{E} < 2.5 \cdot \mathcal{V}) \\ dense, & \text{otherwise} \end{cases} \tag{E.17}$$

Where:

C_g	=	The connectivity of graph G.

The graph density also has an impact on the number of mappings that are link fault tolerant. Each edge creates a ComPair so with more edges creates more ComPairs, and the greater number of ComPairs makes it more difficult to arrange the processes in cores of the many-core array such that, the source node and target node of each ComPair are not on the same row and column i.e.there are no critical links.

E.3 Process Map

A *process map* or *mapping* is a data structure that, for each processing node of the application process graph, gives the location of the core in the array that will run the process. Cores that are running a process are given a task name that is the process name from the application process graph. Many-core array cores that are not allocated a process are spare cores and are considered to be idle so are given a task name of 'i', while failed cores are given a task name of 'f'. Figure E.5 illustrates a 6×6 many-core array.

Figure shows a process map for the application process graph in Figure 4.2 mapped to the many-core array of Figure 4.1

The process map has the same dimensions as the many-core array, given as R rows and C columns defined in Section 4.2.

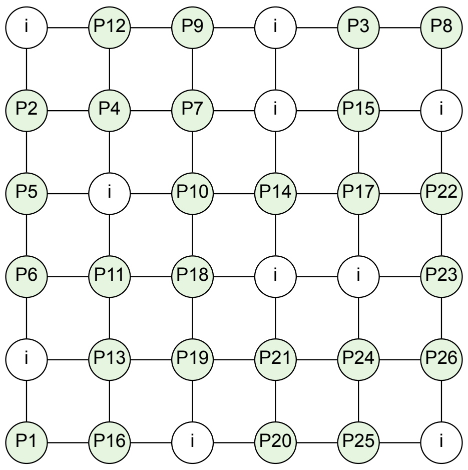


Figure E.5 – A Process Map for a 26 node APG in a 6x6 Many-Core Array

The process map $\mathcal{M}$ is represented as a set of nodes $\mathcal{V}_m$:

$$\mathcal{M} \coloneqq \mathcal{V}_m \tag{E.18}$$

The number of nodes is defined as V_m and is the same as V_a:

$$\begin{aligned} V_m &= |\mathcal{V}_m| \\ &= V_a \\ &= R_m C_m \end{aligned} \tag{E.19}$$

Given that there are R_m rows and C_m columns, the set of nodes are defined as:

$$\mathcal{V}_m = \{v_1, \ldots, v_n \mid 1 \leq n \leq V_m\} \tag{E.20}$$

A node is a set of ordered tuples consisting of a location which is the coordinates of the corresponding node in the many-core array and a process name:

$$\begin{aligned} v_m &= (r, c, p) \\ &= (loc, p) \end{aligned} \tag{E.21}$$

Where:

$$loc = (r, c) \tag{E.22}$$

and:

v_m	Is a node of the set $\mathcal{V}_m$.
r	Is the row coordinate of the node $v_m \mid 0 \leq r < R_m$, where row 0 is the top row.
c	Is the column coordinate of the node $v_m \mid 0 \leq c < C_m$, where column 0 is the left most column.
p	Is the process name of the APG process mapped to the many-core array node at location (r, c) or, if no process is mapped to the many-core array node, the value i representing a spare core $\mid p \in \mathcal{P}_g \cup \{i\}$.

The location coordinates are illustrated in Figure 4.1.

E.4 Communicating Core Pair (ComPair)

A pair of processes from an application process graph connected by an edge will cause data traffic to flow between the cores in the many-core array that the processes are mapped to. The two cores in the many-core array where the processes are located are referred to as a *communicating core pair* or *ComPair*. Since the application process graph is a directed graph each ComPair will have a *source node* from where data traffic originates and a *target node*, where the data is received for use by the target process. For this work, the assumption is made that traffic will use a path of minimal length between the source and the target of the ComPair subnet.

The ComPair is introduced in this chapter to support the calculation of the power metric presented in Section 5.3. ComPairs will also be used extensively in the calculation of traffic metrics presented in Chapter 6.

Definition: Given an edge of an application process graph and the source node and target node of the edge, a ComPair consists of the two cores of the many-core array to which the source node and target node have been mapped. The ComPair has a *subnet* that consists of all the routing nodes and links between and including the node where the source core is located and the node where the target core is located.

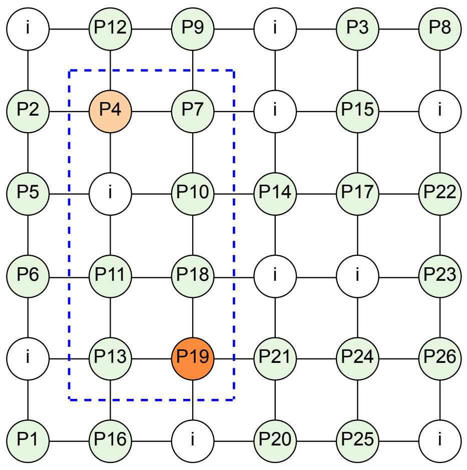
Between the source and target nodes of a ComPair there will one or more *minimal length paths*, which are paths where each step in the path gets closer to the target node. In this work metrics are based on minimal length paths.

Once an application process graph has been mapped to a many-core array there will be as many ComPairs in the array as there are edges in the application process graph. Note that a physical link of the many-core array may carry traffic from multiple ComPairs.

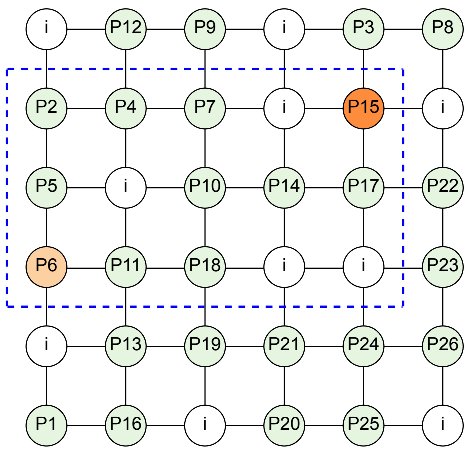
The properties of a ComPair are given in Table E.1 and those of a ComPair subnet inTable E.2.

As an example, two ComPairs that are created by mapping the application process graph of Figure 4.2 to a 6×6 many-core array, are illustrated in Figure E.6.

Note that when discussing ComPairs, they will usually be referred to by the symbol Q,



(a) Subnet for the ComPair with source node P4 and target node P19



(b) Subnet for the ComPair with source node P6 and target node P15

Figure E.6 – ComPair and subnet examples. Source nodes are coloured light orange, target nodes coloured dark orange and the subnets are bounded by a dashed blue line.

taken to mean any unspecified ComPair. When it is necessary to associate a ComPair with an edge in the application process graph the ComPair will be referred to by the symbol Q_e, where e represents the specific edge of the application process graph that generates the ComPair in the many-core array.

The number of rows and columns in a ComPair subnet can be calculated from the source and target locations using equations E.23 and E.24.

$$R_q = |t_r - s_r| + 1 \tag{E.23}$$

$$C_q = |t_c - s_c| + 1 \tag{E.24}$$

The number of links in a ComPair subnet is given by Equation E.25.

$$Q_l = R_q(C_q - 1) \; + (R_q - 1)C_q \tag{E.25}$$

The number of shortest length paths from the source node to the target node of a ComPair subnet that is fault free is given by Equation E.26 as explained in Section 6.7. If there are faulty links in the subnet then the number of links can be calculated using the recursive algorithm described in Section 6.7.

$$P_q = \frac{(R_q - 1)(C_q - 1)!}{(R_q - 1)!(C_q - 1)!} \tag{E.26}$$

The *path length* of a ComPair in terms of the number of links on a shortest length path

Table E.1 – ComPair Properties

Name	Description
Q_e	The ComPair representing edge e of the application process graph
Q_s	The source node of ComPair Q
Q_{loc_s}	The location of the source node of ComPair Q within the array
s_r	The row of the source location
s_c	The column of the source location
Q_t	The target node of ComPair Q
Q_{loc_t}	The location of the target node of ComPair Q within the array
t_r	The row of the target location
t_c	The column of the target location
R_q	The number of rows of nodes in ComPair Q
C_q	The number of columns of nodes in ComPair Q
Q_l	The path length of the ComPair in terms of the number of links on the shortest path between the source and target nodes
P_q	the number of shortest length paths between the source and target nodes
Q_d	The network load for the ComPair taken from d_n for the edge e_n of the application process graph that the ComPair represents

between the source and target nodes of is given by Equation E.27.

$$P_q = (R_q - 1) + (C_q - 1) \tag{E.27}$$

Table E.2 – ComPair Path and Link Properties

Name	Description
$\mathcal{P}_q$	The set of unique shortest length paths between the source and target nodes of the ComPair Q
P_q	The number of shortest length paths in the set $\lvert\mathcal{P}_q\rvert$
p_n	The n^{th} path of ComPair Q
$\mathcal{L}_{(q)}$	The set of links in the subnet of ComPair Q
L_q	The number of links in set $\lvert\mathcal{L}_{(q)}\rvert$ in the subnet of ComPair Q
l_n	The n^{th} link of ComPair Q

E.5 Link Criticality

The many-core array architecture described in Section 4.2 uses a lattice of routers for transmission of traffic through the network. Traffic from the source to the target of a ComPair will be transmitted as a number of packets which will pass through a series of routers, each with a buffer where received packets are temporarily stored before being forwarded to the next router.

Link criticality is a method of categorizing the effect that the failure of a physical link can have on traffic in a ComPair. The classification of a link is specific to a ComPair, so can have different criticalities for each ComPair subnet that it is part of. A link is classified as one of the three following types defined by the effect its failure will have on the transmission of data in the ComPair subnet of which is is part:

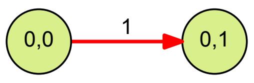
- Critical Links
- Significant links
- Normal links

Each of these three types of links will be explained below with reference the the graphs in Figure E.7. When developing the metrics it will become apparent that the number of paths that use each link will be important in the calculation of the metric. The edges of the graphs in Figure E.7 are labelled with the number of paths that use each link.

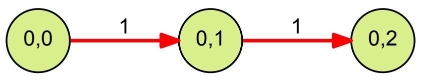
Critical Link
A *critical link* is a link where all paths from the source to the target pass through the link, such that a fault will cause complete failure of communication. Figures E.7a and E.7b illustrate subnets with critical links, shown in red, where the failure of a single link causes complete communication failure. Traffic that passes through a critical link is described as *critical traffic*.

If, for ComPair Q, there are P_q paths between the source and target nodes and there are l_{n_p} paths through link l_n, then link l_n is defined as a critical link by Equation E.28, i.e. when the number of paths through the link is equal to the total number of paths between the source node and target node of the ComPair.

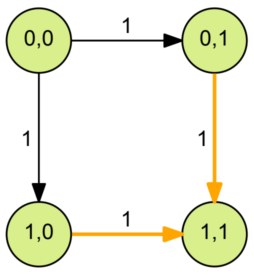
$$l_n \text{ is critical } \Leftrightarrow l_{n_p} = P_q \tag{E.28}$$



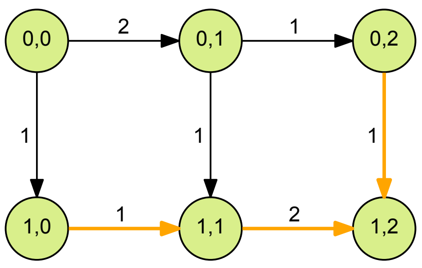
(a) Critical Link of length 1



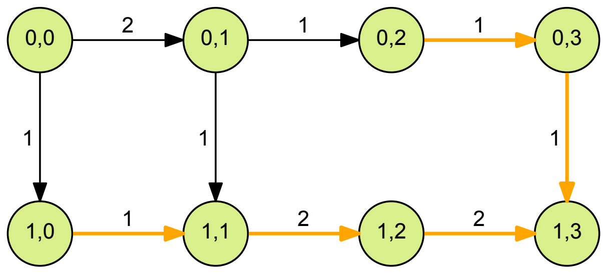
(b) Critical Link of length 2



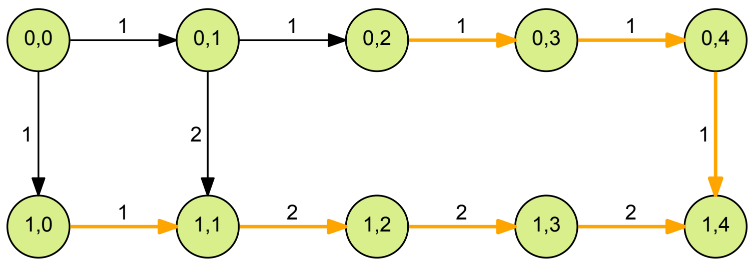
(c) Network with 2 Significant Links



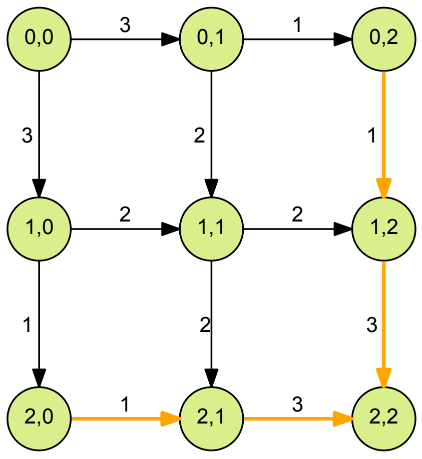
(d) Network with 3 Significant Links



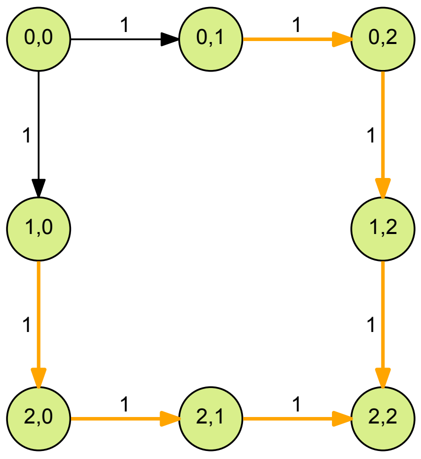
(e) Network with a single Link fault and 5 Significant Links



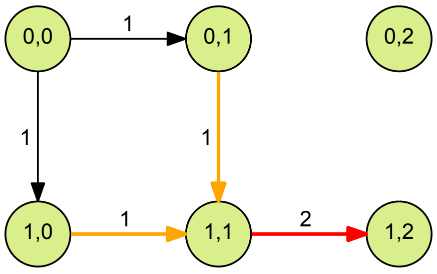
(f) Network with a 2 Link faults and 7 Significant Links



(g) Network with 4 Significant Links



(h) Network with a 4 Link faults and 6 Significant Links



(i) Network with 2 Link faults, 2 Significant Links and 1 Critical Link

Figure E.7 – Networks illustrating Critical and Significant links

Where:

P_q	=	The total number of paths of the ComPair subnet.
l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
l_{n_p}	=	The number of paths through link l_n of ComPair Q.
Q_l	=	The number of links in the ComPair subnet.

Significant Link

A *significant link* is a link which is the only link from its source node and is a link where only a (proper) subset of paths from the source to the target pass through the link i.e. not all paths use pass through the link, as defined by Equation E.29:

$$l_n \text{ is significant } \Leftrightarrow (l_{n_p} < P_q) \wedge (l_{s_{out}} = 1) \tag{E.29}$$

Where:

P_q	=	The total number of paths of the ComPair subnet.
l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
l_{n_p}	=	The number of paths through link l_n of ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
$l_{s_{out}}$	=	The number of outbound edges of source node l_s
l_s	=	The source node of link l

A faulty significant link will sever the paths that use the link while still leaving one or more viable paths that do not use the faulty link. Traffic that passes through a significant link is described as *significant traffic*. A fault in a significant link will potentially cause some packets to be lost where packets are stored in the buffers of routers that have become disconnected from the target node due to the appearance of the fault. Figures E.7c and E.7d illustrate subnets where there are significant links shown by orange arrows. Packets will be lost when the routing node attempts to transmit packets through the link after the link has failed.

