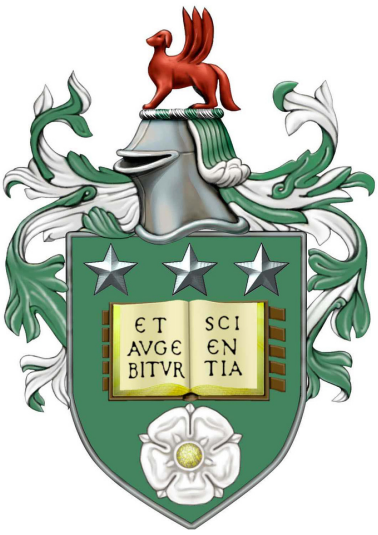
Efficient Domain Partitioning for Stencil-based Parallel Operators

Gaurav Saxena

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy



The University of Leeds
School of Computing
August 2018

The candidate confirms that the work submitted is his/her own, except where work which has formed part of a jointly authored publication has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others.

Some parts of the work presented in Chapters 1, 4, 5 and 6 have been published in the following articles:

Saxena, G., Jimack, P.K. and Walkley, M.A., 2016, July. A Cache-aware Approach to Domain Decomposition for Stencil-based Codes. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on* (pp. 875-885). IEEE.

Saxena, G., Jimack, P.K. and Walkley, M.A., 2017, December. A Cache-aware Approach to Adaptive Mesh Refinement in Parallel Stencil-based Solvers. In *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on* (pp. 364-371). IEEE.

Saxena, G., Jimack, P.K. and Walkley, M.A., 2018. A Quasi-cache-aware Model for Optimal Domain Partitioning in Parallel Geometric Multigrid. *Concurrency and Computation: Practice and Experience*, 30(9), p.e4328.

The above publications are primarily the work of the candidate.

Acknowledgements

It has always been my dream to become a good researcher and I could not have chosen a better path than to pursue this PhD as the first step to achieving this goal. In this long, tough and fruitful journey, it has been an absolute honour and pleasure to have worked under the supervision of Professor *Peter K. Jimack* and *Dr. Mark A. Walkley*. Together, they form a perfect team. Besides being brilliant researchers, they are amazing human beings who make every effort to understand the needs and problems of the student. Though one's imagination is unlimited, it is not possible for me to imagine better supervisors than them. From the bottom of my heart, I profusely thank them for their guidance, care and encouragement. Thank you for letting me pursue my dream and I really hope this journey blossoms into a life-long academic collaboration.

My heartfelt thanks to *Dr. Karim Djemame* for taking out the time for my yearly progress meetings and supplying some extremely interesting ideas. His constructive feedback has helped me to expand the scope of the future work outlined in this thesis. My thanks to *Dr. Brandon Bennett* for very timely approving the travel grants for conferences. From the day I walked in, *Judi Drew* has constantly helped me to adjust to the life of the department. The number of times she has helped me book tickets, register for conferences and forwarded changes in flight schedules, qualifies for a world-record. Thank you to *Dr. Peter Bollada* for all the interesting conversations from across the table. His pleasant personality, jovial nature and helping attitude is certainly contagious. Despite the deficiency of grey cells in my right brain, I have been able to enjoy many conversations with *Dr. Thomas Ranner*. Beyond doubt, he is an excellent academician and person. Many thanks to *Mark Dixon* for the constant support he offered regarding the hardware and software on ARC2/ARC3. His knowledge of hardware is unparalleled. Thanks to *Martin Callaghan* for making all the training sessions extremely interesting. I am very grateful to *Ann S. Almgren* for taking out the time to meet me in the SIAM conference in Atlanta, US, to solve my doubts regarding BoxLib. Thanks to *Weiqun Zhang* for helping me understand some specific subroutines in BoxLib. *Meng-Huo Chen* (Alan) happily shared his PhD experience and wisdom with me whenever we had the time to look away from the screen. I really enjoyed these breaks. My thanks to all the *anonymous reviewers* for their time and suggestions which helped us to improve the work in this thesis. Many thanks to *my teachers* at the University of Edinburgh who sparked my monotonically increasing interest in High Performance Computing. A big thanks to *Dr. Sanjeev Singh* and *Dr. M. K. Das* for helping and having faith in me in the worst of times. I am very grateful to *Sanjay Batra* sir, *Dr. Harmeet Kaur*, *Dr. Baljeet Kaur*, *Negi* sir, *Anita* maam, *Dr. Manoj Aggarwal*, *Sanjay* sir, *Ajit* ji, *Amit* ji, *Bharat* ji and *Shakti* ji for accepting me as a member of the HRC family. My stay at HansRaj college was one of the happiest times I ever had and it was because of you all.

Thanks is a very small word for my wonderful friend *Sabby* who, beyond doubt, is the finest human being I ever came across. My amazing friend *Sachin* taught me the value of hard work and set an example on how to survive despite extreme adversities. Thanks to my good friend *Anshu* whose amazing sense of humour and spiritual knowledge is beyond this realm. *Swapnil Laxman Gaikwad* has constantly shared, advised, helped and encouraged me in this tough journey and forever will I remain indebted to him. I profusely thank my super-amazing friends *Rahul Arora, Divya Jain, Rashmi Shakya* and *Pranav Kumar Singh* who have a golden heart and a hand that is always ready to help. I must thank *Swapnil Sahu* for his timely help and all the good times we shared. A super big thanks to *Kanika Malik* for being a wonderful friend and making time to meet me every time I was about to start a new session at work. Thanks to *Jyoti Balwani* for reminding me again and again that I am a good person. My wonderful childhood friend *Vasuda Arora*, who has been a constant source of support, deserves a big chunk of the thank-you cookie.

Little did I know that my *grandfather's* predictions about my education would turn out to be true. A big thanks to him for gifting my brother and me with a never-ending supply of books. I thank my supremely talented *uncle* who has always been an amazing friend to me. Love and thanks to my *nephews* for loving me even when I have not been able to do anything for them. A huge thanks to my *sister-in-law* for completing our family and making me feel at home during my visit. A big thanks to my *brother* for his intermittent yet excellent streams of advice and for easing my financial burdens. My *grandmother* forged my character and I am thankful to the universe that I was loved by the most noble soul ever to walk on earth. Thanks is a small word for my *father* who stood like a rock in front of me when I needed him the most. Last and the most, I take this opportunity to thank my *mother* who was with me every step of this journey. If I have achieved anything in this life, it is because of her. I have not met anyone as learned and educated as her. Although she is too humble to accept but she has an honorary doctorate in a very rare subject called ... *Life*.

Abstract

Partial Differential Equations (PDEs) are used ubiquitously in modelling natural phenomena. It is generally not possible to obtain an analytical solution and hence they are commonly discretized using schemes such as the Finite Difference Method (FDM) and the Finite Element Method (FEM), converting the continuous PDE to a discrete system of sparse algebraic equations. The solution of this system can be approximated using iterative methods, which are better suited to many sparse systems than direct methods.

In this thesis we use the FDM to discretize linear, second order, Elliptic PDEs and consider parallel implementations of standard iterative solvers. The dominant paradigm in this field is distributed memory parallelism which requires the FDM grid to be partitioned across the available computational cores. The orthodox approach to domain partitioning aims to minimize only the communication volume and achieve perfect load-balance on each core. In this work, we re-examine and challenge this traditional method of domain partitioning and show that for well load-balanced problems, minimizing only the communication volume is insufficient for obtaining optimal domain partitions. To this effect we create a high-level, quasi-cache-aware mathematical model that quantifies cache-misses at the sub-domain level and minimizes them to obtain families of high performing domain decompositions. To our knowledge this is the first work that optimizes domain partitioning by analyzing cache misses, establishing a relationship between cache-misses and domain partitioning.

To place our model in its true context, we identify and qualitatively examine multiple other factors such as the Least Recently Used policy, Cache Line Utilization and Vectorization, that influence the choice of optimal sub-domain dimensions. Since the convergence rate of point iterative methods, such as Jacobi, for uniform meshes is not acceptable at a high mesh resolution, we extend the model to Parallel Geometric Multigrid (GMG). GMG is a multilevel, iterative, optimal algorithm for numerically solving Elliptic PDEs. Adaptive Mesh Refinement (AMR) is another multilevel technique that allows local refinement of a global mesh based on parameters such as error estimates or geometric importance. We study a massively parallel, multiphysics, multi-resolution AMR framework called BoxLib, and implement and discuss our model on single level and adaptively refined meshes, respectively.

We conclude that "close to 2-D" partitions are optimal for stencil-based codes on structured 3-D domains and that it is necessary to optimize for both minimizing cache-misses and communication. We advise that in light of the evolving hardware-software ecosystem, there is an imperative need to re-examine conventional domain partitioning strategies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the stagnation of processor speeds [5], the delivery of continued computational performance improvements over the coming years will be through the exploitation of multicore processors. In the post-Moore [5] era, where researchers are exhaustively hunting for performance in the hardware-software ecosystem, sequential is no longer tolerable. Hence, to quench the thirst for performance, the world is going parallel. The wide heterogeneity in multicore architectures, for example *Manycore, Graphics Processing Units* (GPUs) and the *Field-Programmable Gate Arrays* (FPGAs) to name a few, has already created a requirement for performance portability. Irrespective of the multitude of architectures, the fundamental way in which problems are decomposed and distributed onto these parallel machines has not changed. *Functional decomposition* perceives work to be made up of a set of functions that need to be mapped onto multicores whereas *Domain decomposition or Domain partitioning* divides the largest shareable data-structures among cooperating processes with the universal aim to minimize the communicated volume of data between them.