A significant link exists when it is on the only path from its routing node to the target node of the ComPair but the path it is on is not the only path between the ComPair. Failure of a significant link will create a dead end path of arbitrary length causing the loss of packets that travel down the "dead end". A router only has knowledge of the status of directly connected links so each router will continue to route packets down the dead end until the loss of packets is detected and propagated back up the path until the router at the head of the dead end path is reached. The router at the head of the dead end will eventually establish that packets are not being received through that path so will modify its behaviour to only send packets down the alternative path, thereby maintaining communication. The loss of a significant link will redistribute traffic to other links which may in turn create bottlenecks and reduce the overall performance of the system. In some case, for example in figures E.7 (c), (d), (e), (f), (h) and (i), bit not (g), the loss of a significant link will have the effect of changing the criticality of other links from significant to critical, making the network vulnerable to further link failure.

Notice that a ComPair subnet will contain as least one critical link or two significant links. This can be confirmed by considering the target node: if there is only one inbound link to the target node then it must be a critical link, if there are two inbound links to the target node then they must both be significant links since failure of either link will cause packet loss.

Normal Link

A *Normal Link* is a link whose failure does not cause the loss of transmission of packets, more specifically it is a link whose routing node has more than one outgoing link to the target of the ComPair. A normal link is defined by Equation E.30:

$$l_n \text{ is significant } \Leftrightarrow (l_{s_{out}} = 2) \tag{E.30}$$

Where:

l_n	=	The l^{th} link of the Q_l links in the ComPair subnet.
$l_{s_{out}}$	=	The number of outbound edges of source node l_s
l_s	=	The source node of link l

Traffic that passes through a normal link is described as *normal traffic*. Figures E.7c to E.7i illustrate subnets where normal links are shown by black arrows.

Failure of a normal link has no negative effect on the transmission of packets because the routing node it is connected to has a choice of two links for the transmission of packets. Given that the router has knowledge of the status of its directly connected links it is able to transmit packets along the remaining healthy link when one link fails. Each of the two links are on distinct, alternative paths from the source to the destination.

E.6 Hardware Map

A *hardware map* or *network topology map* models, for a single region, the cores and links of the many-core array and the interfaces that connect the many-core array to external resources or neighbouring regions and the status of each element of the map.

The purpose of the hardware map is to model the essential elements of the many-core array and interfaces to external resources in sufficient detail to enable valid mappings to be generated and the metrics of the mappings to be calculated.

E.6.1 Nodes and Links of a Hardware Map

The definition of the hardware map is similar to the many-core array defined in Section 4.2 with the addition of information defining the type of each node and the status of each core and link.

The hardware map must maintain information to:

- Specify the type of each node
- Specify the status of each node and link
- Uniquely identify each node and link

Nodes
Nodes represent the cores of a many-core array or interfaces to resources external to the many-core array or cores of neighbouring regions. Interfaces to external systems can be classified as capable of acting as both sources or sinks, sources only or sinks only. The node types are summarised in Table E.3.

Table E.3 – Hardware Map Node Types

Type		Description
c	Core:	A node that can be used by an APG process node.
s	Source:	A node that can be used only by an APG source nodes.
k	Sink:	A node that can be used only by an APG sink nodes.
b	Both:	A node that can be used by an APG source or sink node.
r	Region:	A node that is either shared with or belongs to a neighbouring region.

Links
All links are unidirectional and are identical whether they are part of the internal circuitry of the many-core array or a link between the many-core array and external interfaces or regions.

Node Identity
Each node is given a location (c, r) starting with coordinate $(0, 0)$ at the top left most corner as illustrated in Figure 4.1.

Link Identity
Each link is directional so is identified by the location of the source node followed by the location of the target node $(loc(s), loc(t))$.

E.6.2 Modelling Faults

Node and Link Status
Each node and link in the hardware map has a status which indicates if the element is in good working condition or faulty as summarised in Table E.4

Table E.4 – Core and Link Status Values

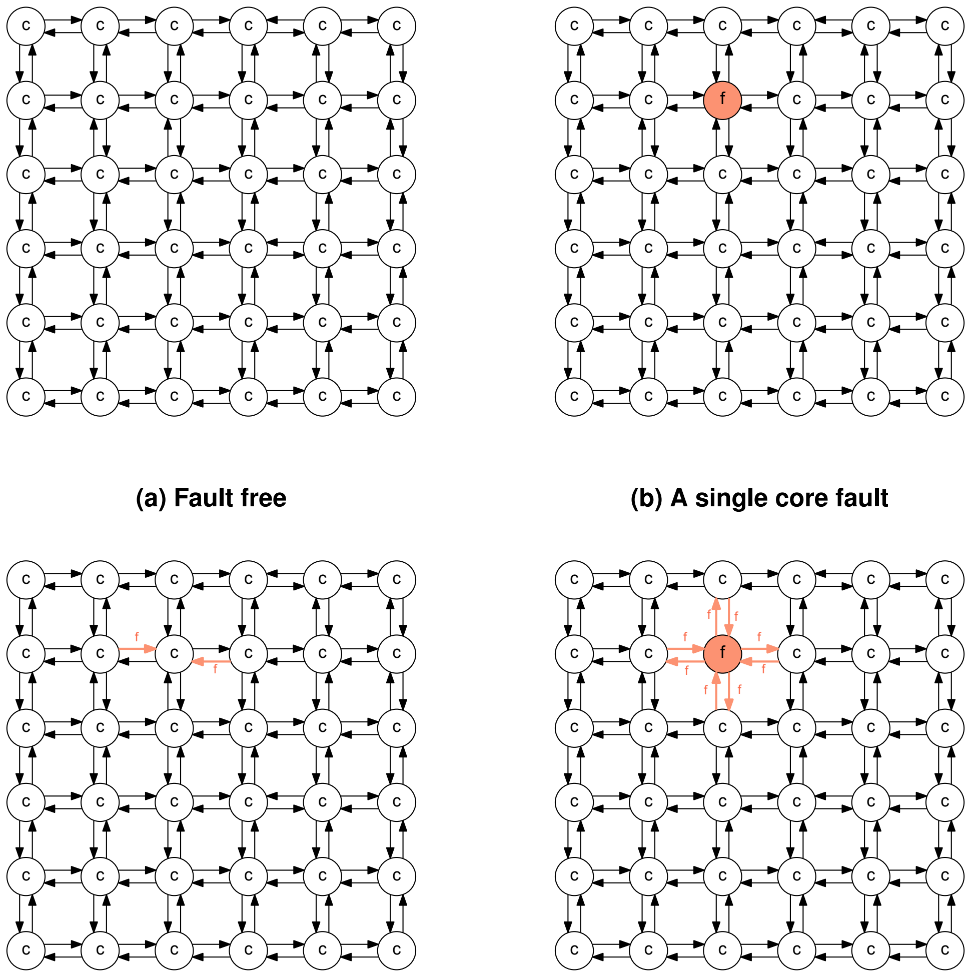
Status		Description
g	Good:	The element is fully functioning.
f	Faulty:	The element is faulty and cannot be used.

Using the status of nodes and links the hardware map can model:

- Processing Core Faults
- Link Faults
- Routing Node Faults

A fault free hardware map and three types of hardware faults are illustrated in the hardware maps of Figure E.8 with the faulty nodes and links being marked ***f*** and highlighted in red.

Processing Core Faults
Figure E.8b illustrates a hardware map with a single faulty processing core, while the



(a) Fault free **(b) A single core fault**

(c) Two link faults **(d) A single routing node fault**

Figure E.8 – Hardware Maps representing a 6×6 many-core array with interfaces to external resource on each of the four edges of the many-core array.

routing node and all links are functioning normally. Traffic passing though the routing node is unaffected.

Link Faults

Figure E.8c illustrates a hardware map with two faulty links that are unable to carry any traffic.

Routing Node Faults

Figure E.8d illustrates a hardware map with a single faulty routing node, which is modelled by marking the node, and all 8 adjacent links, as failed. If the routing node fails then the processing node becomes unavailable because it cannot receive or transmit any data; this allows the routing node to be modelled as a combination of a processing node failure together with the failure of all links attached to the routing node which is a considerable simplification compared to modelling the processing node and routing node separately.

E.6.3 Hardware Map Definition

The hardware map $\mathcal{H}$ is represented as a tuple of the set $\mathcal{V}_h$ of nodes and the set $\mathcal{E}_h$ of links:

$$\mathcal{H} \coloneqq (\mathcal{V}_h, \mathcal{E}_h) \tag{E.31}$$

Given that there are R_h rows and C_h columns, the number of nodes and links are defined as V_h and E_h:

$$\begin{aligned} |\mathcal{V}_h| &= V_h \\ &= R_h C_h \end{aligned} \tag{E.32}$$

$$\begin{aligned} |\mathcal{E}_h| &= E_h \\ &= R_h(C_h - 1) + (R_h - 1)C_h \\ &= 2R_hC_h - R_h - C_h \end{aligned} \tag{E.33}$$

The set of nodes is defined as:

$$\mathcal{V}_h = \{v_1, \ldots, v_m \mid qm = V_h\} \tag{E.34}$$

with a node defined as an ordered tuple consisting of the row and column coordinates of the node within the many-core array, which is also its location:

$$v_n = (loc, t, s) \tag{E.35}$$

Where:

$$loc = (r, c) \tag{E.36}$$

and:

r	is the row coordinate of node v_n.
c	is the column coordinate of node v_n.
t	is the type of the node $v_nt \mid t \in \{c, b, s, k, r\}$ as defined in Table E.3.
s	is the status of the node $v_n \mid s \in \{g, f\}$ as defined in Table E.4.

Row and column coordinates are arbitrarily defined as beginning at 0 and location $(0, 0)$ referring to the top-left hand core of the many-core array with location $(R_h - 1, C_h - 1)$ referring to the bottom-right hand core of the many-core array.

The set of links is defined as:

$$\mathcal{E}_h = \{e_1, \ldots, e_m \mid m = E_h\} \tag{E.37}$$

A link is defined as a set of ordered tuples consisting of a source node location, a target node location and a status:

$$e_n = (loc_s, loc_t, s) \tag{E.38}$$

Where:

e_n	is a link from the set $\mathcal{E}_h$.
loc_s	is the location of the source node of link e_n.
loc_t	is the location of the target node of link e_n.
s	is the status of the link $e_n \mid s \in \{g, f\}$ as defined in Table E.4.

E.6.4 Hardware Map Configurations

The hardware map is designed to be flexible so that it can model a range of configurations, a selection of which are illustrated in Figure E.9. Configurations are defined through the use of the set of parameters listed in Table E.5.

Table E.5 – **Hardware Map Definition Parameters**

Parameter	Description
R_h	The number of rows of nodes in the hardware map
C_h	The number of columns of nodes in the hardware map
R_a	The number of rows of nodes in the many-core array
C_a	The number of columns of nodes in the many-core array
R_o	The position of the first of the row of the many-core array relative to the first row of the hardware map
C_o	The position of the first of the column of the many-core array relative to the first row of the hardware map
B_n	The border type of the north edge, see Table E.6
B_w	The border type of the west edge, see Table E.6
B_e	The border type of the east edge, see Table E.6
B_s	The border type of the south edge, see Table E.6
H_{df}	A dataflow machine is modelled when this boolean switch is set to ***true***
$L_{(h)}$	The set of links in the hardware map

Border types are used to define how the nodes on the rows and columns, that are on the edges of the hardware map are configured. The parameters allow for borders to have of multiple rows or columns, which allow for a variety of multi-region arrangements. The types of borders are listed in Table E.6.

Table E.6 – Border Types

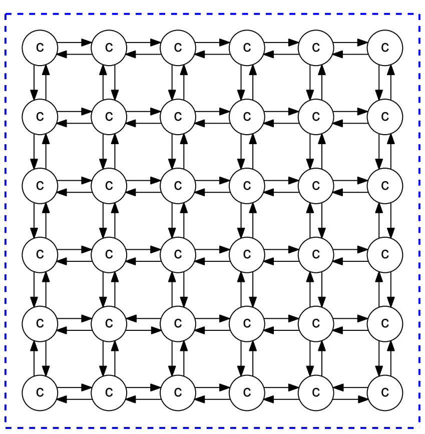
Type		Description
s	Source:	A node that can be used only by an APG source nodes.
k	Sink:	A node that can be used only by an APG sink nodes.
b	Both:	A node that can be used by an APG source or sink node.
r	Region:	A node that is either shared with or belongs to a neighbouring region.
n	None:	There is no border.

A hardware map has four borders which are described using the four compass points of north, east, south and west. The four borders can be represented as a list of border types and width pairs, starting with the north border and working clockwise. For example, a hardware map that has a single row/column on each side that can be used for both sources and sinks can be described as $B((b,1),(b,1),(b,1),(b,1))$. If there is no border on one of the edges then the border is described using the border type and width pair of $(n,0)$.

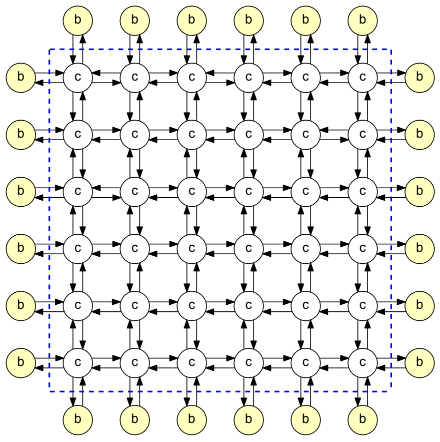
A selection of possible hardware map configurations are presented in Figure E.9, each showing a 6×6 array of processing cores with different combinations of border types. The four example configurations are described below.

To fully specify a configuration, the following information is required:

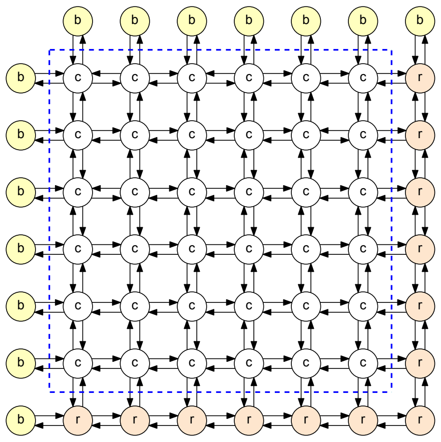
- The number of rows and columns of the hardware map.
- The number of rows and columns of the many-core array.
- The offset row and column of the many-core array with respect to the top-left corner of the hardware map.
- The type and width of the border of each edge of the hardware map.



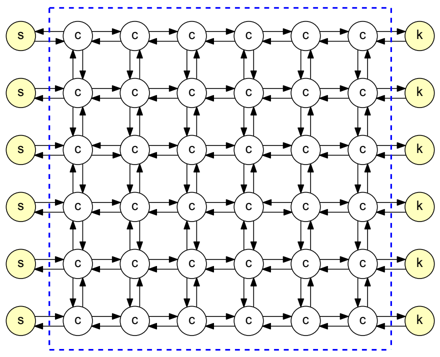
(a) A detached 6x6 Hardware Map with no source or sinks: $B((n,0),(n,0),(n,0),(n,0))$.



(b) A connected 8x8 Hardware Map were all borders can be either sources or sinks: $B((b,1),(b,1),(b,1),(b,1))$.



(c) A multi-region 8x8 Hardware Map for a with north and west borders that can be either sources or sinks and east and south borders that are neighbouring regions: $B((b,1),(r,1),(r,1),(b,1))$.



(d) A 6x8 Hardware Map for a Dataflow Machine with no borders on the north and south edges, sources on the west border and sinks on the east border: $B((n,0),(k,1),(n,0),(s,1))$.

Figure E.9 – Hardware Maps for a Variety of Configurations. The nodes inside the blue dotted lines represent the cores of the many-core array while the nodes outside of the blue dotted box represent interfaces to resource external to the many-core array and cores of adjacent regions.

Configuration for Detached Many-Core Array
The many-core arrays being modelled in Figure E.9a is defined as a *detached* many-core array with a 6×6 array of nodes with no external data sources or sinks, or neighbouring regions. Although this is an unrealistic configuration for practical systems, it was useful for the initial experiments in chapters 4 and 5. Since the configuration is modelling only cores and links of the many-core array, the hardware map and the many-core array have the same dimensions and the application process graph must consist only of processing cores. This hardware map has a border definition of $B((n,0),(n,0),(n,0),(n,0))$ and is the model used is most literature on the subject of fault tolerance in many-core arrays.

Configuration for Connected Many-Core Array
The hardware map configuration in Figure E.9b is modelling a 6×6 *connected* many-core array which has sources and sinks to external resources but does not have any neighbouring many-core arrays or regions. The many-core array is located in the inner 6×6 nodes of the hardware map with source and sink communication ports to external resources modelled by the outside edges of the 8×8 hardware map. This hardware map has a border definition of $B((b,1),(b,1),(b,1),(b,1))$.

The connected configuration will be used in the majority of the experiments of this chapter.

Configuration for Multi-Region Many-Core Array
The hardware map configuration shown in Figure E.9c models a many-core array that is part of a *multi-regio*n arrangement many-core arrays. The north and west edges have a border type of b indicating the the nodes can be either sources or sinks while the east and south edges have a border type of r to indicate that the nodes are either under shared control with a neighbouring region or under the exclusive control of a neighbouring region. This hardware map has a border definition of $B((b,1),(r,1),(r,1),(b,1))$.

Configuration for Dataflow Many-Core Array
The hardware map in Figure E.9d is an arrangement that forces all source nodes to be located on the left edge of the array and all sink nodes to be located on the right edge of the array. This arrangement will ensure that all data arrives via the west edge of the array and leaves via the east edge, producing a flow of data from west to east, hence the name *dataflow machine*. This hardware map has a border definition of $B((n,0),(k,1),(n,0),(s,1))$.

E.7 The Environment

The application process graph is a model of the application processes and its sources and sinks. The hardware map is a model of the many-core array and the connections between the many-core array and external resources. A process map is the mapping of process nodes from the application process graph to cores in the many-core array.

Since the process map only includes process nodes from the application process graph, an additional mapping is required to map the application process graph source and sinks nodes and nodes belonging to adjacent regions; this is the role of the environment.

This section describes the environment, and the relationship between the application process graph, the hardware map, the environment and the process map.

The Environment Mapping

The environment maps the resources that are external to the many-core array to the borders defined in the hardware map. Application process graph source nodes can be mapped to nodes in borders of type b or s while sink nodes can be mapped to nodes of borders of type b or k and nodes of neighbouring regions are mapped to nodes of borders of type r.

The environment mapping is established at the beginning of an evolution and remains fixed for the duration of the evolution. The calculation of the metrics of process maps that are generated during the evolution are affected by the position of resources in the environment map and are therefore calculated with respect to the environment mapping.

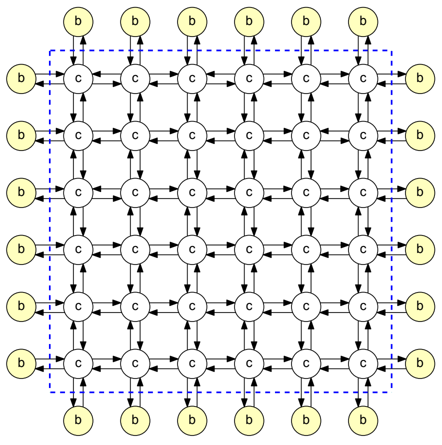
From the hardware map in Figure E.10a, that is a connected many-core array with a border of $B((b,1),(b,1),(b,1),(b,1))$ and the application process graph of Figure 6.2, the environment map of Figure E.10b and process map of Figure E.10c are created, the environment map containing the source and sink nodes from the APG and the process map contain for the processing nodes of the APG. The illustrations of the hardware map, the environment map and the process map all include a dotted blue line, the *many-core array boundary*. Inside the boundary are the nodes relating to the many-core array, while the nodes outside of the boundary relate to the environment.

The environment map consists only of the nodes outside of the many-core array boundary while the process map consists only of the nodes inside the boundary.

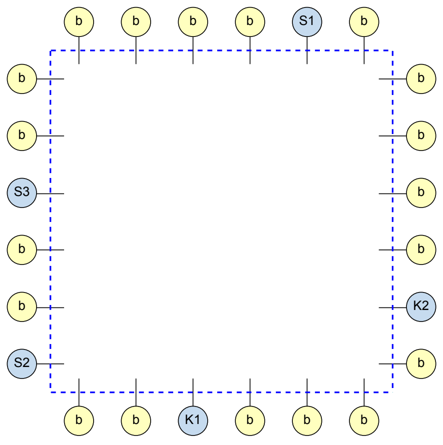
The environment and process maps only tell us which cores in the hardware map each node of the APG is mapped to but do not have any information regarding the status of links between the nodes. In the illustrations of environment and process maps, the lines linking the nodes are included to illustrate that the nodes are connected by links. The status of links is maintained in the hardware map.

The Aggregate Map

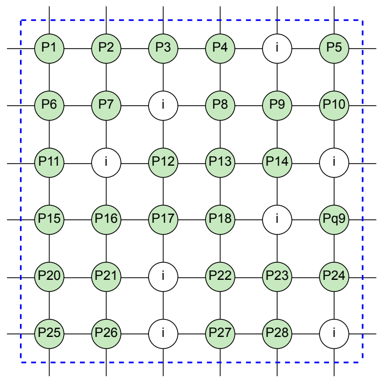
The calculation of metrics for a process map can only be made within the context of the environment. The metrics are therefore calculated using an *aggregate map* which is a combination of the environment map and process map and illustrated in Figure E.10d.



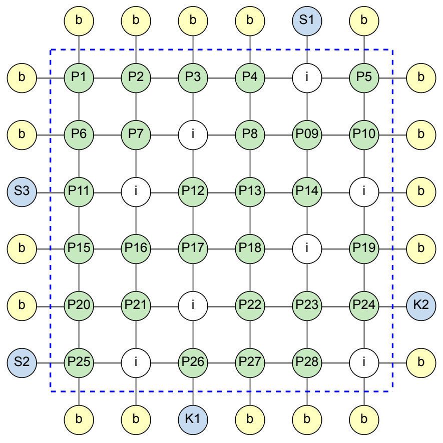
(a) An 8x8 Hardware Map where all borders can be either sources or sinks.



(b) An environment map of the source and sink nodes.



(c) A process map of the processing nodes.



(d) An aggregate map which is the combination of environment map (b) and process map (c).

Figure E.10 – From the information in the APG in Figure 6.2 and the hardware map (a) the environment map (b) and process map (c) are produced. The environment map and process maps are then combined to make the aggregate map (d) used to calculate the metrics of the process map.

Appendix F

Metrics and Objectives

For ease of reference, the metrics and objectives described in Chapter 4 *Core Fault Tolerance*, Chapter 5 *Network Power* and Chapter 6 *Link Fault Tolerance and Network Traffic* are presented here.

F.1 Metrics and Objectives

Metrics are measurements of fundamental properties of the system that is being studied. Here we are interested in measuring properties of a mapping of an application process graph onto a many-core array. Objectives are used by a search algorithm to make a comparative measure of fitness between mappings and often use a metric in the calculation of the value of the objective.

For example, the *rectilinear distance* between two nodes in a many-core array is the sum of the number of horizontal and vertical edges between the two nodes; this is a metric. A corresponding objective could be to minimize the sum of the rectilinear distance between all pairs of communicating cores. In this case, the metric measures the actual distance for a single pair of communicating cores, while the objectives uses the metric to calculate a single value for all pairs of communicating cores in the mapping.

This thesis will present four objectives along with their supporting metrics. These are:

- Core fault tolerance
- Link fault tolerance
- Network power
- Excess traffic

This is only a selection of possible objectives for optimization of mappings for many-core arrays selected to illustrate the work of this thesis.

The core fault tolerance objective is the subject of this chapter. The other objectives will be presented in subsequent chapters.

F.2 Core Fault Tolerance Metric and Objective

Metrics are measurements of fundamental properties of the system that is being studied. Here we are interested in measuring properties of a mapping of an application process graph onto a many-core array. Objectives are used by a search algorithm to make a comparative measure of fitness between mappings and often use a metric in the calculation of the value of the objective.

For example, the *rectilinear distance* between two nodes in a many-core array is the sum of the number of horizontal and vertical edges between the two nodes; this is a metric. A corresponding objective could be to minimize the sum of the rectilinear distance between all pairs of communicating cores. In this case, the metric measures the actual distance for a single pair of communicating cores, while the objectives uses the metric to calculate a single value for all pairs of communicating cores in the mapping.

F.2.1 Core Fault Tolerance

Problem Description
This section develops the metric for measuring the distance between cores and the objective, which makes use of the metric, that is used by the evolutionary algorithm.

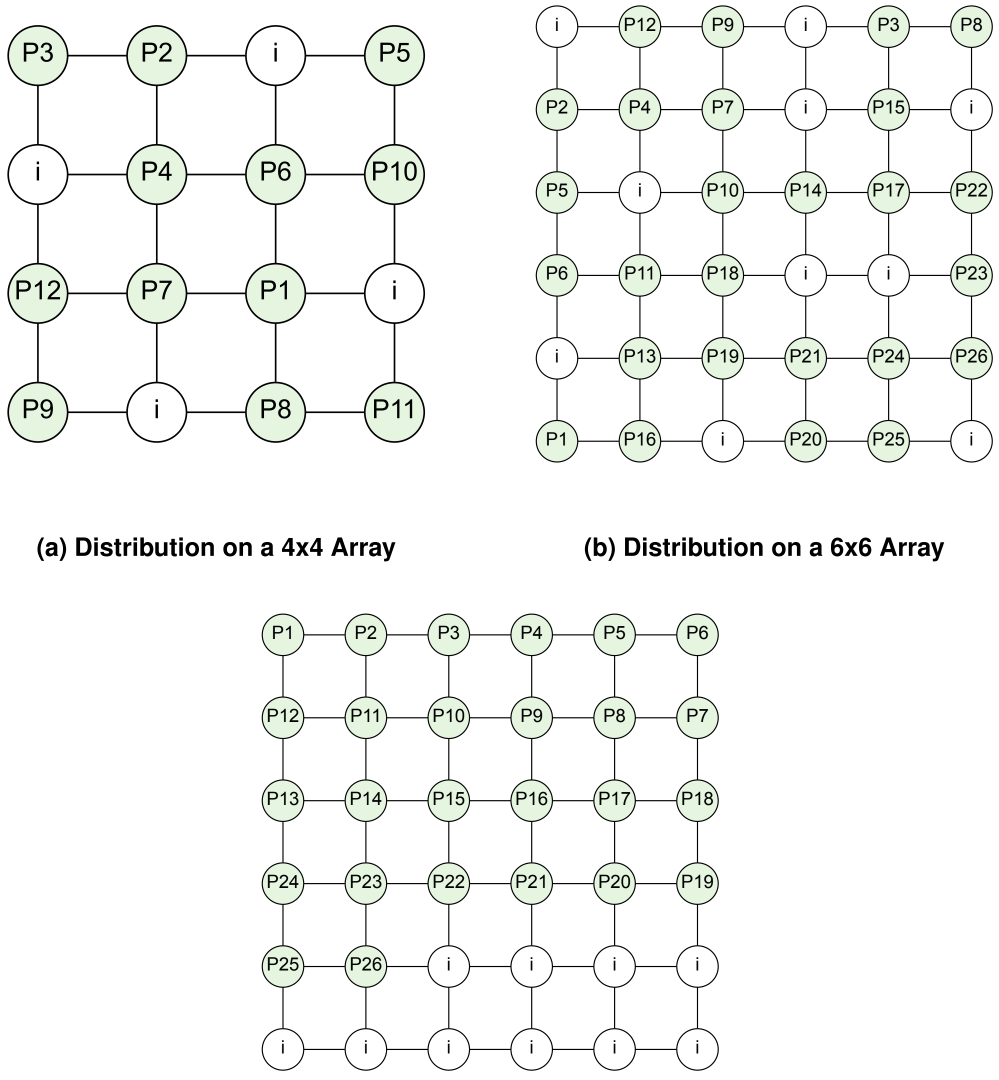
Given a fault free many-core array with R rows and C columns and an application process graph with V_a processes where $V_a < RC$, arrange the spare cores to minimize the cost of task migration in the event of the failure of a processing core.

The cost of task migration will be defined explicitly in Section 4.5.

F.2.2 Distance to Nearest Idle Core Metric

The distance to nearest idle core metric will be a measure of the distance between an individual processing core and its nearest idle core. As stated in Section 4.2, given an array of R rows and C columns, each node's location can be given by a tuple, (r, c) representing the row r and column c of the node relative to the top left-hand corner of the array. The *distance to nearest idle core* metric, $Mdnic_{(r,c)}$, is the distance between a processing core at location $p_{loc} = (p_r, p_c)$ and its nearest idle core at location $i_{loc} = (i_r, i_c)$ and is defined as the rectilinear distance between the location of the processing core and the location of the idle core, given by Equation F.1.

$$Mdnic_{(r,c)} = |(p_r - i_r)| + |(p_c - i_c)| \tag{F.1}$$



(a) Distribution on a 4x4 Array

(b) Distribution on a 6x6 Array

(c) Mapping on a 6x6 array with weak core fault tolerance

Figure F.1 – Examples of Distribution of Spare Cores

Where:

$Mdnic_{(r,c)}$	=	The distance to nearest idle core metric for the process located at (r, c).
p_r	=	The row of the location of the process.
p_c	=	The column of the location of the process.
i_r	=	The row of the location of the nearest idle core.
i_c	=	The column of the location of the nearest idle core

The minimum value of the nearest idle core metric is 1 which is the distance when a processing core is adjacent to an idle core.

F.2.3 Core Fault Tolerance Objective

The fault tolerance objective is to minimize the sum of the distances between each processing core and its nearest idle core. When an idle core is adjacent to a processing core, i.e. one step away, the fault tolerance objective value is defined as zero. When t steps are required to reach the nearest idle core the fault tolerance objective value will be $t-1$ (In other words, to obtain an adjacent idle core objective value of zero the metric value of each process-idle core pair is reduced by one). This is an arbitrary choice made to ensure that the lowest possible fault tolerance value for any array size is zero.

For example, Figure F.1 (a) and (b) illustrate respectively, for 4×4 and 6×6 arrays, arrangements of idle cores among processing cores where every processing core is adjacent to at least one idle core, while Figure F.1c illustrates a mapping which has weak fault tolerance due to some processing cores having no adjacent idle core. The mapping F.1a and F.1b, are assigned an objective value of zero, by the objective function, as no other distribution of idle cores is considered to be any better for the purposes of core fault tolerance.

A fault tolerance cost of zero for the task map as a whole indicates that each processing core is adjacent to at least one idle core. A processing core that has more than one adjacent idle core is not regarded as having any additional benefit from the additional adjacent idle cores. The core fault tolerance objective $Jcore$ is given as:

$$Jcore = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} \begin{cases} Mdnic_{(r,c)} - 1, & \text{if core}(r,c) \text{ is a processing core} \\ 0, & \text{otherwise} \end{cases} \tag{F.2}$$

F.3 Link Fault Tolerance Metric and Objective

Failure of a link can result in disruption to communication in the array by causing packets to take longer routes, lose packets or sever communications completely. The exact effect a failed link has on communications of the array depends of how critical the link is for each ComPair that uses the link. In this section link failures will be explored and a metric developed that is designed to measure the detrimental effect the failure of a link would have on the network.