Scientific Computing employs mathematical techniques to model, simulate and understand physical phenomena on modern computer systems. Undeniably, one of the most important mathematical tools to model phenomena occurring in nature is that of *Partial Differential Equations* (PDEs). In order to approximate such models on computers, it is necessary to map from a continuous domain to a domain represented by a finite set of points or elements spanning the original domain. Such discretizations are subsequently utilized by numerical algorithms to produce an approximated solution to the actual/analytical solution. Parallelism, being pervasive, has heavily influenced the field of Scientific Computing as well, leading to a well documented increase of performance over the years. There is thus an imperative need to continue this quest for computational speed by integrating state of the art techniques of Parallel Computing into Scientific Computing.

PDEs are generally classified as *Elliptic, Parabolic or Hyperbolic* and their solution can be numerically approximated after a suitable discretization has been chosen. A well known method for discretization is the *Finite Difference Method* (FDM) that approximates the derivatives in the PDE using finite differences. Using the FDM on regular domains (e.g. rectangular or hexahedral) leads to the creation of a mesh or grid, where the solution at a particular point is expressed in terms of the weighted average of the solution at some fixed number of neighbouring points. Thus, emerges a fixed geometrical pattern called a *Stencil* which, when coupled with a numerical iterative method, systematically updates the solution at each mesh point. These patterns when implemented on modern computer systems using standard data structures such as arrays, access non-contiguous as well as contiguous memory locations. It is this access pattern of stencil-codes that necessitates an optimal utilization of the *cache-memory* hierarchy as their performance is bounded by the memory bandwidth and *latency*. Further, it is this very access pattern that motivates us to re-examine the fundamental approach of domain partitioning in parallel settings. Our research thus investigates a novel approach of domain partitioning for stencil-based parallel operators and in the process, challenges the orthodox approach of simply minimizing communication volume during domain partitioning.

Traditionally and universally, for load-balanced applications, domain partitioning has been a function of communication volume only. Thus, the aim of this approach has been to obtain equal-sized partitions that minimize the communication volume exchanged between the subdomains. To the best of our knowledge, there is no literature on investigating the effect of cache-misses on domain partitioning. This is the very topic of this thesis, where we take the first step in connecting a pure *Serial Control Parameter* (i.e. cache-misses) to a pure *Parallel Control Parameter* (i.e. Domain Partitioning). To this effect we build a high level mathematical model to quantify/minimize cache-misses and obtain families of high performance domain partitions. For the remainder of this Chapter we aim to provide an overview of the focus of our research, while leaving the details to the chapters that follow.

1.1 Our Focus

Overheads in the form of communication, load-imbalance, limited memory-bandwidth, imbalance between processor and memory speeds, network and memory latencies, and complex memory hierarchies necessitate careful optimization of memory-bandwidth-limited stencil-based codes [8–15]. These overheads can be broadly classified as serial overheads or parallel overheads. Serial overheads, i.e. overheads which would still be present in the absence of multicore architectures, can be differentiated from parallel overheads (overheads which come into existence only because of the utilization of multicores). For example, cache-misses, TLB (Translation Lookaside Buffer) misses and memory latencies are examples of serial overheads which, along with

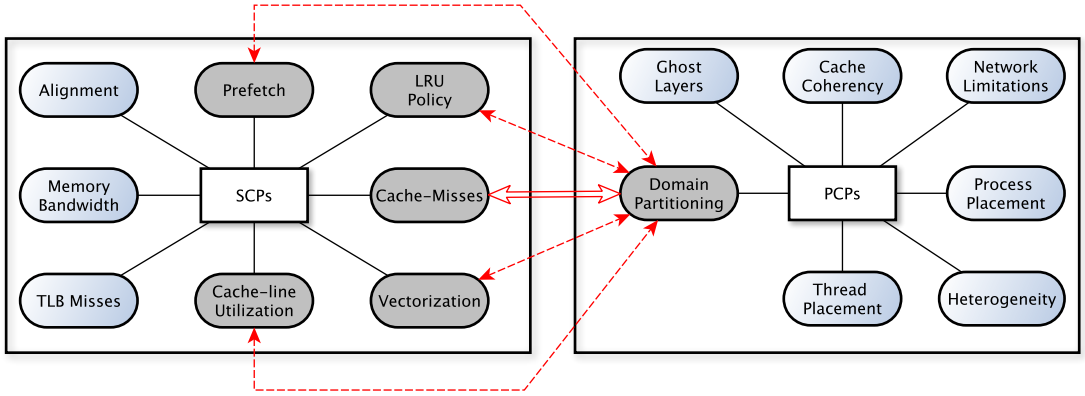


Figure 1.1: Serial Control Parameters (SCPs) Vs Parallel Control Parameters (PCPs): Our focus is on Cache-misses and Domain Partitioning

serial optimizations such as Vectorization, memory alignment etc., we shall collectively refer to as *Serial Control Parameters* (SCPs). Core-to-core latencies, network bandwidth/latencies, non-optimal process placement, non-optimal domain partitions and cache coherence conflicts in shared caches are examples of parallel overheads - a category which, along with the techniques to optimize them, we shall refer to as *Parallel Control Parameters* (PCPs). Figure 1.1 shows some SCPs and PCPs. Our focus is shown with the help of red arrows and grey boxes in Figure 1.1.

More research has explored SCPs as compared to PCPs, whilst there is a complex interaction of SCPs and PCPs which has little literature. Our research focus is to investigate this very interaction but due to the large interaction space between SCPs and PCPs, we restrict ourselves to the most important SCP which we practically (and from the literature [12–14, 16–18]) identify to be Cache-misses and Domain Partitioning - the first fundamental step in distributing data on multicores. We thus take the first step in connecting Cache-misses to Domain Partitioning for single level and multilevel numerical algorithms on structured 3-D domains resulting from finite difference discretizations of Elliptic PDEs. Though applied only to finite difference discretizations of Elliptic PDEs, we argue that our conclusions can be extended to other problems, such as implicit solution of Parabolic PDEs, finite element discretizations using trilinear elements on structured 3-D grids, or any other application that utilizes the same data access and communication pattern as that for the stencil-based codes under study. While constructing a mathematical model to establish this relation, we inherently assume that communication is overlapped with computation but argue that, with appropriate quantitative differences, the model can be applied to scenarios where there is no overlapping.

Since single level codes are frequently less than optimal when employed in isolation to nu-

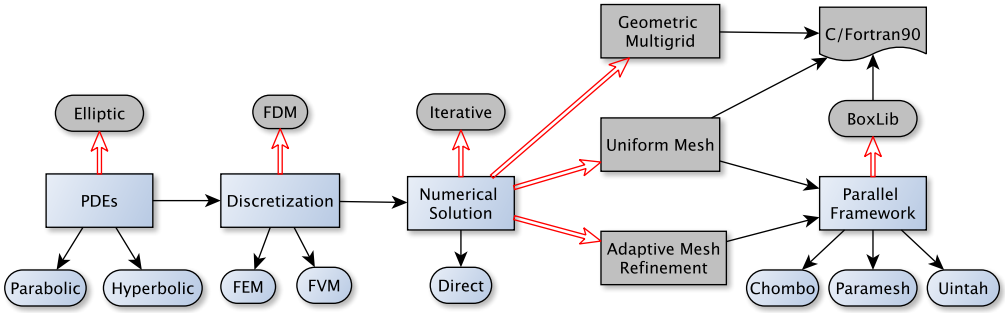


Figure 1.2: Macroscopic view of our research, grey boxes and red arrows show area of focus, FDM (Finite Difference Methods), FVM (Finite Volume Methods) and FEM (Finite Element Methods) are discretizations schemes, PARAMESH [1], Chombo [2], Uintah [3] and BoxLib [4] are parallel Adaptive Mesh Refinement (AMR) frameworks

merically solve PDEs, we also test our hypothesis in adaptively refined meshes implemented in a library called *BoxLib* [19]. *Adaptive Mesh Refinement* (AMR) is a technique that allows a mesh to be refined locally depending on regions of estimated high error, geometric importance or some other parameter. It is an invaluable technique used in Scientific Computing and a key application targeted for *Exascale* Computing [20]. BoxLib is a parallel framework that supports massive, multiscale, multiphysics problems and is written in a combination of C++/Fortran90. We discuss the challenges and the partial success of our model when evaluated in the environment offered by BoxLib. After evaluating our model on adaptive meshes, we then extend the model developed to parallel *Geometric Multigrid* (GMG) - an optimal $\mathcal{O}(N)$ solution algorithm that is based on a hierarchy of grids of decreasing resolution. GMG is one of the most important components of scalable numerical algorithms for solving Elliptic PDEs and is again an extremely important candidate for Exascale systems.

Figure 1.2 shows a macroscopic view of our research, with the area of focus being shown with the help of grey boxes and red arrows. We discretize Elliptic PDEs using Finite Difference Methods (FDMs) (as opposed to Finite Element (FEM) or Finite Volume (FVM) methods) and then use iterative methods (as opposed to direct methods) to solve the PDE on structured regular and adaptively refined meshes. Further, we use Geometric Multigrid for solving the discretized Elliptic PDE. In all the aforementioned scenarios, our aim is to compare the performance of sub-domains derived from our model against traditional communication volume minimizing partitions in parallel settings.