The metrics developed in this section relate to a single ComPair, a ComPair. The objective combines the metrics for all ComPairs into a single value representative of the vulnerability of the whole mapping to link failure.

F.3.1 Problem Description

Develop a metric that measures the effect that failure of a link will have on network traffic and an objective that can be used to minimize the negative effects of possible link failures.

The definition of the metrics in this section refers to discussion of link criticality and the accompanying ComPair subnets in Section 5.2 *Link Criticality*. For convenience the ComPair subnets are reproduced here as Figure F.2.

F.3.2 Vulnerability to Critical Link Failure Metric

A failed critical link causes communication failure because it is used by all paths between the source and target nodes of the ComPair . Comparing figures F.2a and F.2b, it can be observed that the graph of Figure F.2a has a single path consisting of a single critical link and the graph of Figure F.2b has a single path consisting of two critical links. In this respect the graph of Figure F.2b can be described as being twice as vulnerable to critical link failure when compared to Figure F.2a. It is also possible for a critical link to have multiple paths using it as shown in Figure F.2i. Failure of such a link will sever all the paths that pass through it so the metric must reflect the number of paths affected by the link.

The number of paths of a ComPair subnet that include a link l_n is given by l_{n_p} and by Equation 5.6 a link is critical when $l_{n_p} = Q_p$. The metric $Mvclf_c$ is a measure of the *Vulnerability to Critical Link Failure* of a ComPair, is defined as the sum of the number of paths that pass through each of the critical links in the ComPair subnet. Metric $Mvclf_c$ is given by Equation F.3 which include the equivalence that makes use of the fact that all critical links, by definition, must be included in all paths of the ComPair.

$$Mvclf_c = \sum_{l=1}^{Q_l} \begin{cases} l_{n_p}, & \text{if link } l_{n_p} = Q_p \text{ i.e. is a critical link} \\ 0, & \text{otherwise} \end{cases} \equiv Q_p \cdot l_c \tag{F.3}$$

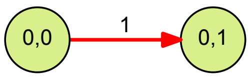
Where:

$Mvclf_c$	=	The vulnerability to critical failure metric for the ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
Q_p	=	The total number of paths of the ComPair subnet.
l_c	=	The number of critical links in a ComPair

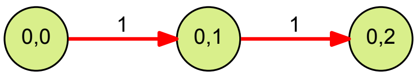
Equation F.3 will give a value of 1 for the ComPair illustrated in Figure F.2a and give a metric value of 2 for the ComPair illustrated Figure F.2b which is consistent with the notion that F.2b is twice as vulnerable as F.2a.

F.3.3 Vulnerability to Significant Link Failure Metric

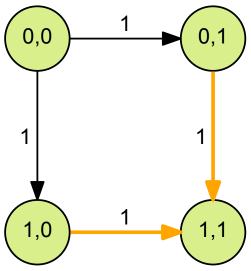
Failure of a significant link is less disruptive than a critical link because, although some packets may be lost, there are alternative paths for the packets from the source to the target. A metric is required to give a value to represent the impact of the failure of a significant link.



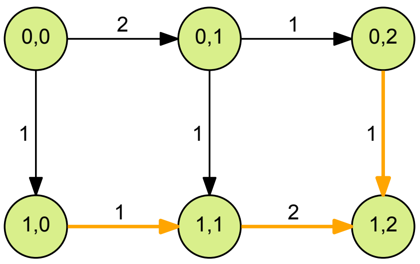
(a) Critical Link of length 1



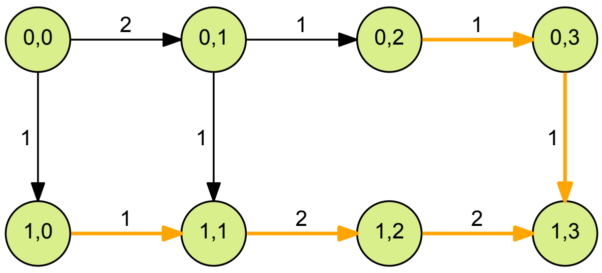
(b) Critical Link of length 2



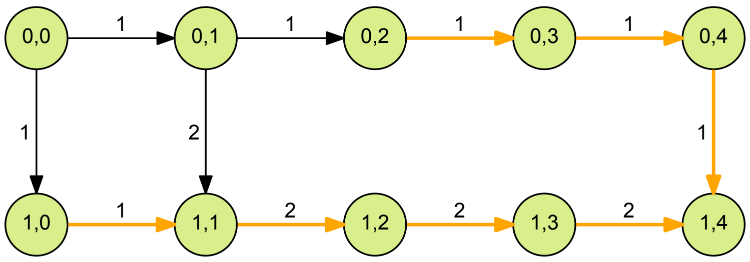
(c) Network with 2 Significant Links



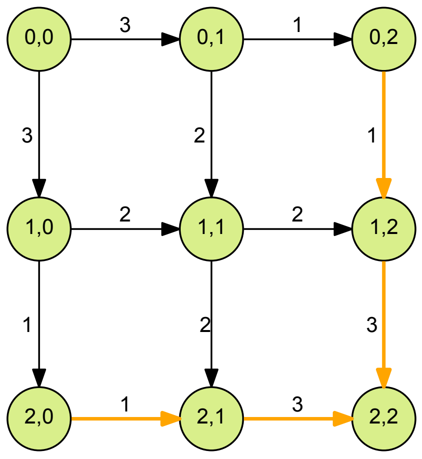
(d) Network with 3 Significant Links



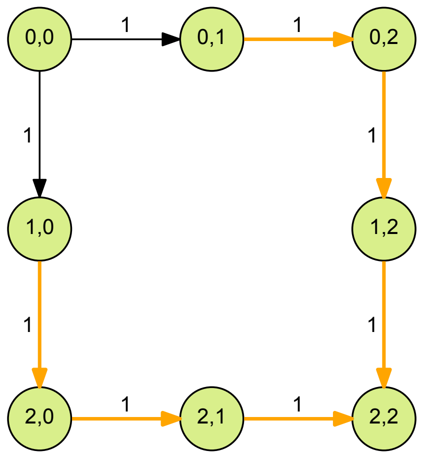
(e) Network with a single Link fault and 5 Significant Links



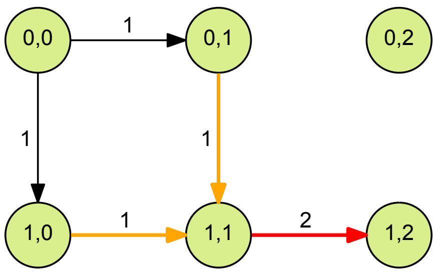
(f) Network with a 2 Link faults and 7 Significant Links



(g) Network with 4 Significant Links



(h) Network with a 4 Link faults and 6 Significant Links



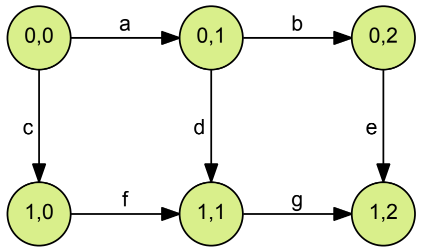
(i) Network with 2 Link faults, 2 Significant Links and 1 Critical Link

Figure F.2 – Networks illustrating Critical and Significant links

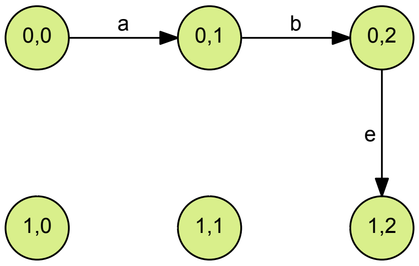
Failure of a significant link can affect multiple paths of a ComPair as illustrated in Figure F.3 where significant link g is a component of paths 2 and 3, while significant links e and f are both on only one path. The impact of link g failing is more severe than if link e or f fails. To account for this, it is important to know the number of paths each significant link affects. The links of the graphs in Figure F.2 are labelled with the number of paths that use the link. If the number of paths that use each significant link are added together for a ComPair then this gives a metric for the *vulnerability to significant link failure* for the ComPair.

$$Mvslf_c = \sum_{l=1}^{Q_l} \begin{cases} l_{n_p}, & \text{if link } l_n \text{ is a significant link} \\ 0, & \text{otherwise} \end{cases} \tag{F.4}$$

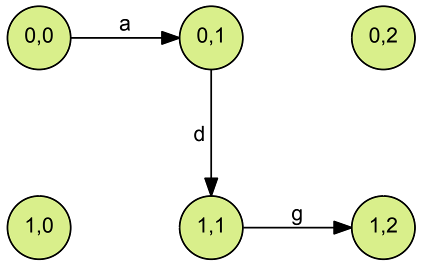
Where:



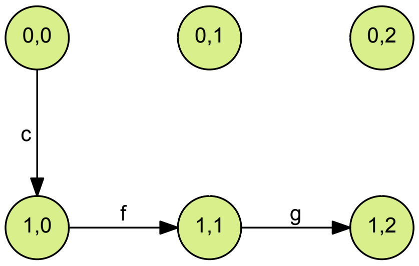
(a) 2×3 **Array**



(b) Path 1



(c) Path 2



(d) Path 3

Figure F.3 – **Paths Through a** 2×3 **Array**

$Mvslf_c$	=	The vulnerability to significant link failure for ComPair Q.
Q_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
Q_p	=	The total number of paths of the ComPair subnet.

F.3.4 Vulnerability to Link Failure Metric

The remaining task is to integrate the two metrics of *Vulnerability to Critical Link Failure* and *Vulnerability of Significant Link Failure* into a single metric that gives a single measure of the *Vulnerability to Link Failure*. Normal links are not included in the metric because they do not cause loss data. The definition of both of the above metrics is identical in that they both measure the number of paths that are affected by the failure of the link. As an initial attempt at creating a single metric, the individual metrics are summed together to produce a single value representing the total number of paths that are affect by a critical or significant link. Figure F.3 list values of the sum of the two metrics in the column $\sum$ under "Vulnerability Metric", for the subnets in Figure F.2.

Table F.1 – Comparison of Vulnerability Metrics

Graph	Graph Dimension		Paths		Vulnerability Metric		
Ref	Height	Width	Total	Affected	$\sum$	$100 \times \frac{\sum}{C_p}$	$100 \times \frac{\sum}{C_p^2}$
F.2a	1	2	1	1	1	100	100
F.2b	1	3	1	1	2	200	200
F.2c	2	2	2	2	2	100	50
F.2d	2	3	3	4	4	133	44
F.2e	2	4	3	7	7	233	77
F.2f	2	5	3	10	10	333	111
F.2g	3	3	6	8	8	133	22
F.2h	3	3	2	6	6	300	150
F.2i	1	2	2	2	4	200	100

Relating the metric values to the graphs, it is evident that this simple sum does not give a consistent value for the purposes of comparing the vulnerability of different graphs. Graph Figure F.2a has the best fitness of 1, while graphs Figure F.2b and Figure F.2c have the same fitness of 2. However graph F.2c is clearly less vulnerable than either F.2a or F.2b. The reason is that graph F.2c is a larger graph and has more paths, so there are more paths that can be affected and the metric is greater. This implies that the metric should take into account the number of paths in the graph between the source and target nodes.

Two additional metrics have been calculated, which are the sum divided by the number of paths in the ComPair and the sum divided by the square of the number of paths in the ComPair, in an attempt to correct the deficiencies of the simple sum metric. Dividing by the number of paths improves the metric but still gives anomalous values, for example the graph of Figure F.2b and F.2c have the same metric value even though the graph F.2c has two paths so is clearly less vulnerable than Figure F.2b. Dividing by the square of the number of paths gives comparative values that work well, for example the metric value for graph F.2h is three times greater than the metric value for F.2c because, although they both have two paths, F.2h has three times more significant links than F.2c, so is three times more vulnerable. The additional metrics are shown in columns $\frac{\sum}{p_l}$ and $\frac{\sum}{p_l^2}$ with the values having been multiplied by 100 to give useful integer values.

The final metric calculation for *Vulnerability to Link Failure* is given by Equation F.5.

$$Mvlf_c = \frac{100}{{C_p}^2} \sum_{l=0}^{C_l} \begin{cases} l_{n_p}, & \text{if link } l \text{ is a critical or significant link} \\ 0, & \text{otherwise} \end{cases} \tag{F.5}$$

Where:

$Mvlf_c$	=	The vulnerability to significant link failure for ComPair C.
C_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
C_p	=	The total number of paths of the ComPair subnet.

F.3.5 Link Fault Objective

This section describes an objective that combines the metrics for all ComPairs into a single value representative of the vulnerability of the whole mapping to link failure.

The *link fault objective* is to minimize the sum of the metric values for all ComPairs in the application process graph. The link fault metric gives a value for a single ComPair so the value for the objective is the sum of the fault metric values for all ComPairs in the application process graph:

$$Jlink = \sum_{l=1}^{C_l} Mvlf_c \tag{F.6}$$

Where:

$Mvlf_c$	=	The vulnerability to link failure for ComPair C.
C_l	=	The number of links in the ComPair subnet.
l_{n_p}	=	The number of paths that use link l_n.
C_p	=	The total number of paths of the ComPair subnet.

F.4 Network Power Metric and Objective

Communication traffic is a major contributor to power consumption in a many-core array [183, 184]. The total traffic flows in the many-core array can be used as an approximation for power consumed by communication traffic. This section will explore the measurement of the traffic in a many-core array and develop an objective designed to direct a search algorithm to find solutions that minimize the traffic.

F.4.1 Problem Description

The power consumed by traffic between a ComPair, for a given router/NoC architecture, is a function of both the traffic volume and the distance in terms of routing nodes and links that the traffic has to traverse between the source and target nodes. Reducing the total network traffic of all ComPairs will reduce the power consumption of the many-core array. The volume of traffic between a ComPair is predetermined by the application and cannot be influenced by the many-core system whereas the distance the traffic has to travel is dependent on the relative position of the source process node and the target process node, which is under the control of the many-core system.

F.4.2 Distance Between Communicating Core Pairs

To relate traffic to power it is necessary to know both the amount of traffic and the distance it has to travel, as the product of these gives us the total amount of traffic that is processed by the routing nodes and transmitted down the links between the source and target nodes.

The distance between a ComPair is the rectilinear distance between the source and target nodes and, assuming minimal length paths, represents the number of links the data will travel through from the source node to the target node. Faults on paths between the source and target have no effect on the length of the path that the data will travel along, providing the faults do not sever all paths between the source and target. For a ComPair Q_n corresponding the edge e_n of the application process graph with source location $Q_n(s_r, s_c)$ and target location of $Q_n(t_r, t_c)$ the *Distance Between a ComPair* metric $Mdccp_{q_n}$ (i.e the distance between the source node and the target node), is given by Equation F.7.

$$Mdccp_{q_n} = |(s_r - t_r)| + |(s_c - t_c)| \tag{F.7}$$

If the assumption is made that traffic volume between all ComPairs is the same then the distance between cores can itself be used as an approximation for the traffic and therefore the power consumption. This is not a particularly good assumption as the traffic between different ComPairs can be very different, so a more precise calculation is desired.

F.4.3 Traffic Volume Metric

An improvement to the distance between ComPairs as a metric is to use a measure that takes into account both the distance and the traffic volume for each ComPair, which will typically be different for each pair.

For a ComPair Q_n corresponding the edge e_n of the application process graph the traffic volume Q_{n_d} is the attribute traffic volume, d, from edge e_n of the application process graph. A metric that measures the *Traffic Volume Between a ComPair*, $Mtccp_{q_n}$, is given by the Equation F.8.

$$Mtccp_{q_n} = Mdccp_{q_n} \times Q_{n_d} \tag{F.8}$$

F.4.4 Power Objective

The corresponding power objective is to minimize the total power consumption across the whole network.