Further, we seek to present our model in the context of all the factors that might influence the choice of sub-domain shape and size. Thus, we qualitatively and quantitatively consider factors such as cache-misses, *prefetching*, *cache-eviction policy*, *Vectorization* etc. (see Figure 1.1), and explore their effect on determining optimal sub-domain dimensions. Though these

factors have been separately well explored in the literature, the focus of our work is on establishing a connection between them and domain partitioning.

1.2 Thesis Contribution

In this section we summarize the main contributions that we claim for this thesis. We itemize these as follows.

- We take the first step in connecting the most important SCP of *cache-misses* to the fundamental PCP of Domain Partitioning for stencil-based codes. To the best of our knowledge, this relation/dependence has not been explored in the literature. We achieve this by building a high level, abstract, quasi-cache aware mathematical model to minimize the cache-misses and obtain families of high performing domain partitions. In this process, we question and challenge the orthodox approach of domain partitioning (for load-balanced codes) based on communication volume only and design experiments to evaluate the same.

- We take a step further to qualitatively establish the effect of other SCPs such as cache-eviction policy, Vectorization etc., on optimal sub-domain dimensions.

- As the model above is constructed using single level meshes, we extend it to multiple levels and evaluate its efficacy on parallel Geometric Multigrid.

- We show that the cache-miss equations for a 7-pt, 19-pt and 27-pt stencil in 3-D have the same form but with appropriate quantitative differences.

- We demonstrate hardware-software independence of our model since the only factor influencing it is the data-layout in memory which is dependent on the language being used to implement the application.

- By implementing a *Cartesian Topology* for single level uniform meshes in BoxLib - an Adaptive Mesh Refinement framework supporting massively parallel, multiscale and multiphysics problems - we are able to show experimentally the efficacy of our model even when communication is not overlapped with computation.

- We propose three dynamic, super-lightweight *cache-tiling heuristics* and evaluate the efficacy of the simplest one of them in our experiments.

- We observe a partial success of our hypothesis when evaluating on adaptively refined meshes. This partial success is important as it shows the communication volume minimizing sub-domain shapes are not always the optimal even in load-imbalanced scenarios.

– Finally, we provide recommendations to application developers and (hopefully) the *MPI Forum* to re-examine and investigate the MPI Cartesian topology returned by the default `MPI_DIMS_CREATE()` function in the context of C (row-major layout) and Fortran (column-major layout) from a performance perspective.

1.3 Thesis Outline

Chapter 2 presents the necessary background along with a literature survey of the associated work. The ideology followed in this Chapter is to explore in depth the concepts which have been utilized in our work but also to span the breadth by broadly discussing associated research.

Chapter 3 describes the hardware test platforms that we use for carrying out experiments, as well as broadly describing the software that we use for implementations and performance measurement. There are two major platforms that we use: *ARC2* and *ARC3* - both resident at and managed by the *University of Leeds*.

Chapter 4 is dedicated to the development of our abstract, high level, mathematical model to obtain cache-minimizing domain partitions using single level, structured 3-D grids. We model the cache-misses by using the *Jacobi iterative* method used in approximating the solution of an Elliptic PDE discretized using the Finite Difference Method. Here, we specifically list our assumptions in creating this model and discuss their generalization to expand the model's applicability.

Chapter 5 evaluates the hypothesis formulated in the previous Chapter on uniform and adaptively refined meshes implemented using BoxLib. We further describe the challenges in adapting BoxLib while evaluating our model.

Chapter 6 is devoted to extending and evaluating the technique developed for single level meshes to parallel Geometric Multigrid, an acceleration convergence scheme that utilizes a hierarchy of grids of decreasing resolution. In addition to cache-misses, we qualitatively discuss how other SCPs affect optimal sub-domain dimensions.

Chapter 7 concludes the research presented in this thesis, discusses its successes and its limitations and presents ideas to open further research avenues.

Chapter 2

Background and Related work

This chapter provides the necessary background and an overview of the work related to the thesis. We start with a discussion of *Partial Differential Equations* (PDEs) since in the current work we focus on *linear, second order, Elliptic* PDEs and their numerical solution using *Iterative* methods [21, 22]. PDEs are routinely used to model phenomena in Elasticity, Fluid Dynamics, Quantum Mechanics, Brownian Motion, Diffusion, Heat Transfer and Electrostatics, among many others [21, 23]. It would not be wrong to say that their numerical solution forms the backbone of *Scientific Computing*. PDE model problems involve continuous dependent variables defined on continuous domains but when their solution is approximated on computer systems, some form of discretization scheme is needed to describe the domain and the unknowns in terms of a finite number of values of a finite number of points or elements. There are many schemes for discretization such as *Finite Difference Methods (FDM), Finite Element methods (FEM), Finite Volume Methods (FVM)* and *Spectral* schemes, etc. We use the FDM in the current work and describe it in some detail. FEM is one of the most widely used schemes and is more flexible than FDM. After furnishing sufficient details, we very briefly touch upon some other discretization schemes. Finite difference discretization of Elliptic PDEs generally give rise to *Sparse* matrices, i.e. matrices which have very few non-zero entries as compared to entries which are zero. An associated concept is that of a *Stencil* - a fixed geometrical pattern used to update the solution on individual points of the domain using weighted contributions of the neighbouring points. We use the 7-pt, 19-pt and 27-pt stencils in 3-D in this work. The Sparse linear systems arising from FDM discretization of Elliptic PDEs can be solved using either *Direct methods* or Iterative methods. We describe the *Gaussian* elimination direct method then move onto describing the iterative methods of *Jacobi* and *Gauss-Seidel* in detail. Jacobi and *weighted Jacobi* iterative methods have been used in the current work to illustrate our research but the same can be extended to the Gauss-Seidel method and its variants.

Since we concentrate on *Domain Partitioning* in parallel settings, we describe various mod-

els of carrying out parallel computing with an emphasis on the *Message Passing Interface* (MPI) [24]. The traditional method of domain partitioning universally revolves around minimizing the communication volume and we describe how the same is associated with the default *MPI Cartesian Topology* for structured domains. After describing the nature of sub-domains obtained after domain partitioning of structured 3-D domains, we discuss the opportunity that MPI provides for overlapping communication with computation for enhancing application performance. Performance metrics such as *Speed-up, Efficiency, Strong Scaling and Weak Scaling* are discussed and used at appropriate points in experiments conducted for validating the concepts developed in the thesis.

Multigrid is a hierarchical, optimal, iterative solver for Elliptic PDEs which accelerates the convergence to the solution. Iterative solvers can thus be used on single grids or form a part of Multigrid [25] in the *smoothing/solve* phase. Our focus is on *Geometric Multigrid (GMG)* in parallel settings in the thesis and hence we also survey the bottlenecks in the parallel implementation of GMG. Since numerical models of physical phenomena can exhibit high errors in localized regions, we next describe the technique of *Adaptive Mesh Refinement (AMR)* that provides the ability to refine localized regions of a mesh depending on various parameters such as a high gradient, high estimated error or the geometric importance of the solution. Both AMR and Multigrid form an integral part of the problems identified for *Exascale* computing [20]. Our background then moves onto describing the basics of cache memory and stresses the fact that a memory bound application must optimally exploit the cache memory for enhancing performance. Stencil codes are memory bound and we next describe the role of caches in optimizing them.

2.1 Partial Differential Equations

A Partial Differential Equation (PDE) [21] is a differential equation which has more than one independent variable i.e. there is some dependent variable u which is an unknown function of at least two independent variables. We can thus write $u = u(x, y, ...)$, where $x, y, ...$ are independent variables. The PDE then is an identity which relates the independent variables, the dependent variables, and the partial derivatives of the dependent variable. The partial derivative of u with respect to x is commonly denoted as $\frac{\partial u}{\partial x}$ or, using a shorter form, u_x. It is to be noted that $u_{xy} = u_{yx} = \frac{\partial}{\partial x}(\frac{\partial u}{\partial y}) = \frac{\partial}{\partial y}(\frac{\partial u}{\partial x})$. The highest derivative that appears in the PDE defines the order of the PDE. A first order PDE in two independent variables x, y and one dependent variable u can be expressed in the general form as:

$$F(x, y, u, u_x, u_y) = 0. \tag{2.1}$$

A second order PDE in two variables in the general form is expressed as:

$$F(x, y, u, u_x, u_y, u_{xx}, u_{yy}, u_{xy}) = 0. \tag{2.2}$$

A solution of a PDE is a function $u(x, y, ...)$ such that it satisfies the equality in at least some region (or completely in a specified domain) of the independent variables.

A PDE is said to be linear if it can be written as

$$\mathcal{L}(u) = g, \tag{2.3}$$

where $\mathcal{L}$ is a differential operator, u is the dependent variable, g is some arbitrary function and the following two conditions hold

1. $\mathcal{L}(u + v) = \mathcal{L}u + \mathcal{L}v$,
2. $\mathcal{L}(cu) = c\mathcal{L}(u)$,

for dependent variables u, v and an arbitrary constant c. A PDE which is not linear is *non-linear*. A PDE in which terms of the highest order derivatives are linear is called a *quasilinear* PDE [26]. The principle of *Superposition* for linear, homogeneous PDEs states that if u_1 and u_2 are solutions of a PDE, then their linear combination is also a solution. This principle of Superposition does not hold for non-linear PDEs though it is sometimes possible to transform non-linear PDEs to linear PDEs and exploit the principle of Superposition [23].

A linear, second order PDE where $u = u(x, y)$ can be represented in the general form as

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial^2 u}{\partial x \partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} + D(x,y)\frac{\partial u}{\partial x} + E(x,y)\frac{\partial u}{\partial y} + F(x,y)u = G(x,y). \tag{2.4}$$

Linear, second order PDEs are classified as *Elliptic, Parabolic* or *Hyperbolic* depending on the relation between the coefficients of the higher order derivatives. Thus, at some point (x_0, y_0), if $B^2(x_0, y_0) - 4A(x_0, y_0)C(x_0, y_0)$ is

1. < 0, then the equation is Elliptic at (x_0, y_0) ;
2. $= 0$, then the equation is Parabolic at (x_0, y_0) ;
3. > 0, then the equation is Hyperbolic at (x_0, y_0).

It is important to note that the Equation (2.4) maybe Elliptic at a point (x_0, y_0) and Parabolic or Hyperbolic at some other point (x_1, y_1). As an example, the linear, second order *Tricomi* equation [23]:

$$\frac{\partial^2 u}{\partial x^2} + x\frac{\partial^2 u}{\partial y^2} = 0, \tag{2.5}$$

is Hyperbolic in $x < 0$, Elliptic in $x > 0$ and Parabolic at $x = 0$. A second order, linear PDE is Elliptic (or Parabolic or Hyperbolic) in a region Ω if and only if it is Elliptic (or Parabolic or Hyperbolic) at every point in Ω. If the coefficients in Equation (2.4) are independent of x, y then the equation is said to be a constant coefficient PDE. A PDE that we use in the current work is the *Laplace* equation - a linear, second order, Elliptic PDE, which in three independent variables x, y and z, is expressed as

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0. \tag{2.6}$$

The operator $\mathcal{L}$ in the Laplace equation above equals $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ and is conveniently represented as ∇^2 or Δ. $\nabla^2 u$ or Δu is interpreted as the *Divergence* of the *Gradient* of u i.e. $\nabla^2 u = \nabla.(\nabla u)$. In the 3-D standard Cartesian coordinate system, $\nabla = \frac{\partial}{\partial x}\hat{i} + \frac{\partial}{\partial x}\hat{j} + \frac{\partial}{\partial x}\hat{k}$, where $\hat{i}, \hat{j}$ and $\hat{k}$ represent the unit vectors along the three Cartesian axes. Thus, in a compact form the Laplace equation shown in Equation (2.6) is represented as

$$\nabla^2 u = 0. \tag{2.7}$$

A linear PDE in which the function $G(x, y) = 0$ is known as a homogeneous PDE. In other words, a linear PDE in which every term either contains the dependent variable or its derivatives is said to be homogeneous. If the function $G(x, y) \neq 0$, then the linear PDE is an inhomogeneous or non-homogeneous PDE. The Laplace equation can now be more accurately classified as a second order, linear, homogeneous, Elliptic PDE. A solution of the Laplace equation is called a *Harmonic* function [21]. The inhomogeneous version of the Laplace equation gives rise to *Poisson's* equation and the latter is represented as:

$$\Delta u = f, \tag{2.8}$$

where $f \neq 0$ is a given function of the independent variables only.

2.2 Discretization

PDEs are defined on continuous regions when modelling physical phenomena. For example, the steady state heat distribution on a plate as a function of spatial coordinates is defined at all points on the 2-D plate. While formulating the numerical approximation of a PDE on paper or a computer, the number of parameters with which the solution is estimated must be finite. Discretization is the process in which a continuous domain is approximated by a finite set of points or elements. In general the greater the number of points (or elements), the higher the accuracy of the computed numerical solution. Three of the most common methods of discretization are Finite Difference, Finite Element and Finite Volume methods. However, there are other schemes such as Spectral methods which are also used for discretization. The

work in this thesis uses only Finite Difference Methods (FDM) which we describe in detail while very broadly covering the others mentioned above.

2.2.1 Finite Difference Method

The Finite Difference Method (FDM) approximates the derivatives at a point by finite differences over a small interval [22]. Thus, if $U(x)$ is a function dependent on the independent variable x, and its derivatives with respect to x are continuous, then we can expand U about the point x_0 using *Taylor's* theorem as shown below

$$U(x_0 + h) = U(x_0) + hU_x(x_0) + \frac{h^2 U_{xx}(x_0)}{2} + \frac{h^3 U_{xxx}(x_0)}{6} + ..., \tag{2.9}$$

where $h > 0$ is the step size and U_x denotes the first derivative of U, U_{xx} denotes the second derivative of U and so on. A Taylor series is an infinite series and its finite truncation may be used to approximate the value of a function at a point in terms of the value of the function and its derivatives at a neighbouring point. Stated simply, it provides a method to approximate a smooth function as a polynomial [27]. Similarly,

$$U(x_0 - h) = U(x_0) - hU_x(x_0) + \frac{h^2 U_{xx}(x_0)}{2} - \frac{h^3 U_{xxx}(x_0)}{6} + ... \tag{2.10}$$

Adding Equation (2.9) and (2.10) produces

$$U(x_0 + h) + U(x_0 - h) = 2U(x_0) + 2\frac{h^2 U_{xx}(x_0)}{2} + O(h^4). \tag{2.11}$$

The term $O(h^4)$ in Equation (2.11) denotes fourth order terms and above in terms of the *Big-Oh* notation (upper bound). If we assume that the contribution of $O(h^4)$ terms is negligible, then by rearranging Equation (2.11), we can show that the second derivative of $U(x)$ at x_0 can be approximated by

$$U_{xx}(x_0) \approx \frac{U(x_0 + h) - 2U(x_0) - U(x_0 - h)}{h^2}. \tag{2.12}$$

The error in Equation (2.12) is $O(h^2)$. An error of $O(h^2)$ means that if the step size is halved, truncation error is reduced by one fourth. If we subtract equation (2.10) from Equation (2.9), we obtain

$$U(x_0 + h) - U(x_0 - h) = 2hU_x(x_0) + O(h^3). \tag{2.13}$$

Ignoring the terms of $O(h^3)$ and above in Equation (2.13), we can obtain a finite difference approximation of U_x at x_0 of $O(h^2)$ given below

$$U_x(x_0) \approx \frac{U(x_0 + h) - U(x_0 - h)}{2h}. \tag{2.14}$$

Equation (2.14) is called the central difference approximation of U_x. Similarly $O(h)$ forward and backward difference approximations of U_x can be obtained by ignoring the $O(h^2)$ terms in $U(x_0 + h)$ (see Equation (2.9)) and $U(x_0 - h)$ (see Equation (2.10)), respectively. Thus the forward difference approximation i.e. $U_x(x_0) \approx \frac{U(x_0+h)-U(x_0)}{h}$ and backward difference approximation i.e. $U_x(x_0) \approx \frac{U(x_0)-U(x_0-h)}{h}$ are both first order approximations in space.

Consider a uniform 2-D mesh (or grid) which has equidistant mesh points on each axis. Thus, the i^{th} mesh point on the X-axis is located at a distance of ih from the origin where h denotes the mesh spacing on the X-axis. Similarly for a point j on the Y-axis, its distance from the origin is jk, with k representing the mesh spacing in the Y direction. In general a point $P_{i,j}$ has coordinates (ih, jk) for $i, j = 0, 1, 2, 3, ...$ If we denote the value of $U = U(x, y)$ (the unknown variable) at point $P_{i,j}$ by $U(ih, jk) = U_{i,j}$, we can represent the central finite difference approximation of $U_{xx} = \frac{\partial^2 U}{\partial x^2}$ by

$$U_{xx}(ih, jk) \approx \frac{U((i+1)h, jk) - 2U(ih, jk) + U((i-1)j, jk)}{h^2} = \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2}. \tag{2.15}$$

A similar expression for U_{yy} can be written based on Equation (2.15). We can extend the above discussion to three dimensions and derive the central difference approximation for the Laplace equation (see Equation (2.6) and (2.7)) in 3-D as:

$$\nabla^2 u \approx \frac{u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1} - 6u_{i,j,k}}{h^2} = 0. \tag{2.16}$$

It is assumed that the grid spacing in all three directions in Equation (2.16) is equal to h. Our focus remains on finite difference discretizations of linear, second order Elliptic PDEs in this thesis.

2.2.2 Finite Element Method

The Finite Element Method is a very powerful method for discretization and can be used for extremely complicated geometries [28, 29]. The first step in the method is to divide the domain into finite elements. Most commonly, these elements can be lines in 1-D, triangles or quadrilaterals in 2-D and tetrahedral or hexahedral elements in 3-D. The entire domain must be completely covered with elements i.e. there should be no empty space and, further the elements should not overlap. If we denote the domain by Ω and the i^{th} finite element with E_i, then

$$\Omega \approx \bigcup_{i=1}^{M} E_i, \tag{2.17}$$

where M is the total number of finite elements. If we denote by $\tilde{E}_i$ the interior region of an element consisting of all the points inside the element but not on the surface (in 2-D and 3-D),

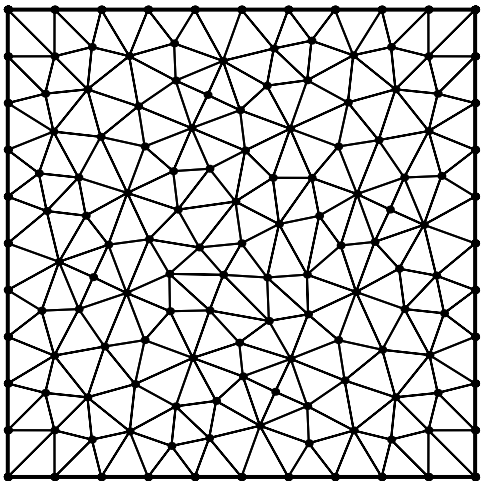


Figure 2.1: Finite Element unstructured mesh covering a square 2-D domain

then

$$\tilde{E_i} \cap \tilde{E_j} = \phi, \forall i \neq j. \tag{2.18}$$

This division of the domain using finite elements results in a finite element mesh with grid points or nodes. The nodes are generally at the vertices of the elements but can be located anywhere on the surface or the interior of the element. Each node has a unique global index but can have multiple local indices (corresponding to each element that shares it). Figure 2.1 shows an unstructured triangular 2-D finite element mesh covering a square 2-D domain.