For the objective, a power consumption of zero is assigned when the source and target of a ComPair are adjacent, so that if the source and target nodes of every ComPair are adjacent then the value for the mapping will be zero. However, when a ComPair's source and target nodes are adjacent, the traffic volume metric $Mtccp_{q_n}$ will have a value of Q_{n_d}. Therefore when the traffic metric is used in the objective it must be adjusted down by the value of Q_{n_d}. The power objective is given in Equation F.9.

$$Jpower = \sum_{e=1}^{E_g} Mtccp_{q_n} - Q_{n_d} \tag{F.9}$$

F.5 Excess Traffic Metric and Objective

This section examines traffic flow through the many-core array that can lead to *excess traffic*. Each link in a many-core array may carry traffic from multiple ComPairs, the total of which could exceed the bandwidth of the link. The excess traffic is the traffic above the bandwidth of a link that the link is required to carry. This can be described as a link-centric with, since the excess traffic is related to each individual link, although the details of the traffic from all the ComPairs that have paths that use the link.

A link that is required to carry excess traffic is referred to as an *overloaded* link. Excess traffic for a link will eventually fill the buffers of the routing node, creating a bottleneck so potentially causing disruption throughout the network as traffic above the link's capacity is rerouted down alternative paths which may then overload other links causing more rerouting which can result in a cascade of overloaded links. If there are alternative paths to a path that uses an overloaded link, then the effect of the bottleneck may be mitigated by using an adaptive routing algorithm.

A routing node can monitor the status of the links it is using to transmit data. If a bottleneck develops on a link, the routing node is able to detect this and then employ an adaptive routing algorithm to redirect traffic down alternative paths, if they exist, using another link. In doing so, the alternative paths must still be minimal-length paths. If no alternative paths exists for the traffic, because the traffic is critical traffic, adaptive routine cannot alleviate the bottleneck. The excess traffic metric is only an approximation of the expected traffic volumes based on the traffic volume data containing the the application process graph. Only a cycle accurate simulation, with knowledge of how each process generates traffic and an understanding of the intended routing algorithm, will be able to accurately predict traffic volumes. As discussed in Chapter 7 *Graceful Degradation and Amelioration*, one of the roles of the Monitor will be to update the traffic volume attributes of the edges in the application process graph with actual observed traffic volume data to increase the accuracy of calculations of metrics and objectives.

Critical traffic cannot be rerouted; significant traffic can be rerouted by the first routing node, back up the path, that has a choice of paths for transmission of the data; normal traffic can be rerouted by the routing node link. The severity of the effect of a bottleneck will therefore depend on whether the link is carrying critical, significant or normal traffic.

Each ComPair in a mapping will have at least one, and often many, paths between the source and the target through which traffic will travel. Each path will consist of at least one link, and typically a chain of links. Each link in the array will typically be part of many paths between many ComPairs and the role of the link in each path may be either critical, significant or normal. As a result the *traffic profile* of each link is complex, consisting of the aggregation of a variety of types of traffic from many paths between many ComPairs.

In this section, a metric will be developed to represent the excess traffic of a link and evolutionary algorithm objectives designed to minimize the potential disruption of excess traffic.

F.5.1 Problem Description

The impact an overloaded link has on the network as a whole will depend of the volume of each traffic type that uses the link. Critical traffic has no other option than to use the link, so a link that is overloaded with critical traffic will cause an unavoidable bottleneck. Significant traffic, does have other available routes, but these can only be used when the congestion of traffic has propagated back up the path until it reaches a node than can direct traffic down an alternative path. Excess normal traffic can be rerouted immediately by a congested link's routing node, so will have lower impact than critical or significant traffic. This suggest the concept of *weighted excess traffic* metric which is a measure of excess traffic obtained by giving a weight to each type of excess traffic so that critical traffic makes a greater contribution to excess traffic metric than significant traffic, which in turn makes a greater contribution to the metric than normal traffic. This penalizes mappings with excess critical and excess significant traffic. For a fault free array, critical links can be avoided completely while significant links can be reduced in number but not eliminated completely since the links attached directly to the target node cannot be normal (see Section 5.1 *Communicating Core Pair (ComPair)*).

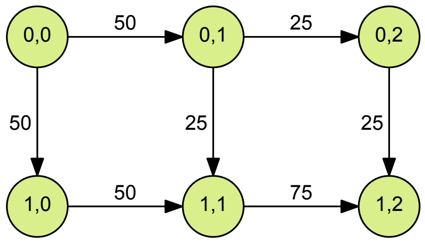
A metric that measures the excess traffic of a link can only suggest which links could be overloaded in normal operation. Calculating the expected traffic through each link, using either the non-weighted or weighted calculations, is only an approximation used to reduce computational time, based on the average predicted network load taken from the application process graph, which might prove to be very different to the actual traffic resulting from adaptive routing. In a working many-core array the on-line Monitor, discussed in Chapter 7 *Graceful Degradation and Amelioration*, will be be able to detect real bottlenecks and actual data flows between ComPairs, which can then be used in these metrics to increase accuracy.

F.5.2 Excess Traffic Metric

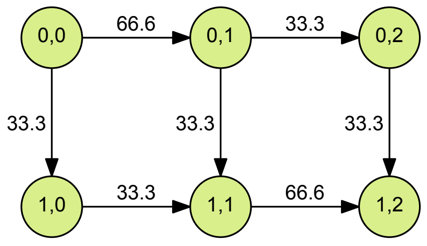
In this section the metric will be developed starting with a simple calculation using raw traffic followed by a more sophisticated approach involving link level traffic weighting.

Traffic Distribution
Traffic flowing between the source and target of a ComPair will be distributed amongst the available paths by the routing algorithm used by the routing nodes between the source and the target nodes. Two algorithms for distribution of the traffic through the ComPair subnet have been considered. The first is that the traffic is split equally between the available paths at each routing node, illustrated in Figure F.4a; the second it that the traffic is split equally between the paths from the source to the target, illustrated in Figure F.4b. Both strategies would require an adaptive routing algorithm that has the ability to understand

the strategy and keep track of the traffic that has been sent down each link so that it sends each new packet down the link that will maintain the desired balance.



(a) Distribute traffic evenly between links at each node.



(b) Distribute traffic evenly between ComPair paths at each node.

Figure F.4 – **Traffic distribution across the links used by a ComPair. The numbers on the links are percentages of the total ComPair traffic.**

When the traffic is divided equally between links at each node (Figure F.4a) the minimum and maximum traffic volumes on individual links is between are 25% and 75%, giving a range of 50% When the traffic is divided equally among the ComPair paths at each node (Figure F.4b) the minimum and maximum traffic volumes on individual links is between 33.3% and 66.6% giving a range of 33.3%. As it is preferable for the traffic to be as evenly distributed as possible the smaller range of 33.3% is more attractive than 50%. Consequently in the sections that follow the traffic will be evenly distributed between ComPair paths.

Non-Weighted Excess Traffic

The starting point is to calculate the actual traffic that is expected to flow through a link. The method is to consider each ComPair, and for each pair calculate the traffic that will travel through each link between the source and target nodes of the ComPair. Between the source and target nodes there may be many paths. In many cases paths will overlap i.e. they will use links that are used by other paths.

As an example, take a ComPair with source and target nodes that are one row and two columns apart as in Figure 6.8a. There are a total of three paths in the ComPair, illustrated in Figures 6.8b, 6.8c and 6.8d. Links b, c, e and f are used by only one path while links a, d and g are used by multiple paths. The volume of traffic between each ComPair is taken from the application process graph which provides a traffic volume for each edge in the graph. Traffic for a ComPair is distributed evenly between each of the paths between the source and target, as discussed in Subsection F.5.2. Traffic that travels along a path must travel through each link of the path. Each path is analysed in turn with traffic from each path being added to each link in the path. Each ComPair must be analysed and traffic added to each link in each path of each ComPair to produce a total traffic for each link.

Having calculated the total traffic for each link the *excess traffic* is calculated which is the traffic above the bandwidth for the link. Each overloaded link will have a value for excess traffic that is greater than zero while links that are not overloaded will have an excess traffic value of zero.

The traffic volume through a link l_n of a ComPair is the network load, Q_d, of the ComPair multiplied by the proportion of paths for the ComPair that include the link and is given by Equation F.10.

$$l_{n_t} = \frac{l_{n_p}}{Q_p} \cdot Q_d \tag{F.10}$$

Where:

l_{n_t}	=	the traffic through link l_n of ComPair Q
l_{n_p}	=	the number of paths through link l_n of ComPair Q
Q_p	=	the total number of paths in ComPair Q
Q_d	=	the network load of ComPair Q

Given that there is a mapping from each link in a ComPair path to a link in the hardware map represented by the function $f : L_{(q_e)} \to L_{(h)}$ each link l_m of the hardware map has an associated set of traffic volumes from the ComPair paths that use the hardware link l_m. If each of the paths from all ComPairs of the mapping is assigned an integer identifier from 1 to Q_{tp}, where Q_{tp} is the total number of paths from all ComPairs of the mapping, then the traffic volume $t_{(l_m,q)}$ is the traffic contributed from path q to hardware link l_m. The traffic volumes for each ComPair link calculated using Equation F.10 can be mapped to a traffic volume $t_{(l_m,q)}$ which allows the definition of traffic volume metrics in terms of $t_{(l_m,q)}$.

The metric of total traffic for link l_m is the sum of all traffic volumes from the ComPair paths that use the link and is given by Equation F.11.

$$Mtt_{(l_m)} = \sum_{q=1}^{Q_{tp}} t_{(l_m,q)} \tag{F.11}$$

The excess traffic for link l_m is given by Equation F.12

$$Mxt_{(l_m)} = \begin{cases} 0, & \text{if } Mtt_{(l_m)} \leq l_{m_b} \\ Mtt_{(l_m)} - l_{m_b}, & \text{otherwise} \end{cases} \tag{F.12}$$

Where:

$Mtt_{(l_m)}$	=	The total traffic through link l_m.
$Mxt_{(l_m)}$	=	The excess traffic through link l_m.
l_m	=	The m_{th} link of the hardware map.
l_{m_b}	=	The bandwidth of link l_m.
Q	=	The number of paths of all ComPairs for the mapping.
$t_{(l_m,q)}$	=	The traffic through link l_m contributed by path q.

Weighted Excess Traffic

A more sophisticated approach is to apply weighting to traffic through a link, based on the criticality of the link within each ComPair, producing an artificially high traffic value when the link is critical or significant. The rationale for this approach is that if a link is a critical link, then there are no alternative paths for the data, so the effect of the link being overloaded will be more severe than if the link is not a critical link. Similarly, if a link is a significant link, then there is only a single path between the significant link and the target node of a ComPair even though there is at least one other alternative path from the source node to the target node of the ComPair. The effect of a significant link being overloaded will be less severe than for a critical link but more severe than for a normal link. This section will propose a method of modifying the value of the traffic metric by applying weighting to traffic passing though different types of link.

For each link in a path the link may be critical, significant or normal. A link may be a critical link in one path, while being a significant link in another path and a normal link in yet another path.

The approach taken is to calculate the total critical, significant and normal traffic through a link from all the paths that use the link, then apply weighting in the following manner:

1) If the critical traffic is above the link's bandwidth then the weighted traffic is calculated by taking the excess critical traffic and applying the *critical traffic weight*, adding the significant traffic weighted by the *significant traffic weight* and adding the normal traffic.

2) If the critical traffic is below the link's bandwidth then the sum of critical traffic and significant traffic is compared to the link's bandwidth, and if greater than the bandwidth then the weighted traffic is the excess of the sum of critical traffic and significant traffic weighted using the significant traffic weight added to the normal traffic.

3) If sum of critical traffic and significant traffic is below the the link's bandwidth, then the sum of the critical traffic, significant traffic and normal traffic is compared to the link's bandwidth, and if greater than the bandwidth then the weighted traffic is the excess of the sum of critical traffic the significant and the normal traffic.

4) If the sum of the critical traffic, significant traffic and normal traffic is below the link's bandwidth then the weighted traffic is zero.

The weighting applied to critical traffic is greater than the weighting applied to significant traffic, while the normal traffic is not weighted. The initial values for weights are set at 5 for critical traffic and 2 for significant traffic. These values are arbitrary and will be explored by experiments.

Table F.2 shows the *non-weighted excess traffic (NWET)* and the *weighted excess traffic (WET)* for a selection of bandwidths and fixed traffic on a link, illustrating that the weighted traffic calculation produces larger traffic and excess traffic values than the non-weighted calculation for the same traffic profile. For bandwidths above 500 the actual traffic is less than the bandwidth, but the presence of critical and significant traffic exaggerates the calculated traffic which would cause the search algorithm to discriminate against such a mapping in favour of one with less critical and significant traffic.

Table F.2 – **Comparison of Non-Weighted and Weighted Excess Traffic**

Traffic Criticality	Excess Weight	Link Traffic	Weighted Traffic	Bandwidth 500	1000	1500	2000
				Weighted Excess Traffic			
Critical	5	200	1000	500	0	0	0
Significant	2	200	400	400	400	0	0
Normal	1	600	600	600	600	500	0
WET Total			2000	1500	1000	500	0
NWET Total		1000		500	0	0	0

Table F.3 shows a variety of traffic profiles for a fixed bandwidth illustrating how the weighted traffic calculation affects the excess traffic when compared to the non-weighted traffic. The four different traffic profiles are indistinguishable when weighting is not used. When weighting is used there is a clear difference in the calculated traffic values between the profiles, which will direct the search algorithm to select solutions with lower critical and significant traffic.

Table F.3 – **Comparison of Non-Weighted and Weighted Excess Traffic**

Traffic	Excess	Bandwidth : 5000							
Criticality	Weight	Tr	WET	Tr	WET	Tr	WET	Tr	WET
Critical	5	1500	2500	1000	0	500	0	500	0
Significant	2	1000	2000	1500	3000	1000	0	500	0
Normal	1	500	500	500	500	1500	1000	2000	500
Weighted Total		10000	5000	8500	3500	6000	1000	5500	500
Non-Weighted Total		3000		3000		3000		3000	

The excess traffic metric is calculated for each link in the network using the following set of equations. First the critical, significant and normal traffic components through each link from each path are defined given that $T_{(l,q)}$ is the traffic through link l contributed by path q.

$$Mct_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is critical for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{F.13}$$

$$Mst_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is significant for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{F.14}$$

$$Mnt_{(l,q)} = \begin{cases} T_{(l,q)}, & \text{if link } l \text{ is normal for the path } q \\ 0, & \text{otherwise} \end{cases} \tag{F.15}$$

Now the components for each traffic type for each link are summed to give a total of each traffic type for each link.

$$Mct_l = \sum_{q=1}^{Q} Mct_{(l,q)} \tag{F.16}$$

$$Mst_l = \sum_{q=1}^{Q} Mst_{(l,q)} \tag{F.17}$$

$$Mnt_l = \sum_{q=1}^{Q} Mnt_{(l,q)} \tag{F.18}$$

Now the excess traffic for each link can be defined in terms of each of the individual traffic types and the traffic type weights.

$$Mxwt_l = \begin{cases} (Mct_l - Bw_l) \times Wc + & \\ \quad Ts_l \times Ws + Mnt_l, & \text{if } Mct_l > Bw_l \\ (Mct_l + Ts_l - Bw_l) \times Ws + & \\ \quad Mnt_l, & \text{if } (Mct_l + Ts_l) > Bw_l \\ Mct_l + Ts_l + Mnt_l - Bw_l, & \text{if } (Mct_l + Mst_l + Mnt_l) > Bw_l \\ 0, & \text{otherwise} \end{cases} \tag{F.19}$$

Where:

$Mxwt_l$	=	The excess weighted traffic through link l.
Mct_l	=	The critical traffic through link l.
$Mct_{(l,q)}$	=	The critical traffic through link l from path q.
Mst_l	=	The significant traffic through link l.
$Mst_{(l,q)}$	=	The significant traffic through link l_n from path q.
Mnt_l	=	The normal traffic through link l.
$Mnt_{(l,q)}$	=	The normal traffic through link l from path q.
Wc	=	The critical traffic weight.
Ws	=	The significant traffic weight.
$T_{(l,q)}$	=	The traffic through link l contributed by path q.