To construct the global solution, the nodal values of the unknown variable are interpolated using *nodal basis functions* N_k:

$$u = \sum_{k=1}^{N} u_k N_k(x, y, z), \tag{2.19}$$

where u_k is the value of the unknown variable at node $\vec{p}_k$, N is the total number of nodes and N_k has the property that at each node, $\vec{p}_j$,

$$N_k(\overrightarrow{p_j}) \begin{cases} = 1, \forall k = j \\ = 0, \forall k \neq j \end{cases} . \tag{2.20}$$

Further,

$$\sum_{k=1}^{N} N_k = 1. \tag{2.21}$$

To approximate a solution, the values of u_k for $k = 1, N$ must be determined. In this thesis we consider only structured grids in 3-D and although we use the FDM for approximating the

solution on the mesh, a FEM discretization using hexahedral elements could also be considered. Thus, we believe that the concepts that we derive in this thesis are equally applicable when using FEM discretization using regular, eight node (i.e. trilinear), hexahedral elements. For a hexahedral element with eight nodes, the value of the dependent variable u in the element e may be approximated by

$$u^e(x, y, z) = \sum_{i=1}^{8} N_i^e u_i, \tag{2.22}$$

where the N_i^e's are the 8 shape functions satisfying both Equations (2.20) and (2.21) that are non-zero on element e. It should be noted that these shape or basis functions decay linearly along the edges. The approximation of $u^e(x, y, z)$ in 3-D using 8 node hexahedral elements can also be done in terms of global coordinates x, y, z using a symmetric but incomplete polynomial [29] as shown in Equation (2.23) below

$$u^e(x, y, z) = a_0^e + a_1^e x + a_2^e y + a_3^e z + a_4^e xy + a_5^e yz + a_6^e xz + a_7^e xyz. \tag{2.23}$$

The form given in Equation (2.22) is preferred because it uses the FEM basis functions, which allows the efficient assembly into a global system that may be solved for each of the unknowns in (2.19). As a result of application of the finite element assembly, for a linear PDE, the resulting element equations will be in the form of a set of linear equations and can be expressed in the form:

$$[K]\{u\} = \{F\}, \tag{2.24}$$

where $[K]$ denotes the *Stiffness matrix*, $\{u\}$ is a column vector of unknowns at the nodes and $\{F\}$ is a column vector denoting any external influence. Equation (2.24) denotes a system of sparse linear algebraic equations which can be solved by using appropriate Direct or Iterative methods. A detailed description of every step of this method is beyond the scope of the thesis. The interested reader can refer to [28] for more details.

2.2.3 Other Schemes

We provide a very high level overview of some other discretization schemes. The Finite Volume Method is another scheme for discretization which is used frequently in *fluid mechanics* [30]. FVM starts by dividing the domain under consideration into a set of *control volumes* (sub-domains) with nodes. The nodes are defined at the center of or at the vertices of the control volume and each volume generates one equation to find the unknown variable at the nodes. In 3-D, hexahedral and tetrahedral sub-domains are commonly used. After the control volumes are created, balance equations in an integral form are formulated for each volume by integrating the PDE over a control volume. The integrals are evaluated using numerical integration (e.g. Trapezoidal rule or *Simpsons rule* [27]) followed by an approximation of the unknown variables and its derivatives by interpolating the nodal values. The final step involves solving discrete

algebraic equations [30].

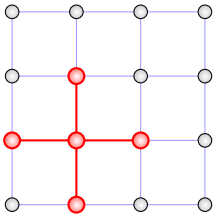
Spectral methods [31] are global methods that represent the solution as a truncated series of the independent variable. As an example, the *Fourier* sine series solution to the heat equation can be truncated after N terms to represent the solution. Spectral methods are global in the sense that the basis functions used to build the solution are each generally non-zero over the whole domain. They can be viewed as belonging to a class of methods to solve PDEs called the *Method of Weighted Residuals* (MWR) that uses trial (or expansion or approximating functions) and test (or weight) functions. The trial functions chosen in Spectral methods are infinitely differentiable global functions as opposed to local element functions in FEM, thus serving as a major distinguishing factor between these two. The type of test functions result in the *Galerkin*, *Collocation* or *Tau* Spectral methods. The trial functions are the same as test functions in the Galerkin Spectral method. The trial functions in the Collocation methods are the *Dirac Delta* functions while the Tau method is very similar to the Galerkin method except for the difference that test functions do not need to satisfy the boundary conditions. For a detailed discussion of these methods, the reader is referred to [31, 32].

2.2.4 Stencils and Sparse Matrices

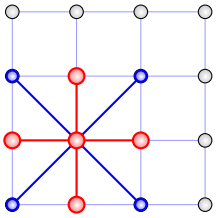
When PDEs are discretized using FDMs (or FEM on a structured grid), the weighted contributions of the values of the neighbours of a point in geometrical space are used to update the numerical solution at a point. In 2-D it is very common to consider a 5-pt stencil or a 9-pt stencil. A 7-pt, 19-pt or a 27-pt stencil is often used in discretized problems in a 3-dimensional space [6, 12, 13]. As an example, if we consider Equation (2.16) in the previous section that shows the finite difference approximation of the Laplacian in 3-D space, the mesh points make a 7-pt stencil. Thus, in a 7-pt stencil, two of the data neighbours in each direction of the mesh point being updated are considered. To visualize, the points in a 7-pt stencil lie at the center of the six faces of a cube. If we add the points in the center of the 12 edges to the 7-pt stencil, we obtain a 19-pt stencil. Further, if we add the eight points at the corners or vertices of the cube to the 19-pt stencil, we obtain a 27-pt stencil. In this thesis we consider the 7-pt, 19-pt and 27-pt stencils in our experiments. Figures 2.2 and 2.3 show these common stencils in 2-D and 3-D, respectively.

The finite difference discretization of linear, second order Elliptic PDEs gives rise to a system of linear equations which must be solved in order to obtain the approximation of the value of the dependent variable at various mesh points. We illustrate this with the help of an example in 1-D, using the FDM. Consider the PDE

$$\frac{\partial^2 u}{\partial x^2} = f(x) \; for \; 0 < x < 1, u(0) = \alpha, u(1) = \beta. \tag{2.25}$$

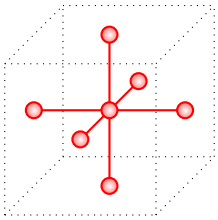


(a) 5-pt stencil

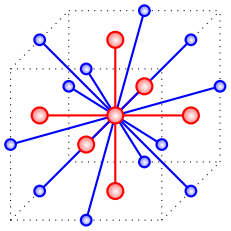


(b) 9-pt stencil

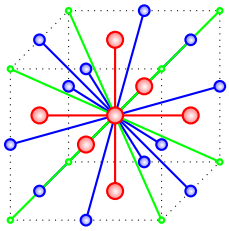
Figure 2.2: Common stencils in 2-D



(a) 7-pt stencil



(b) 19-pt stencil



(c) 27-pt stencil

Figure 2.3: Common Stencils in 3-D

The conditions at $x = 0$ and $x = 1$ in Equation (2.25) specify the *Dirichlet* boundary conditions i.e. they specify the value of the dependent variable itself at the endpoints. Another type of boundary condition called the *Neumann* boundary condition specifies the value at the endpoints in terms of the derivative of u. We assume that the domain is discretized using $m + 2$ equally spaced points i.e. the mesh spacing or width $h = \frac{1}{m+1}$. Let $u_j \approx u(x_j)$ be the approximation of the solution at $x = jh$. It is given that $u_0 = \alpha$ and $u_{m+1} = \beta$ are the boundary conditions. Thus, we have m unknowns, namely, $u_1, u_2, ...u_m$, whose value is to be determined. We can approximate the LHS in Equation (2.25) using a central finite difference scheme to obtain

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = f(x_i) = f_i \; for \; i = 1, 2, 3, ..., m. \tag{2.26}$$

Equation (2.26) specifies a linear system of m equations in m unknowns. For clarity, we can write the equations separately as implied by Equation (2.27) below.

$$\begin{aligned}
\frac{1}{h^2}(u_0 - 2u_1 + u_2) &= f_1, \\
\frac{1}{h^2}(u_1 - 2u_2 + u_3) &= f_2, \\
\frac{1}{h^2}(u_2 - 2u_3 + u_4) &= f_3, \\
&\ldots, \\
&\ldots, \\
\frac{1}{h^2}(u_{m-1} - 2u_m + u_{m+1}) &= f_m.
\end{aligned} \tag{2.27}$$

We can write the m linear equations in m unknowns in Equation (2.27) in the matrix form

$$AU = F, \tag{2.28}$$

where A is the $m \times m$ coefficient matrix given by

$$\frac{1}{h^2}\begin{bmatrix}
-2 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\
1 & -2 & 1 & 0 & \ldots & 0 & 0 & 0 \\
0 & 1 & -2 & 1 & \ldots & 0 & 0 & 0 \\
 & & & \ldots\ldots\ldots\ldots\ldots & & & & \\
0 & 0 & 0 & 0 & \ldots & 1 & -2 & 1 \\
0 & 0 & 0 & 0 & \ldots & 0 & 1 & -2
\end{bmatrix}_{m\times m}. \tag{2.29}$$

U is the column vector of unknowns i.e.

$$U = [u_1 \quad u_2 \quad \ldots \quad u_m]^T, \tag{2.30}$$

and F is the column vector specifying the RHS of every equation i.e.

$$F = [f_1 - \frac{\alpha}{h^2} \quad f_2 \quad f_3 \quad \ldots \quad f_m - \frac{\beta}{h^2}]^T. \tag{2.31}$$

It can be seen that the coefficient matrix give by Equation (2.29) contains many more zero values as compared to non-zero values. To be precise, it contains $3(m-2)+4 = 3m-2$ non-zero values out of a total of m^2 values. Typically if the number of non-zeros is proportional to m rather than m^2, we consider the system sparse.

2.3 Solution of Sparse Linear Systems

A systems of linear equations can be solved in multiple ways. Standard methods which handle a small number of equations are the elimination of unknowns and *Cramers's* rule. Methods which

are capable of handling a larger system of equations are Gaussian elimination, *Gauss-Jordan*, Jacobi, and Gauss-Seidel etc. In general the methods can be separated into two classes: Direct methods and Iterative methods. The current work concentrates on iterative methods and hence after briefly covering direct methods, we describe some iterative methods in detail [27, 33, 34].

2.3.1 Direct methods

Gaussian elimination [27, 35] is one of the oldest and most useful methods falling into the category of direct methods to solve a linear system of equations. It consist of two steps. The Forward elimination step reduces the system of equations to an upper triangular system. This is done by choosing a *pivot* equation and pivot element and eliminating the variable associated with the pivot element from all other equations. This process involves division by the pivot coefficient, which clearly cannot be zero. After the upper triangular system is obtained, the last equation can be solved directly and the result substituted in the second last equation. This process can continue recursively till the first equation and hence this step is known as Back substitution. The total floating point operations (divisions, multiplications and subtractions) in Gaussian elimination is $\mathcal{O}(n^3)$, where n represents the number of unknowns. Gaussian elimination can suffer from many drawbacks such as division by zero (easily overcome for a non-singular system by a "pivoting" strategy), propagation of round-off errors and large errors for *ill-conditioned* systems due to round-off errors. *Partial pivoting* allows one to interchange rows to make the highest coefficient variable as the pivoting element. Thus, before normalization, all columns below the pivot element are scanned for the highest coefficient in order to interchange the rows. Partial pivoting often alleviates the effects of round-off errors and is essential for ill conditioned systems.

The Gauss-Jordan method is a small variation of the Gauss elimination method in the sense that the variable associated with the pivot element is eliminated from all the equations as opposed to its elimination only from subsequent equations (i.e. rows below the row containing the pivot element). Thus, the elimination step in Gauss-Jordan results in an *Identity matrix* and no back substitution is required for the final solution. Since the focus of this thesis remains on solving linear systems of equations arising from finite difference discretizations of Elliptic PDEs which are sparse, we emphasize that it is imperative to take advantage of the sparsity in terms of storage and the solution algorithm. An excellent survey of sparse direct methods, storage of sparse matrices and associated software packages can be found in [36].

2.3.2 Iterative methods

Iterative methods use an initial approximation of the solution to calculate the next approximation. An iterative method is said to converge when the difference between the actual solution and current approximation tends to zero on increasing the iterations [22]. These methods are

useful for large sparse systems of linear equations when the memory and computational time requirements of the direct methods become overwhelmingly high. We now describe the two simplest iterative methods for solving sparse systems of linear equations.

2.3.2.1 Jacobi

The Jacobi method one of the simplest iterative methods. Consider a linear system of three unknown variables having the form $Au = f$ as shown in Equation (2.32).

$$\begin{aligned} a_{11}u_1 + a_{12}u_2 + a_{13}u_3 &= f_1 \\ a_{21}u_1 + a_{22}u_2 + a_{23}u_3 &= f_2 \\ a_{31}u_1 + a_{32}u_2 + a_{33}u_3 &= f_3 \end{aligned} \tag{2.32}$$

The i^{th} equation in Equation(s) (2.32) can be used to express u_i in terms of the RHS and the remaining unknowns. This is shown below.

$$\begin{aligned} u_1 &= \frac{1}{a_{11}}(f_1 - a_{12}u_2 - a_{13}u_3) \\ u_2 &= \frac{1}{a_{22}}(f_2 - a_{21}u_1 - a_{23}u_3) \\ u_3 &= \frac{1}{a_{33}}(f_3 - a_{31}u_1 - a_{32}u_2) \end{aligned} \tag{2.33}$$

Thus, in general, the unknown u_i, $i = 1, 2, 3, ..., n$ can be expressed as

$$u_i = \frac{1}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j - \sum_{j=i+1}^{n} a_{ij}u_j). \tag{2.34}$$

When the solution at the k^{th} iteration is used to compute the approximate solution at the $(k+1)^{th}$ iteration, we can re-write Equation (2.34) as Equation (2.35) below:

$$u_i^{k+1} = \frac{1}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j^k - \sum_{j=i+1}^{n} a_{ij}u_j^k). \tag{2.35}$$

The number of computations on the RHS can reduce drastically if the coefficient matrix is sparse i.e. most of a_{ij}'s are zero. Jacobi iterations can also be written in a matrix form by splitting the coefficient matrix (A) into *diagonal* (D), *strictly lower* (L) and *strictly upper triangular* (U) matrices. The coefficient matrix is thus expressed as $A = D - L - U$. Substituting this in the matrix form of linear equations (see Equation (2.28)), we get

$$Au = f \Rightarrow (D - L - U)u = f \Rightarrow u = D^{-1}(L + U)u + D^{-1}f. \tag{2.36}$$

The notation of Equation (2.36) allows us to write Jacobi iteration (2.35) in matrix form as

$$u^{k+1} = D^{-1}(L+U)u^k + D^{-1}f. \tag{2.37}$$

The matrix $D^{-1}(L+U)$ above is called the point Jacobi iteration matrix. The word "point" stems from the fact that the iterative step computes the value of the solution at a single point in terms of the solution at other points [22]. A variation of the Jacobi iteration method is the weighted Jacobi (ω-Jacobi) iterative method where the solution at point u_i^k also contributes to the approximate solution u_i^{k+1}. For some $\omega > 0$, the weighted Jacobi iteration can be written as:

$$u_i^{k+1} = (1-\omega)u_i^k + \omega(\frac{1}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j^k - \sum_{j=i+1}^{n} a_{ij}u_j^k)). \tag{2.38}$$

2.3.2.2 Gauss-Seidel

The Gauss-Seidel iterative method is a variation of the Jacobi iterative method in the sense that it utilizes the most recently computed approximation of the unknowns to update the solution. The following equation expresses this idea:

$$u_i^{k+1} = \frac{1}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}u_j^k). \tag{2.39}$$

Equation (2.39) suggests that to update any u_i^{k+1}, the most recently computed values of $u_0^{k+1}, u_1^{k+1}, ..., u_{i-1}^{k+1}$ and the old values of $u_{i+1}^k, u_{i+2}^k, ..., u_n^k$ are used. Using Equation (2.39), we can directly write the matrix form in terms of the diagonal (D), strictly lower triangular (L) and strictly upper triangular matrices (U) as

$$u^{k+1} = D^{-1}(f - (-Lu^{k+1} - Uu^k)). \tag{2.40}$$

Pre-multiplying Equation (2.40) by D and re-arranging the terms for u^{k+1} on the LHS, we get

$$(D-L)u^{k+1} = Uu^k + f. \tag{2.41}$$

Pre-multiplying Equation (2.41) by $(D-L)^{-1}$, we obtain

$$u^{k+1} = (D-L)^{-1}Uu^k + (D-L)^{-1}f, \tag{2.42}$$

which gives us the point Gauss-Seidel iteration matrix as $(D-L)^{-1}U$. Although the Gauss-Seidel method generally converges faster than the Jacobi method, it has its drawbacks in the sense that u_i^{k+1} cannot be computed unless $u_0^{k+1}, u_1^{k+1}, ..., u_{i-1}^{k+1}$ have been computed. Thus, this necessitates an ordering on updating the solution - a limitation which is not present in Jacobi's method as it uses the approximate solution at the k^{th} iteration to compute the value

at the $(k+1)^{th}$ iteration. This property of the Jacobi method makes it very suitable for applications in parallel computing although a variant of the Gauss-Seidel method called the *Red-Black Gauss-Seidel* (RBGS) [37] method can be applied when computing the numerical solutions in parallel environments.