F.5.3 Excess Traffic Objectives

The previous section described methods for calculating the traffic metric for a link. The next step is to determine how the link excess traffic metric can be used to produce objective measures for an evolutionary algorithm. The link traffic metric can be analysed in a number of ways, each of which has different merits.

Link Traffic Landscape

To evaluate the alternatives it is necessary to have an understanding of the *link traffic landscape* that is desirable. If we view the links as points on a 2-dimensional plane surface where the traffic metric can be viewed as the height of each point above plane. When all links have a zero excess traffic, which is the ideal landscape, the surface will be perfectly flat.

A small number of links with significantly higher values than the other links would be represented by a surface with a small number of high peaks. Links with high values would represent bottlenecks which could seriously degrade the whole network, so are undesirable. Another possibility is that the excess traffic is evenly distributed across the network, which would be represented by a raised but relatively level surface.

The landscape has two properties of interest: the first is the total of the excess traffic, which is the total height of each point above the plane; the second is how smooth or rugged the landscape is, i.e. how varied is the height of the points. These properties are independent of each other so require two different objectives for the evolutionary algorithm to minimize: *Total Excess Traffic* and *Excess Traffic Variance*.

Sum of Excess Traffic

The excess traffic of all links in the network are summed giving a measure of the total excess traffic for the mapping. The sum of excess traffic does not distinguish between two mappings that have a similar sum but significantly different landscapes, one with a small number of very high values alongside many small values, the other where there is a larger number of similar values without a single very large value. The sum of excess traffic of all links is a candidate for the total excess traffic metric. The sum of excess traffic of all links in the network, Jxt_{sum} is given by Equation F.20.

$$Jxt_{sum} = \sum_{l=1}^{L} Mxwt_l \tag{F.20}$$

Mean Average of Excess Traffic

The excess traffic values of all links in the network are summed and then divided by the number of links in the network. This measure gives the same information as the simple sum since the number of links in a network is constant. The mean average of excess traffic of all links in the network is another candidate for the total excess traffic metric. The mean average of excess traffic of all links in the network, Jxt_{mean} is given as:

$$Jxt_{mean} = \overline{Mxwt} = \frac{1}{L}\sum_{l=1}^{L} Mxwt_l \tag{F.21}$$

This measure of traffic suffers from the same inability as the simple sum measure to distinguish between mappings with similar traffic values but significantly different landscapes.

Maximum Value of Excess Traffic

This objective considers only the highest value of excess traffic of a single link from all the links in the network. Minimising the objective reduces the maximum single value of excess traffic in the network but does not consider the overall level of excess traffic, which is the opposite of using a simple sum measure of excess traffic. This is a candidate for a simple excess traffic variance objective, since a lower maximum single value will also reduce the variance. The maximum value of excess traffic of all links in the network, Jxt_{max} is given by Equation F.22.

$$Jxt_{max} = \max\{Mxwt_l : l = 1, \ldots, L\} \tag{F.22}$$

Standard Deviation (SD) of Excess Traffic

The standard deviation gives a measure of how much variation from the mean there is in a population. A value of zero means that all values are equal, so in the case of excess traffic a zero value translates to all links having equal levels of excess traffic, however the SD does not give any information about the absolute level of excess traffic across the whole network. The standard deviation of excess traffic of all links in the network is a candidate for the excess traffic variance objective. The standard deviation of excess traffic of all links in the network, Jxt_{sd} is given as:

$$Jxt_{sd} = \sqrt{\frac{1}{L}\sum_{l=1}^{L}(Mxwt_l - \overline{Mxwt})^2} \tag{F.23}$$

On its own, using standard deviation will not help to reduce the overall level of excess traffic as a zero SD can be obtained for any level of excess traffic.

Absolute Mean Deviation (AMD) of Excess Traffic

Absolute Mean Deviation is an alternative to standard deviation that is simpler to calculate because it uses the absolute difference in place of the square of the difference. For a calculation that will be repeated many millions of times the difference in computation time between finding the absolute difference and computing the square of a difference and then later obtaining a square root, is significant. The absolute mean deviation of excess traffic of all links in the network is a candidate for the excess traffic variance objective. The absolute mean deviation of excess traffic of all links in the network, Jxt_{amd} is given as:

$$Jxt_{amd} = \frac{1}{L}\sum_{l=1}^{L}|Mxwt_l - \overline{Mxwt}| \tag{F.24}$$

It can be argued that the AMD is as good as and in some cases a better measure than SD [185]. The same comments made for the suitability of SD can also be made for AMD, however due to the simpler computation AMD would be preferred over SD.

Glossary

Absolute mean deviation (AMD)
A statistical measure similar that uses the differences between means.

Adjacent Cores
Two cores are adjacent if they are separated by a single step in one direction.

Aggregate Map
A combination of the environment map and the process map use to calculate metrics that rake into account data sources and sinks external to the many-core array.

Amdahl's Law
Was stated by Getov [66] as: "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

Application
Software designed to complete a well define computational task which may be subdivided into processes.

Application Process Graph (APG)
A graph representing the processes and process to process relationships of an application.

Cloned
EA: a copy of an individual without any changes to the genome.

ComPair
A pair of communicating cores in a many-core array. Used in the calculation of metrics.

Computational Budget
A set amount of processor resource allocated for an evolutionary algorithm expressed in term of the number of individuals evaluated.

Core
The processing element of a many-core array consisting of a CPU, accelerator and local resources, including memory.

Core Fault Tolerance
On objective that is used to measure how well a mapping protects a many-core system from core faults.

Correlation (1)
EA: Describes the how closely changes to the genome are reflected in the phenome.

Correlation (2)
EA: Describes the how well aligned are the fitness values of multiple objectives.

Critical Links
A link whose failure will sever communication between a ComPair.

Critical Traffic
Traffic that traverses a critical link.

Critical Traffic Weight
A weighting given to critical traffic the calculation the excess traffic metric.

Crossover
A genetic operator that takes two parents and generates a descent using genetic information form both parents.

Densely Connected
An APG that has 2.5 or more edges per node.

Descendants
EA: Individuals derived from mutations applied to one parent or crossover of two parents.

Directed Acyclic Graph(DAG)
A graph with directed edges such that information only flows from the source node of the edge to the target node of the edge.

Double Modular Redundancy (DMR)
A system design where hardware modules are duplicated, one used for processing and the second one in standby mode.

Edge
A connection in the application process graph between two processes that represents data transfer from one process to the other.

Edge
An edge connects two nodes and is identified by the locations of the nodes that it connects. An edge has a source node, the node closest the origin of a lattice or source node of the subnet, and a target node, the node furthest from the origin of a lattice or closest to the target node of the subnet. An edge e is, therefore, identified by an ordered 2-tuple of node locations, (sloc, tloc) = ((sr, sc), (tr, tc)).

Edge-Coincident Paths
A set of edges that have at least one path that traverse all edges in the set.

Elite Individuals
EA: The individuals in a generation that are copied unchanged to the next generation.

Engineered Mappings
Mappings that are generated from deterministic algorithms.

Environment
A mapping that contains the source and sinks to resource external to the many-core array.

Evolutionary algorithm (EA)
An algorithm that mimics biological evolutionary processes to search for a solution space.

Evolutionary Cycle
EA: the series of populations and processes that transformations one generation to the next.

Excess Traffic
Traffic exceeds the bandwidth of a hardware link.

Failed Core
A core that has a fault which will make it permanently unavailable for use.

Fault-Free
A fault-free lattice is a lattice where all nodes and edges are present and can be used by paths.

Fault-tolerance
is the architectural attribute of a digital system that keeps the logic machine doing its specified tasks when its host, the physical system, suffers various kinds of failures of its components.

Fault-Recovery Cycle
The control software used to implement fault detection, graceful degradation and graceful amelioration.

Faulty Edge
A faulty edge is an edge that cannot be used by a path, in effect removing the edge from the lattice.

Faulty Node
A faulty node is a node that cannot be visited by a path, in effect removing, from the lattice, the node and all edges leading to and from the node.

Fitness Evaluated Population
The population in the evolutionary cycle created by calculating the objective values of each individual.

Flynn's Taxonomy
In 1972 Flynn defined four processing models based on the combination of types of instruction stream and types of data stream the the processor exploits resulting in the following four categories as described in [61]:

Generation Zero
The first generation of an evolution that is created from engineered and random mappings.

Genetic Operator
A method of changing the genome of an individual.

Genome
A representation of the process map that is used to manipulate the process map.

Globally Asynchronous Local Synchronous (GALS)
The concept of a large device using individual clocks for localise processing units, to overcome clock propagation delays in large devices.

Graceful Amelioration
The ability of a system to improve performance

Graceful Degradation
The ability of a system to continue functioning with reduced performance when faults occur.

Hardware Map
A representation of the hardware of a many-core system.

Idle Core
A core of a many-core does not have a process allocated to it (synonym with spare)

Initial Population
The population in the evolutionary cycle created as the stating point for the evolutionary process.

Intermediate Population
The population created by selection and mutation of individuals from the primary population.

Lattice
A lattice consists of nodes arranged in equally spaced rows and columns. In general a lattice can be considered to have an infinite number of rows and columns. For practical purposes lattices can be made finite by specifying a size of R rows and C columns

Link
A hardware link between two routing nodes.

Link Criticality
A description of how the failure of a link will affect a ComPair.

Link Fault Tolerance
An objective that measure how well mapping protects a many-core system from link faults.

Location
A location is a 2-tuple specifying the row and column coordinates

Many-core
A SoC with 10s, 100s or 1000s of independent processing cores.

MIMD
Flynn's Taxonomy: Multiple Instruction streams, Multiple Data streams.

Minimum discovered fitness
The minimum fitness for an objective that was found during one or more evolutionary runs.

Minimum Length Path
A path that is the shortest possible path between a ComPair.

MISD
Flynn's Taxonomy: Multiple Instruction streams, Single Data stream.

Moderately Connected
An APG with at least 2 edges per node and less than 2.5 edges per node.

Monitor
A process that is responsible for monitoring and supervising a region of cores.

Monitor Node
The many-core node assigned to execute the monitor process.

Multi-core
A processor technology with multiple (between 2 an 64) sophisticated cores.

Mutation
EA: The action of making changes to the genetic representation of an individual.

Mutation Rate
EA: the number of genes changed during mutation of a genome.

Nearest Spare Core Search
A search algorithm to find the nearest spare core to a processing core.

Network hop
A network hop is the minimum possible distance between two many-core nodes.

Network Power
A measure of the power used by communication of data through a NoC.

Network on Chip (NoC)
A system where processing elements are interconnected via routing nodes creating an on chip communications network.

Node
A node is uniquely identified by it's location loc(r, c); being the row and column coordinates of the node within the lattice. Each node can be connected, via edges, to each of its north, south, east and west neighbours (where they exist) as in figure 6.10.

Non-Weighted Excess Traffic (NWET)
A measure of network traffic with no weighting applied.

Normal links
Links whose failure causes no immediate disruption to traffic.

Normal Traffic
Traffic that traverses a normal link.

NSGAII
A non-dominated Pareto front sorting genetic algorithm.

Ordered Set of Edges
A set whose members are order by defined criteria.

Origin
The location (0, 0) of a lattice

Pareto Front
A set of points in a multi-objective space that are considered equivalent because no point can be considered to better, for all objectives, than any of the other points in the set.

Path-Coincident Edges
A set of edges for which there is at least one path that passes through all of the edges in the set.

Permutation
EA: a genetic operator that takes a genome and produces a new genome by swapping one or pairs of genes.

Pf0
Pareto Front 0: A cumulative Pareto Front that includes all non-dominated solution from the generations of an evolution.

Pf1
The non-dominated solutions from a single generation.

Phenome
A direct representation of a process map.

Primary Population
The population that is the produced at the end of each evolutionary cycle.

Process
The smallest part of an application that can be executed independently on a dedicated core.

Process Map
A mapping of APG processes to cores in a many-core array (synonym for task map).

Process Node
A node in an APG that represents a process of the application.

Processing Node
A hardware unit of a many-core array consisting of a network router and processing core.

Protected Core
A core with at least one adjacent idle core.

Rectilinear distance
The distance between two nodes in a square lattice. The sum of the absolute difference of the x-coordinates and the absolute difference of the y-coordinates, of the two nodes.

Region
A group of cores in a many-core array that are collectively monitored by a single monitor.

Routing Node
The node in a NoC that is responsible for receiving and transmitting data packets through the network.

Significant links
A link whose failure will cause disruption and possible loss of traffic between a ComPair but will not completely sever communication between the ComPair.

Significant Traffic
Traffic that traverses a significant link.

Significant Traffic Weight
A weight applied to significant traffic when calculating the excess traffic metric.

SIMD
Flynn's taxonomy: Single Instruction stream, Multiple Data streams.

Sink Node
A node which represents a receiver of data external to the many-core array.

SISD
Flynn's Taxonomy: Single Instruction stream, Single Data stream.

Sorted Population
EA: The population created by performing a Pareto sort on an evaluated population

Source Node
A node which represents a source of data external to the many-core array.

Spare Core
A core of a many-core does not have a process allocated to it (synonym with idle).

Sparsely Connected
An APG with less than 2 edges per node.

Step
A single network hop.

Subnet
A subnet is any portion of a lattice defined by a source location and a target location.

Synchronised Redundant Systems
Two or more systems that carry out the same processing and periodically synchronize to ensure they their computations agree

System on Chip
A system which is composed of a collection of IPs on a single device

Task Migration
Relocation of a task from one core to another.

Task Map
A mapping of application process graph processes to cores in a many-core array (synonym for process map).

Triple Modular Redundancy (TMR)
System where three identical modules carry out the same processing and the results of all three are used as the input of a voting system to determine the final output.

Vulnerability of Significant Link Failure
A metric that measures how vulnerable a ComPair is to the failure of a significant link.

Vulnerability to Critical Link Failure
A metric that measures how vulnerable a ComPair is to the failure of a critical link.

Vulnerability to Link Failure

A metric that measures how vulnerable a ComPair is to the failure of any link.

Weighted Excess Traffic (WET)

A measure of network traffic with weighting applied to critical and significant traffic.

Bibliography

[1] R. N. Noyce and M. E. Hoff. “A History of Microprocessor Development at Intel”. In: *Micro, IEEE* 1.1 (1981), pp. 8–21.

[2] G. E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (1965), pp. 114–117.

[3] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. “Network on chip: An architecture for billion transistor era”. In: *Proceeding of the IEEE NorChip Conference*. 2000, pp. 166–173.

[4] SIA-Semiconductor-Industry-Association. “The National Technology Roadmap for Semiconductors”. In: *Semiconductor Industry Association* (1997).

[5] S. Borkar. “Thousand Core Chips - A Technology Perspective”. In: *Proceedings of the 44th annual Design Automation Conference* (2007), pp. 746–749.

[6] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. “Toward Dark Silicon in Servers”. In: *IEEE Micro* 31.4 (2011), pp. 6–15.

[7] C. Martin. “Multicore Processors: Challenges, Opportunities, Emerging Trends”. In: *Embedded World Conference 2014* (2014).

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. “Dark Silicon and the End of Multicore Scaling”. In: *Proceeding of the 38th annual international symposium on Computer architecture - ISCA ’11* (2011), pp. 365–376.

[9] J. Hu and R. Marculescu. “Energy-Aware Mapping for Tile-based NoC Architectures Under Performance Constraints”. In: *Design Automation Conference, ASP-DAC 2003. Asia and South Pacific* (2003), pp. 233–239.

[10] T. Lei and S. Kumar. “A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture”. In: *Euromicro Symposium on Digital System Design (DSD’03).* 2003.

[11] S. Murali and G. De Micheli. “Bandwidth-Constrained Mapping of Cores onto NoC Architectures”. In: *Design, Automation and Test in Europe Conference and Exhibition, 2004.* (2004), pp. 1–5.

[12] G. Ascia, V. Catania, and M. Palesi. “Multi-objective Mapping for Mesh-based NoC Architectures”. In: *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004.* (2004), pp. 182–187.

[13] M. N. S. M. Sayuti and L. S. Indrusiak. “Simultaneous Optimisation of Task Mapping and Priority Assignment for Real-Time Embedded NoCs”. In: *Proceedings - 23rd*

Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015 (2015), pp. 692–695.

[14] O. Derin, D. Kabakci, and L. Fiorin. “Online Task Remapping Strategies for Fault-tolerant Network-on-Chip Multiprocessors”. In: *NOCS* (2011).

[15] M. N. S. M. Sayuti and L. S. Indrusiak. “A Constructive Task Mapping Algorithm for Hard Real-Time Embedded NoCs”. In: *2015 IEEE Conference on Systems, Process and Control (ICSPC)* December (2015), pp. 123–128.

[16] A. Das and A. Kumar. “Fault-aware task Re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs”. In: *Proceedings - IEEE International Symposium on Rapid System Prototyping, RSP* (2012), pp. 149–155.

[17] F. Khalili and H. R. Zarandi. “A fault-tolerant core mapping technique in networks-on-chip”. In: *IET Computers & Digital Techniques* 7.6 (2013), pp. 238–245.

[18] N. Chatterjee, S. Paul, and S. Chattopadhyay. “Fault-Tolerant Dynamic Task Mapping and Scheduling for Network-on-Chip-Based Multicore Platform”. In: *ACM Transactions on Embedded Computing Systems* 16.4 (2017), pp. 1–24.

[19] G. E. Moore. “Progress in digital integrated electronics”. In: *Electron Devices Meetings, 1975 International* 21 (1975), pp. 11–13.

[20] J. Crawford. “The Execution Pipeline of the Intel i486 CPU”. In: *Compcon Spring ’90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE*. 1990, pp. 254–258.

[21] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. “The Microarchitecture of the Pentium 4 Processor”. In: *Intel Technology Journal* Q1 (2001), pp. 1–13.

[22] C. Stephens and M. Dennis. “Engineering time: inventing the electronic wristwatch”. In: *The British Journal for the History of Science* 33.4 (2000), S0007087400004167.

[23] A. M. Volk, P. A. Stoll, and P. Metrovich. “Recollections of Early Chip Development at Intel”. In: *Intel Technology Journal* Q1 (2001), pp. 1–12.

[24] W. Wolf, A. A. Jerraya, and G. Martin. “Multiprocessor system-on-chip (MPSoC) technology”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1701–1713.

[25] P. Guerrier and A. Greiner. “A Generic Architecture for On-Chip Packet-Switched Interconnections”. In: *Design, Automation and Test in Europe Conference and Exhibition 2000.* (2000).

[26] J. Backus. “Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641.

[27] W. B. Ackerman. “Data Flow Languages”. In: *Computer* 15.2 (1982), pp. 15–25.

[28] J. B. Dennis and D. P. Misunas. “A Computer Architecture for Highly Parallel Signal Processing”. In: *December 1973 ACM ’74: Proceedings of the 1974 annual ACM conference*. ACM, 1974, pp. 402–409.

[29] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. “Advances in dataflow programming languages”. In: *ACM Computing Surveys* 36.1 (Mar. 2004), pp. 1–34.

[30] J. B. Dennis. “Data Flow Computer Architecture”. In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 508–512.

[31] A. Davis. “The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine”. In: *Proceedings of the 5th Annual Symposium on Computer Architecture* July (1978), pp. 210–215.

[32] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. “Data-Driven and Demand-Driven Computer Architecture”. In: *ACM Computing Surveys (CSUR)* 14.1 (1982), pp. 93–143.

[33] A. Kathail and V. Kathail. “A Multiple Processor Data Flow Machine that Supports Generalized Procedures”. In: *ISCA 81 Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press Los Alamitos, CA, USA, 1981, pp. 291–302.

[34] H. Veen. “Dataflow Machine Architecture”. In: *ACM Computing Surveys (CSUR)* 18.4 (1986), pp. 365–396.

[35] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transaction on Computers* C-21.9 (1972), pp. 948–960.

[36] J. Gregoryt and R. Mcreynoldst. “The SOLOMON Computer”. In: *IEEE Transactions on Electronic Computers* (1963), pp. 774–781.

[37] R. P. Mohanty, A. K. Turuk, and B. Sahoo. “Analysing the Performance of Multi-core Architecture”. In: *1st Int.Conf. On Computing Communication and Sensor Networks-CCSN’2012* (2012).

[38] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *AFIPS Conference Proceedings - Spring Point Computer Conference, 1967* 30 (1967), pp. 483–485.

[39] G. M. Amdahl. “Computer Architecture and Amdahls Law”. In: *Computer* 46.12 (2013), pp. 38–46.

[40] V. Getov. “A Few Notes on Amdahls Law”. In: *Computer* 64.12 (2013), p. 45.

[41] J. L. Gustafson. “Reevaluating Amdahls Law”. In: 31.5 (1988), pp. 532–533.

[42] J. Bui, C. Xu, and S. Gurumurthi. “Understanding Performance Issues on both Single Core and Multi-core Architecture”. In: (2007).

[43] I. Onyuksel and S. H. Hosseini. “Amdahls Law: A Generalization Under Processor Failures”. In: *IEEE Transactions on Reliability* 44.3 (1995), pp. 455–462.

[44] B. Parhami. “Amdahls Reliability Law: A Simple Quantification of the Weakest-Link Phenonmenon”. In: *Computer* 48.7 (2015), pp. 55–58.

[45] M. Annaratone. “MPPs, Amdahls Law, and Comparing Computers”. In: *Proceedings of The Fourth Symposium on the Frontiers of Massively Parallel Computation* (1992), pp. 465–470.

[46] M. D. Hill and M. R. Marty. “Amdahls Law in the Multicore Era”. In: *Computer* 41.July (2008), pp. 33–38.

[47] D. Sylvester and K. Keutzer. “Getting to the Bottom of Deep Submicron II : A Global Wiring Paradigm”. In: *Proceedings of the 1999 international symposium on Physical design*. ACM, 1999, pp. 193–200.

[48] D. Sylvester and K. Keutzer. “A Global Wiring Paradigm for Deep Submicron Design”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 19.2 (2000), pp. 242–252.

[49] D. Chapiro. “Globally-Asynchronous Locally-Synchronous Systems”. PhD thesis. Stanford University, 1984.

[50] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, P. Ellervee, and D. Lundqvist. “Lowering power consumption in clock by using Globally Asynchronous Locally Synchronous deisgn style”. In: *ACM/IEEE Design* (1999).

[51] A. Iyer and D. Marculescu. “Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors”. In: *ACM SIGARCH Computer Architecture News* 30.2 (May 2002), p. 158.

[52] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner. “Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems”. In: *Twelfth Annual IEEE International ASIC/SOC Conference (Cat. No.99TH8454)* (1999), pp. 317–321.

[53] K. Y. Yun and R. P. Donohue. “Pausible Clocking: A First Step Toward Heterogeneous Systems”. In: *iccd*. Published by the IEEE Computer Society, 1996, p. 118.

[54] D. S. Bormann and P. Y. Cheung. “Asynchronous Wrapper for Heterogeneous Systems”. In: *Proceedings International Conference on Computer Design VLSI in Computers and Processors* (1997), pp. 307–314.

[55] S. Moore, G. Taylor, R. Mullins, and P. Robinson. “Point to point GALS interconnect”. In: *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002, pp. 69–75.

[56] R. R. Dobkin, R. Ginosar, and C. P. Sotiriou. “Data Synchronization Issues in GALS SoCs”. In: *10th International Symposium on Asynchronous Circuits and Systems, 2004. Proceedings.* (2004), pp. 170–180.

[57] A. Royal and P. Y. Cheung. “Globally Asynchronous Locally Synchronous FPGA Architectures”. In: *Field-Programmable Logic and Applications*. Springer, 2003, pp. 355–364.

[58] P. Kermani and L. Kleinrock. “Virtual Cut-Through : A New Computer Communication Switching Technique”. In: *Computer Networks* 3.4 (1979), pp. 267–286.

[59] P. Baran. “On Distributed Communications Networks”. In: *Communications Systems* 12.1 (1963), pp. 1–9.

[60] W. J. Dally and C. L. Seitz. “The torus routing chip”. In: *Distributed Computing* 1.4 (1986), pp. 187–196.

[61] W. J. Dally and C. L. Seitz. “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”. In: *Computers, IEEE Transactions on* 100.5 (May 1987), pp. 547–553.

[62] W. J. Dally. “Performance Analysis of k-ary n-cube Interconnection Networks”. In: *Computers, IEEE Transactions on* 39.6 (June 1990), pp. 775–785.

[63] S. Felperin, P. Raghavan, and E. Upfal. “A Theory of Wormhole Routing in Parallel Computers”. In: *IEEE Transactions On Computers* 45.6 (1996).

[64] L. Kleinrock. *Queueing System, Volume II Computer Applications*. Vol. II. Wiley, New York, 1976, p. 1976.

[65] W. J. Dally. “A VLSI Architecture for Concurrent Data Structures”. PhD thesis. California Institute of Technology, 1986.

[66] W. J. Dally. “Virtual-Channel Flow Control”. In: *ACM SIGARCH Computer Architecture News* 18 (1990), pp. 60–68.

[67] X. Lin and L. M. Ni. “Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks”. In: *ISCA'91 18th International Symposium on Computer Architecture* (1991), pp. 116–125.

[68] C. J. Glass and L. M. Ni. “Adaptive Routing in Mesh-Connected Networks”. In: *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference* (1992), pp. 12–19.

[69] C. J. Glass and L. M. Ni. “The Turn Model for Adaptive Routing”. In: *Journal of the ACM (JACM)* (1992).

[70] C. J. Glass and L. M. Ni. “Maximally Fully Adaptive Routing in 2D Meshes”. In: *International Conference on Parallel Processing*. 1992.

[71] G.-M. Chiu. “The Odd-Even Turn Model for Adaptive Routing”. In: *and Distributed Systems, IEEE Transactions on* 11.7 (2000), pp. 729–738.

[72] E. Nilsson, M. Millberg, J. Oberg, and A. Jantsch. “Load distribution with the Proximity Congestion Awareness in a Network on Chip”. In: *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2003.

[73] M. Li, Q.-A. Zeng, and W.-B. Jone. “DyXY - A Proximity Congestion-Aware Deadlock-Free Dynamic Routing Method for Network on Chip”. In: *DAC '06 Proceedings of the 43rd annual Design Automation*. 2006, pp. 849–852.

[74] M. Ebrahimi, M. Daneshtalab, J. Plosila, and F. Mehdipour. “MD: Minimal path-based fault-tolerant routing in on-Chip Networks”. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Ieee, Jan. 2013, pp. 35–40.

[75] I. Caragiannis, C. Kaklamanis, and I. Vergados. “Greedy Dynamic Hot-Potato Routing on Arrays”. In: *Parallel Architectures, Algorithms and Networks, 2000. I-SPAN 2000.* 2000, pp. 178–185.

[76] L. Wang, X. Wang, and T. S. Mak. “Dynamic Programming-Based Lifetime Aware Adaptive Routing Algorithm for Network-on-Chip”. In: *IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*. Vol. 2015-Janua. January. 2015.

[77] L. Wang. “Lifetime Reliability and Performance Optimization in NoC-Based Many-Core Computing Systems”. In: June (2017).

[78] W. J. Dally and B. Towles. “Route Packets, Not Wires: On-Chip Interconnection Networks”. In: *Design Automation Conference, 2001.* (2001), pp. 684–689.

[79] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. “A network on chip architecture and design methodology”. In: *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002* (2002), pp. 117–124.

[80] S. B. Akers and B. Krishnamurthy. "A Group-Theoretic Model for Symmetric Interconnection Networks". In: *IEEE Transactions on Computers* 38.4 (Apr. 1989), pp. 555–566.

[81] Y. Robert. "Task graph scheduling". In: *Encyclopedia of Parallel Computing* (2011), pp. 2013–2025.

[82] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Fourth. Elsevier, 2009, p. 2009.

[83] A. W. Burks, H. H. Goldstine, and J. von Neumann. *Preliminary Discussion of The Logical Design of an Electronic Computing Instrument*. Tech. rep. Institute of Advanced Study report for the U.S. Army Ordnance Department, 1946.

[84] L. M. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems". In: *IEEE Transactions on Computers* C-27.12 (1978), pp. 1112–1118.

[85] F. Baskett, T. Jermoluk, and D. Solomon. "The 4D-MP graphics superworkstation: computing+graphics=40 MIPS+MFLOPS and 100000 lighted polygons per second". In: *Compcon Thirty-Third IEEE Computer Society International Conference*. 1988, pp. 468–471.

[86] J. Croll. "VAX 6000 model 400 system overview". In: *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE* (1990), pp. 110–114.

[87] C. Tang. "Cache system design in the tightly coupled multiprocessor system". In: *AFIPS'76 National Computer Conference* (1976), pp. 749–754.

[88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence". In: *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on* (1988), pp. 280–289.

[89] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. "Directory-Based Cache Coherence in Large-Scale Multiprocessors". In: *Computer* 23.June (1990), pp. 49–58.

[90] P. Stenstrom. "A Survey of Cache Coherence for Multiprocessors". In: *IEEE Computer* 23.6 (1990), pp. 12–24.

[91] N. Eisley, L.-S. Peh, and L. Shang. "In-Network Cache Coherence". In: *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International* (2006), pp. 321–332.

[92] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. "Cuckoo directory: A scalable directory for many-core systems". In: *Proceedings - International Symposium on High-Performance Computer Architecture* (2011), pp. 169–180.

[93] D. Hilbert and W. Ackermann. *Principles of Mathematical Logic*. Chelsea Publishing Company, New York, 1950, 1928.

[94] K. Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 173–198.

[95] M. Davis. *The Undecidable*. Raven Press, Hewlett, New York, 1965.

[96] M. Hirzel. "On Formally Undecidable Propositions of Principia Mathematica and Related Systems (Translation)". In: *Philosophical Books* 4 (2000), pp. 1–22.

[97] A. M. Turing. “ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM”. In: *Proceedings of the London Mathematical Society* 42 (1936), pp. 230–265.

[98] E. L. Post. “Finite Combinatory Processes-Formulation 1”. In: *The Journal of Symbolic Logic* 1.3 (1936), pp. 103–105.

[99] A. Church. “The Calculi of Lambda-Conversion”. In: *Annals of Mathematics Studies Number 6* (1941). arXiv: `arXiv:1011.1669v3`.

[100] J. C. Shepherdson and H. E. Sturgis. “Computability of Recursive Functions”. In: *Journal of the ACM (JACM)* 10.2 (1963), pp. 217–255.

[101] J. von Neumann. *First Draft of a Report on the EDAVAC*. Tech. rep. 1945, pp. 1–49.

[102] R. M. Keller, G. Lindstrom, and S. Patil. “A Loosely-Coupled Applicative Multi-Processing System”. In: *National Computer Conference*. 1979, pp. 613–622.

[103] I. Watson and J. Gurd. “A Practical Data Flow Computer”. In: *Computer* 15.2 (1982), pp. 51–57.

[104] R. M. Karp and R. E. Miller. “Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing”. In: *SIAM Journal on Applied Mathematics* 14.6 (1966), pp. 1390–1411.

[105] J. B. Dennis. “On the design and specification of a common base language”. In: *Proceedings of the Symposium on Computers and Automata*. Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, N. Y., 1971, pp. 47–74.