To discuss the RBGS method in a parallel environment, consider a mesh in two dimensions. A mesh point having indices (i, j) is given the color Red if $i + j$ is even and the color Black if $i + j$ is odd. This coloring scheme can also be reversed in the sense that a mesh point can be colored Black if $i + j$ is odd, and Red when $i + j$ is even. The algorithm for updating the solution consists of two phases. In the first phase, the red points are updated using only the value of the solution at the neighbouring black points. For example, using a 5-pt stencil in 2-D and the unweighted Jacobi iterative method, the solution at a red point is updated according to the weighted average of the solution at the four neighbouring black points. The updated values of the solution at the red points next to the sub-domain boundary are then communicated to the neighbouring processes. In the second phase, the solution at the black points is updated using the latest value of the solution at the red points. It is important to note that the update of red points (or black points) can be done in any order but the neighbouring processes must synchronize and communicate the updated values of the solution at the red points (or black points) before starting the solution update at the black points (or red points). Although the Red-Black ordering described above works correctly with a 5-pt stencil, it fails with a 9-pt stencil in 2-D. The reason is that the corner points needed for the update of a red point (or black point) are also red (or black). This problem can be overcome by using two additional colors as described in [38]. The technique of multi-color ordering can be extended to more than two dimensions [37, 38].

A further variation of the Gauss-Seidel iteration method is the *Successive Over-relaxation* (SOR) method where the $(k + 1)^{th}$ approximation is the sum of the k^{th} approximation and a correction in a single Gauss-Seidel iteration [22]. Adding and subtracting u_i^k from the RHS of Equation (2.39), we obtain:

$$u_i^{k+1} = u_i^k + \frac{1}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j^{k+1} - \sum_{j=i}^{n} a_{ij}u_j^k). \tag{2.43}$$

The term in parenthesis on the RHS of Equation (2.43) can be seen as a change (or correction or displacement) made to u_i^k by one Gauss-Seidel iteration. If the successive corrections are one-signed, the convergence can be accelerated by using a larger correction term. This is the idea behind successive over-relaxation and is expressed in the general form as shown in Equation (2.44) below:

$$u_i^{k+1} = u_i^k + \frac{\omega}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}u_j^{k+1} - \sum_{j=i}^{n} a_{ij}u_j^k). \tag{2.44}$$

The factor ω in Equation (2.44) is called the acceleration or relaxation parameter and generally $1 < \omega < 2$. For $\omega = 1$, the SOR method reduces to the Gauss-Seidel method [22]. To obtain the iteration matrix of the SOR method, we first subtract u^k from both sides of the Equation (2.40) and multiply the RHS by ω to obtain:

$$u^{k+1} - u^k = \omega D^{-1}(f + Lu^{k+1} + Uu^k - Du^k). \tag{2.45}$$

Re-arranging the terms for u^{k+1} and u^k in Equation (2.45), we obtain the point SOR iteration matrix $H(\omega)$ as:

$$H(\omega) = (I - \omega D^{-1}L)^{-1}((1-\omega)I + \omega D^{-1}U). \tag{2.46}$$

2.3.2.3 Other Iterative Methods

Several other iterative methods exist. A sophisticated and efficient class of non-stationary iterative methods called the *Krylov subspace* methods do not have constant iteration matrices such as Jacobi, Gauss-Seidel and SOR. The idea behind Krylov subspace methods is to generate systematic approximate solutions $u^k \in u^0 + \kappa_n(A, r^0)$ of the solution to $Au = f$, where u^k is the k^{th} iterate of the approximate solution, u^0 is the initial approximation, $r^0 = f - Au^0$ is the initial residual and $\kappa_n(A, r^0)$ is the n^{th} Krylov subspace generated by A from r^0. Formally, $\kappa_n(A, r^0) = span(r^0, Ar^0, A^2r^0, ..., A^{n-1}r^0)$ [39].

The problem of solving the linear system of equations of the form $Au = f$ can be visualized as a problem of minimizing $||f - Au||$ for $u \in R^m$, where $||.||$ denotes some norm (generally the L_2 norm) and R^m is the set of all real vectors. The Krylov subspace $\kappa_n \subseteq R^m$ and $\kappa_m = R^m$. For $n = 1$, $u^1 = \alpha r_0$ and we need to choose an α which minimizes $||f - Au^1||_2$. Similarly for $n = 2$, $u^2 = \alpha r_0 + \beta A r_0$ and we need to choose both α and β such that $||f - Au^2||_2$ is minimized. Krylov methods can generate u^k from u^{k-1} efficiently and they are successful because we can find an $n << m$ such that $||f - Au^n||_2 < \epsilon$, where ϵ is sufficiently small. Well known methods such as *Conjugate Gradient* (CG), *Arnoldi, Lanczos, Generalized Minimum Residual* (GMRES), *Biconjugate Gradient Stabilized* (BiCGSTAB) etc., all belong to the Krylov family [33, 40, 41]. Out of these the CG method is the most efficient but only applicable to symmetric positive definite matrices. GMRES is the most general method applicable to all matrix types but is less efficient as compared to the CG method. Another method called the *MinRES* method is applicable to symmetric matrices which need not be positive definite. A detailed description of these methods falls out of the scope of the thesis, but the interested reader can refer to [33].

2.3.2.4 Multilevel Iterative Methods

Iterative methods such as the Geometric Multigrid (GMG) and *Algebraic Multigrid* (AMG) are specializations of the general class of multilevel iterative methods. Multilevel iterative methods use a hierarchy of approximations of decreasing resolution. Although iterative methods such as

$\omega-$Jacobi and RBGS can effectively remove the high frequency error components on a fine grid, they fail to effectively eliminate the low frequency error components. The idea behind using these coarser approximations in multilevel iterative methods is to accelerate the reduction of the lower frequency components of the error (as high frequency error components of a coarser representation) and thus improve the overall convergence. Multilevel methods can be classified as *Additive* or *Multiplicative*. The main difference between these is that in Multiplicative methods, the update of the solution using iterative methods is carried out sequentially i.e. one level after the other, whereas in the Additive Multilevel schemes, these operations can be performed in parallel for various levels. In both the methods, the inter-grid transfer operations are carried out sequentially [25, 42, 43]. Multiplicative methods are generally used as stand-alone solvers and can be applied to asymmetrical problems, while Additive methods are used as *preconditioners* to accelerate other iterative methods such as the Conjugate Gradient (CG). The standard or classical Multigrid is an example of a Multiplicative Multilevel method [25]. We expand on the Multigrid methods briefly, later in this chapter, and describe the Geometric Multigrid method in detail in Chapter 6. For a detailed discussion of Multiplicative and Additive methods, the interested reader can refer to [42].

2.4 Parallel Computing

Parallel computing involves the division of an algorithm into multiple tasks and their concurrent, coordinated solution on multiple *Processing Elements* (PEs). The aim of parallelism is to reduce the time to solution and to achieve scalability. Parallelism is ubiquitously found in pipe-lining of instruction execution, manipulation of multiple data units in vector registers, threads or processes running on multicore processors, GPUs (Graphics Processing Units) and many-core processors, etc [5]. Some form of synchronization or communication is usually needed by parallel programs and it is because of this overhead that the practically achievable time to solution may not coincide with the theoretical projection of the solution time.

The results of parallel execution can differ from the results of sequential execution for applications involving floating point arithmetic operations. This is because the result for such applications becomes dependent on the order in which operations are done and due to the approximate representation of floating point data. Floating point arithmetic as defined in the IEEE-754 standard [44] is commutative but not associative [45, 46]. IEEE standard is the de-facto industry standard for representing floating point numbers and is ubiquitous across implementations. The finite precision of the mantissa and exponent necessitates rounding off/truncation during long accumulations/reductions [45]. For example, the result of adding n numbers on a single core can differ from the result of the same addition when the computations are divided among multiple processes/threads as the order of addition and intermediate rounding-off/truncation may propagate non-deterministic numerical errors. If the specific order

of addition as dictated by the source code is imposed in a parallel environment to ensure reproducibility, a performance penalty must be paid for imposing the particular sequence. Thus, small variations between results obtained from the sequential and parallel execution of applications involving floating point arithmetic are an acceptable trade-off to permit optimized hardware implementations of floating point operations.

2.4.1 Models for representing Parallel Computation

A *Task/Channel* model [47] is the fundamental model of representing parallel computation where a Task represents a sequential program and local memory. Tasks execute concurrently and are capable of sending messages, receiving messages, creating new tasks and terminating. Tasks are interfaced to the environment via *outports/inports* and Channels represent message queues for connecting outport/inport pairs. These message queues can be created/destroyed dynamically. Messages can carry Channel identifiers to identify the outport/inport pair.

A *Message Passing* model is a variation of the Task/Channel model in the sense that messages are passed or received from named tasks and not using Channel identifiers [47]. The *Message Passing Interface* (MPI) [48] standard is a formalization of the Message Passing model. Although there is no restriction on the creation and destruction of Tasks in the Message Passing model, in practice the implementations of the MPI standard rarely use these features. Generally, a fixed number of identical tasks are created at start-up and this *Single Program Multiple Data* (SPMD) paradigm executes the same program on different data. Though rare, a *Multiple Program Multiple Data* (MPMD) design is also feasible. The current MPI standard 3.1 specifies the bindings only for C/Fortran and the bindings for C++ stand deprecated. MPI, beyond doubt, has become the de-facto standard for distributed computing.

Data Parallelism is another model of parallel programming where the granularity of data computation is small. The difference from Message Passing is that here the compiler automatically generates a SPMD model and hides the communication from the user. The user in turn specifies how data is to be divided, for example, in a round-robin manner or using static partitioning. As an example, *High Performance Fortran* is an extension to the *Fortran* programming language that implements data parallelism [47].

The Shared Memory model allows the Tasks to share a common memory, the access to which can be controlled by using *locks* and *semaphores* [49–51]. As an example, OpenMP [50] uses threads which communicate by reading or writing to the common shared memory. It is easier to parallelize a problem using OpenMP as compared to MPI but it is more difficult to write a deterministic program and debug the race conditions among threads [47, 51].