[106] J. B. Dennis, J. Fosseen, and J. Linderman. “Data Flow Schemas”. In: *International Symposium on Theoretical Programming*. Vol. 5. 1972, pp. 187–216. arXiv: `arXiv:1011.1669v3`.

[107] J. B. Dennis. “First Version of a Data Flow Procedure Language”. In: *Programming Symposium*. Springer Berlin Heidelberg, 1974, pp. 362–376.

[108] G. Kahn and D. B. MacQueen. “Coroutines and networks of parallel processes”. In: *IFIP Congress Proceedings*. 1977.

[109] A. L. Davis and R. M. Keller. “Data Flow Program Graphs”. In: *Computer* 15.2 (1982), pp. 26–41.

[110] Arvind, D. E. Culler, and G. K. Maa. “Assessing the Benefits of Fine- Grain Parallelism in Dataflow Programs”. In: *International Journal of High Performance Computing Applications* 2.3 (1988), pp. 10–36.

[111] J. B. Dennis. “Data Flow Graphs”. In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 512–518.

[112] M. E. Conway. “Design of a Separable Transition-Diagram Compiler”. In: *Communications of the ACM* 6.7 (1963), pp. 396–408.

[113] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[114] L. Lamport. “The Parallel Execution of DO oops”. In: *Communications of the ACM* 17.2 (1974), pp. 83–93.

[115] J. R. Allen and K. Kennedy. “Automatic Loop Interchange”. In: *SIGPLAN ’84 Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. Vol. 19. 6. 1984, pp. 233–246.

[116] A. Aiken and A. Nicolau. “Perfect Pipelining: A New Loop Parallelization Technique”. In: *2nd European Symposium on Programming*. Vol. 300. 1988, pp. 221–235.

[117] R. Gupta. “Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 3.4 (1992), pp. 505–512.

[118] K. Lakshmanan, S. Kato, and R. (Rajkumar. “Scheduling Parallel Real-Time Tasks on Multi-core Processors”. In: *Real-Time Systems Symposium (RTSS)*. 2010, pp. 259–268.

[119] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. “Multi-core Real-Time Scheduling for Generalized Parallel Task Models”. In: *2011 IEEE 32nd Real-Time Systems Symposium* (2011), pp. 217–226.

[120] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. “Parallel Real-Time Scheduling of DAGs”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3242–3252.

[121] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. “Implicitly Parallel Programming Models for Thousand-Core Microprocessors”. In: *ACM/IEEE 44th Design Automation Conference (DAC-07)* (2007), pp. 754–759.

[122] A. Avizienis. “Fault-Tolerance: The Survival Attribute of Digital Systems”. In: *Proceedings of the IEEE* 66.10 (1978).

[123] A. Avizienis and J. P. J. Kelly. “Fault Tolerance by Design Diversity: Concepts and Experiments”. In: *Computer* 17.8 (1984), pp. 67–80.

[124] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. “Parameter Variations and Impact on Circuits and Microarchitecture”. In: *DAC ’03: Proceedings of the 40th Annual Design Automation Conference* (2003), pp. 338–342.

[125] S. Borkar. “Designing Reliable Systems From Unreliable Components: The challenges of Transistor Variability and Degradation”. In: *Ieee Micro* (2005), pp. 10–16.

[126] T. C. May. “Alpha-Particle-Induced Soft Errors in Dynamic Memories”. In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9.

[127] R. C. Baumann. “Radiation-induced soft errors in advanced semiconductor technologies”. In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–315.

[128] C. L. Chen and M. Y. Hsiao. “Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review”. In: *IBM Journal of Research and Development* 28.2 (1984), pp. 124–134.

[129] F. Alt. “A Bell Telephone Laboratories’ Computing Machine-I”. In: *Mathematical tTables and Other Aids to Computation* 3.21 (1948), pp. 1–13.

[130] F. Alt. “A Bell Telephone Laboratories’ Computing Machine-II”. In: *Mathematical tTables and Other Aids to Computation* 3.22 (1948), pp. 69–84.

[131] R. Hamming. “Error Detecting and Error Correcting Codes”. In: *The Bell System Technical Journal* XXIX.2 (1950).

[132] A. B. Fontainet and R. G. Gallagert. “Error Statistics and Coding for Binary Transmission Over Telephone Circuits”. In: *Proceedings of the IRE* 49.6 (1961), pp. 1059–1065.

[133] R. W. Doran. “The Gray Code”. In: *CDMTCS Research Reports CDMTCS-304* March (2007).

[134] J. M. Berger and B. Mandelbrot. “A New Model for Error Clustering in Telephone Circuits”. In: *IBM Journal of Research and Development* 7.3 (1963), pp. 224–236.

[135] B. Mandelbrot. “Self-Similar Error Clusters in Communication Systems and the Concept of Conditional Stationarity”. In: *Communication Technology, IEEE Transactions on* 13.1 (1965), pp. 71–90.

[136] W. W. Peterson and D. T. Brown. “Cyclic Codes for Error Detection”. In: *Proceedings of The IRE* (1960), pp. 228–235.

[137] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan-Kaufman, 2007, p. 400.

[138] J. Bartlett, J. Gray, and B. Horst. “Fault Tolerance in Tandem Computer Systems”. In: (1986).

[139] J. E. Tomayko. *Computers in spaceflight: the NASA experience*. Vol. 18. NASA Contractor Report 182505, 1988.

[140] J. Anderson and F. Marci. “Multiple redundancy applications in a computer(Design, redundancy, reliability and tradeoffs for Launch Vehicle Digital Computer and Data Adapter/LVDC/LVDA/ for uprated Saturn I and V vehicles, noting logic circuit, memory, input/ output, power supply)”. In: *Annual Symposium on Reliability* (1967), pp. 553–562.

[141] Y. Yeh. “Triple-Triple Redundant 777 Primary Flight Computer”. In: *1996 IEEE Aerospace Applications Conference. Proceedings* 1 (1996), pp. 293–307.

[142] J. von Neumann. “Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components”. In: *Automata studies* (1956).

[143] R. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability”. In: *IBM Journal of Research and Development* 6.2 (Aug. 1962), pp. 200–209.

[144] F. P. Mathur and A. Avizienis. “Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair”. In: *Proceedings of the May 5-7, 1970, spring* (1970).

[145] J. A. Abraham and D. P. Siewiorek. “Algorithm for the Accurate Reliability Evaluation of Triple Modular Redundancy Networks”. In: *Computers, IEEE Transactions* C.7 (1974), pp. 682–692.

[146] S. Mitra and E. J. McCluskey. “Word-Voter: A new voter design for triple modular redundant systems”. In: *vts* (2000).

[147] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffery, and K. Lundgreen. "Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs". In: *Nuclear Science,* 54.6 (Dec. 2007), pp. 2037–2043.

[148] E. Fuller, M. Caffrey, P. Blain, C. Carmichael, N. Khalsa, A. Salazar, and A. V. Fpga. "Radiation Test Results of the Virtex FPGA and ZBT SRAM for Space Based Reconfigurable Computing". In: *1999 MAPLD (Military & Aerospace Applications of Programmable Logic Devices)* 836 (1999).

[149] P. K. Samudrala and J. Ramos. "Selective Triple Modular Redundancy (STMR) Based Single-Event Upset (SEU) Tolerant Synthesis for FPGAs". In: *Nuclear Science, IEEE* 51.5 (Oct. 2004), pp. 2957–2969.

[150] L. Sterpone and M. Violante. "A New Analytical Approach to Estimate the Effects of SEUs in TMR Architectures Implemented Through SRAM-Based FPGAs". In: *Nuclear Science, IEEE* 52.6 (2005), pp. 2217–2223.

[151] J. F. Wakerly. "Microcomputer Reliability Improvement Using Triple-Modular Redundancy". In: *Proceedings of the IEEE* 15.5 (1976), pp. 375–375.

[152] N. Quach. "High Availability and Reliability in the Itanium Processor". In: *IEEE Micro* 20.5 (2000), pp. 61–69.

[153] T. Chen and G. Sunada. "Design of a Self-Testing and Self-Repairing Structure for Highly Hierarchical Ultra-Large Capacity Memory Chips". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions* 1.2 (1993), pp. 88–97.

[154] F. J. J. Aichelmann. "Fault-Tolerant Design Techniques for Semiconductor Memory Applications". In: *IBM journal of research and development* 28.2 (Mar. 1984), pp. 177–183.

[155] I. Kim, Y. Zorian, G. Komoriya, H. Pham, F. P. Higgins, and J. L. Lewandowski. "BUILT IN SELF REPAIR FOR EMBEDDED HIGH DENSITY SRAM". In: *Test Conference, 1998. Proceedings.* 1998, pp. 1112–1119.

[156] V. Schober, S. Paul, and O. Picot. "Memory Built-In Self-Repair using redundant words". In: *Test Conference, 2001.* 2001, pp. 995–1001.

[157] M. Nicolaidis, N. Achouri, and S. Boutobza. "Optimal Reconfiguration Functions for Column or Data-bit Built-In Self-Repair". In: *Design, Automation and Test in Europe Conference and Exhibition, 2003.* 2003.

[158] M. Nicolaidis, N. Achouri, and S. Boutobza. "Dynamic Data-bit Memory Built-In Self-Repair". In: *Computer Aided Design, 2003. ICCAD-2003.* 2003, pp. 588–594.

[159] S.-Y. Kuo and K. W. Fuchs. "EFFICIENT SPARE ALLOCATION IN RECONFIGURABLE ARRAYS". In: *Design Automation, 1986. 23rd Conference.* 1986, pp. 385–390.

[160] V. G. Hemmady and S. M. Reddy. "ON THE REPAIR OF REDUNDANT RAMs". In: *DAC '89: 26th ACM/IEEE Design Automation Conference.* 1989, pp. 710–713.

[161] T. Kawagoe, J. Ohtani, M. Niiro, T. Ooishi, M. Hamada, H. Hidaka, and M. Technology. "A Built-In Self-Repair Analyzer (CRESTA) for embedded DRAMs". In: *Test Conference, 2000.* 2000, pp. 567–574.

[162] Y. Zorian. “Embedded Memory Test & Repair: Infrastructure IP for SOC Yield”. In: 2002, pp. 340–349.

[163] X. Du, S. M. Reddy, and W.-T. Cheng. “At-Speed Built-in Self-Repair Analyzer for Embedded Word-Oriented Memories”. In: *VLSI Design, 2004. Proceedings. 17th International Conference*. 2004, pp. 895–900.

[164] J.-F. Li, J.-C. Yeh, R.-F. Huang, and C.-W. Wu. “A Built-In Self-Repair Design for RAMs With 2-D Redundancy”. In: *Very Large Scale Integration (VLSI) Systems* 13.6 (2005), pp. 742–745.

[165] P. Öhler, S. Hellebrand, and H.-J. Wunderlich. “An Integrated Built-in Test and Repair Approach for Memories with 2D Redundancy Hans-Joachim Wunderlich”. In: *Test Symposium, 2007. ETS ’07. 12th IEEE European*. 2007, pp. 91–96.

[166] T.-W. Tseng, J.-F. Li, and C.-C. Hsu. “ReBISR : A Reconfigurable Built-In Self-Repair Scheme for Random Access Memories in SOCs”. In: *Very Large Scale Integration (VLSI) Systems* 18.6 (2010), pp. 921–932.

[167] D. A. Patterson, G. Gibson, and R. H. Katz. “A Case for Redundant Arrays of Inexpensive Disks (RAID)”. In: *SIGMOD international conference on Management of data*. 1988, pp. 109–116.

[168] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz. “Introduction to redundant arrays of inexpensive disks (RAID)”. In: *COMPCON Spring ’89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*. 1989, pp. 112–117.

[169] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. “RAID: High-Performance, Reliable Secondary Storage”. In: *Computing Surveys (CSUR)* 26.2 (1994), pp. 145–185.

[170] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. “Failure Correction Techniques for Large Disk Arrays”. In: *ASPLOS III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*. 1989, pp. 123–132.

[171] M. Blaum, J. Brady, J. Bruck, and J. Menon. “EVENODD : An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures”. In: *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium*. 1994, pp. 245–254.

[172] Z. Lei, H. Yinhe, L. Huawei, and L. Xiaowei. “Fault Tolerance Mechanism in Chip Many-Core Processors”. In: *Tsinghua Science & Technology* 12.July (2007), pp. 169–174.

[173] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997, p. 1997.

[174] R. Michael and S. David. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979, p. 1979.

[175] K. Deb. “Multi-objective Genetic Algorithms: Problem Difficulties and Construction of Test Problems”. In: *Evolutionary computation* 7.3 (1999), pp. 205–230. arXiv: `arXiv:1011.1669v3`.

[176] R. Steuer. *Multiple Criteria Optimization: Theory, Computation and Application*. John Wiley, 1986.

[177] R. Wang, P. J. Fleming, and R. C. Purshouse. “General framework for localised multi-objective evolutionary algorithms”. In: 258 (2014), pp. 29–53.

[178] K. Deb, S. Pratab, S. Agarwal, and T. Meyarivan. “A Fast and Elitist Multiobjective Genetic Algorithm: NGSA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[179] M. T. Jensen. “Reducing the Run Time Complexity of Multiobjective EAs: The NGSA-II and Other Algorithms”. In: *IEEE Transactions on Evolutionary Computing* 7.5 (2003), pp. 503–515.

[180] F.-A. Fortin and M. Parizeau. “Revisiting the NSGA-II Crowding-Distance Computation”. In: *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO ’13* (2013), p. 623 630.

[181] H. Fang, Q. Wang, Y.-C. Tu, and M. F. Horstemeyer. “An efficient non-dominated sorting method for evolutionary algorithms”. In: *Evolutionary computation* 16.3 (2008), pp. 355–384.

[182] M. Buzdalov, I. Yakupov, and A. Stankevich. “Fast Implementation of the Steady-State NSGA-II Algorithm for Two Dimensions Based on Incremental Non-Dominated Sorting”. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015), pp. 647–654.

[183] H. Wang, L. S. Peh, and S. Malik. “Power-driven design of router microarchitectures in on-chip networks”. In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* 2003-Janua (2003), pp. 105–116.

[184] R. R. Dobkin, Y. Perelman, T. Liran, R. Ginosar, and A. Kolodny. “High Rate Wave-pipelined Asynchronous On-chip Bit-serial Data Link”. In: *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC’07)* (2007), pp. 3–14.

[185] S. Gorard. “REVISITING A 90-YEAR-OLD DEBATE : THE ADVANTAGES OF THE MEAN DEVIATION”. In: *British Journal of Educational Studies,* 53.4 (2005), pp. 417–430.

[186] B. Pascal. *Traite du Triangle Arithmetique,: auec quelques autres petits traitez sur la mesme matiere*. Chez Guillaume Desprez, 1665.

[187] D. Pengelley. “Pascal’s Treatise on the Arithmetical Triangle: Mathematical Induction, Combinations, the Binomial Theorem and Fermat’s Theorem”. In: (), pp. 1–14.

[188] E. J. Borowski and B. M. Jonathan. *Collins Dictionary of Mathematics*. Harper Collins, 1989.

[189] J. Wang, X. Wang, L. Huang, T. S. Mak, and G. Li. “A Fault-tolerant Routing Algorithm for NoC Using Farthest Reachable Routers”. In: *Proceedings - 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing, DASC 2013*. 2013, pp. 153–158.

[190] P. Campos, N. Dahir, C. A. Bonney, M. A. Trefzer, A. M. Tyrrell, and G. Tempesti. “XL-STaGe: A Cross-Layer Scalable Tool for Graph Generation, Evaluation and

Implementation”. In: *IEEE Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2016*. 2016.

[191] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. “Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools”. In: *Graph Drawing Software*. Springer Berlin Heidelberg, 2004, pp. 127–148.

[192] E. R. Gansner and S. C. North. “An open graph visualization system and its applications to software engineering”. In: *Software — Practice and Experience* 30.May (1999), pp. 1203–1233.