2.4.2 Parallel Performance

The parallel performance of a program can be measured in several ways and we now describe the most common methods of quantifying this performance. Speedup is defined as the ratio between the sequential execution time and the parallel execution time. Thus, Speedup = $\frac{T_s}{T_p}$, where T_s is the execution time of the best sequential implementation on a serial system and T_p is the execution time using p processing elements. More precisely, if $\psi(n,p)$ represents the Speedup of a problem of size n being parallelized by p processors[1], $\sigma(n)$ is the cost of the purely sequential part, $\phi(n)$ is the cost for the purely parallel part and $\kappa(n,p)$ is the synchronization cost/parallel overhead, then

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\frac{\phi(n)}{p}+\kappa(n,p)}. \tag{2.47}$$

A realistic assumption is that the part $\phi(n)$ is not perfectly parallel and hence this explains the presence of the inequality sign in the expression for $\psi(n,p)$ [52]. Further, as we increase the number of processors, the cost $\frac{\phi(n)}{p}$ decreases (in theory at least) but the cost of inter-processor communication increases ($\kappa(n,p)$).

The above definition of Speedup assumes that the size of the problem is kept constant and not varied when new processors (or cores) are added. In literature this is known as Strong Scaling, as opposed to Weak Scaling where the size of the problem per processor is kept constant. Thus, Speedup is only a key performance indicator for the problem of Strong Scaling and not Weak Scaling. In general, when the only constraint is that the time taken to execute the problem of a given size should be minimized, Strong Scaling is used. If a problem size is to be scaled up to multiple processors, possibly because of high memory requirements, Weak Scaling is utilized [53]. Further, for a (theoretical) perfectly parallel program $\kappa(n,p)=0$, $\sigma(n)=0$ and we obtain $\psi(n,p)=p$ for p processors. For a completely serial program $\phi(n)=0$, $\kappa(n,p)=0$ and thus, $\psi(n,p)=1$. Hence,

$$1 \leq \psi(n,p) \leq p.$$

Amdahl's law states that the maximum Speedup achievable by a parallel program is limited by the serial fraction of the program [52, 54], as illustrated by Equation (2.47).

Efficiency is defined as the Speed-up per processor and is a measure of processor utilization [52]. Thus, denoting $\varepsilon(n,p)$ as the Efficiency of parallelizing a problem of size n with p processors,

$$\varepsilon(n,p) \leq \frac{\sigma(n)+\phi(n)}{p(\sigma(n)+\frac{\phi(n)}{p}+\kappa(n,p))}.$$

With arguments similar to that in case of parallel Speedup, we get the bounds for $\varepsilon(n,p)$ (using

[1]We assume the word processors is synonymous with cores while discussing parallel performance.

the theoretical argument for a perfectly serial and perfectly parallel program) as:

$$0 \leq \varepsilon(n,p) \leq 1.$$

There are other metrics such as the *Karp-Flat metric* that defines the experimentally determined serial fraction to take into account the contribution of the $\kappa(n,p)$ term. This metric helps to point out other reasons for parallel in-efficiency - load-imbalance being an example. It is defined as

$$e = \frac{\frac{1}{\psi(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}},$$

where $p > 1$ is the number of processors and $\psi(n,p)$ is the parallel Speedup [52]. A detailed description of other metrics is beyond the scope of the thesis.

2.4.3 MPI

MPI (Message-Passing Interface) [48] is a specification of the interface for a message passing library. It is largely based on the Message Passing model and extends it in some cases. It is important to note that it is a specification and not an implementation. The main focus of MPI is to address the movement of data from the address space of one process to the address space of another process. All MPI operations are expressed in terms of functions (or methods or subroutines) and the C and Fortran language bindings for these operations are part of the MPI standard. The latest version of the MPI standard as of date is version 3.1 [24].

The most basic operations in MPI fall into the class of *Point-to-Point* operations, a class which specifies the functions with which messages can be sent from one process to another process. In this class of operations there is only one sender and only one receiver. The MPI processes are identified by means of *ranks* in a *communicator*, the latter referring to a collection of processes which can communicate with each other. The class of *Collective* operations specify methods by which all processes in a communicator can participate in a communication. For both these categories, MPI provides *blocking* as well as *non-blocking* versions of functions. The non-blocking versions become extremely useful when communication is to be overlapped with computation. In addition to these classes, MPI provides several other classes of functions which address communicator management, dynamic process creation (and management), parallel I/O, *One-sided* communications and others. A detailed description of these can be found in the MPI standard [24].

2.4.4 Hybrid Programming using MPI and OpenMP

A Hybrid programming model uses a combination of distributed memory and shared memory programming. This model maps much more closely to the overall architecture of shared memory nodes interconnected by high speed networks used in current High Performance clusters today

as compared to only a message passing or a shared memory model. The most popular choice for implementing this model is a combination of MPI for distributed memory access and OpenMP for shared memory programming. The Hybrid model lies between a pure MPI implementation and a pure OpenMP implementation in the sense that a single MPI process (or rank) usually contains more than one thread. In general each thread runs on a separate core. The Hybrid programming model can be a superior solution because of the reduced number of MPI processes and messages, reduced memory footprint and improved load balance. Researchers stress that determining an optimal Hybrid model i.e. the optimal combination of MPI processes and threads per process for an application is not a trivial task [55]. They further recommend that a benchmarking process is a must [55, 56]. For a single SMP, an efficient MPI implementations must prevent messages from going through the MPI software layers and utilize the shared caches to emulate communication. Most MPI implementations optimize intra-node message passing and thus make the Message Passing model a suitable choice for naive application development [57]. The cost of spawning/waking up threads, frequent synchronization, first touch policy on ccNUMA (cache coherent Non-Uniform Memory Access), access of non-local memory by the communicating thread are some problems associated with OpenMP. Nonetheless, OpenMP's incremental approach to parallelization due to the ease of use of compiler directives, library routines and run-time variables makes it an ideal candidate for hybrid applications.

2.4.5 Domain Decomposition/Domain Partitioning

The first step in a parallel implementation of a problem is the division of computational work or data among processes/cores. *Domain Decomposition* [47, 52] or *Domain Partitioning*[2] is the process of dividing and assigning the largest data-structures associated with the problem domain to multiple cores of a multiprocessor [52, 58]. Another approach is that of *functional decomposition* where the computation is decomposed first and then data is associated with it. Some authors stress that Domain Decomposition be differentiated from Domain Partitioning in the sense that the former is a special technique where individual sub-domains independently solve the global problem without any communication and further, converge to the global solution by adapting to the local solutions of neighbouring processes [59]. Data Decomposition within the field of PDEs can either refer to the separation of domains which can be modelled with different equations or division of large linear systems into smaller problems while preconditioning [42]. In the current work, we parallelize the finite difference discretizations of Elliptic PDEs resulting in sub-domains on individual cores that require communication for solving the PDE.

The domain which is discretized using FDM, FEM or FVM can either result in an unstruc-

[2]We use the terms Domain Decomposition and Domain Partitioning interchangeably in this thesis to refer to the same concept.

tured or structured grid/mesh. The discussion in this thesis remains limited to the Domain Partitioning of structured grids only. Naturally occurring data tends to be 3-dimensional and thus the problem domain can be divided into 1-D, 2-D or 3-D partitions depending on the problem. A 1-D partition on structured grids permits partitions along a single Cartesian direction only. Similarly, a 2-D partition and a 3-D partition allow partitions in 2 and 3 Cartesian directions, respectively. In general a higher dimensional partition gives us the opportunity to use a larger number of Processing Elements (PEs) for the problem [52]. Theoretically, a d-dimensional partition for d-dimensions containing a total of n^d elements can allow us to use n^d PEs. However practically, the cost of communication among n^d PEs can be so high that parallelization may not yield any benefits in terms of application speed-up.

MPI offers a convenient way of specifying Domain Partitioning by allowing one to specify a virtual geometrical arrangement of MPI processes known as an MPI Cartesian Topology [48]. This arrangement is virtual as it need not follow any specific process-to-core mapping. Two functions play a major role in the creation of a Cartesian Topology, namely, `MPI_DIMS_CREATE()` and `MPI_CART_CREATE()`. The C language bindings for the `MPI_DIMS_CREATE()` function is shown in Listing 2.1. This function takes as input the number of MPI processes (`nnodes`) for which a topology is to be created, the dimension (`ndims`) of the topology and also the individual number of processes in each dimension as entries into an array (`dims`) of size `ndims`. If `dims[i]=0` for all $i = 1, ndims$, then the function returns into the `dims[]` array positive values, in decreasing order, that are set as close to each other using an appropriate divisibility algorithm which the standard does not describe. In this thesis, we call this the *default MDC* (`MPI_DIMS_CREATE()`). As an example for `nnodes=64, ndims=3`, the default MDC returns `dims[0]=4,dims[1]=4,dims[2]=4`. Thus, in 3 dimensions the default `MDC` is the closest to a cubic topology. As another example for `nnodes=24, ndims=3`, the default MDC returns `dims[0]=4,dims[1]=3,dims[2]=2`. If the user provides a non-zero, non-negative value of `dims[i]`, it is not altered by the function `MPI_Dims_create()` but in all the cases

$$\prod_{0}^{nnodes-1} dims[i] = nnodes$$

must hold else an error is returned.

```
1 int MPI_Dims_create(int nnodes, int ndims, int dims[])
```

Listing 2.1: `MPI_Dims_create()` function

For a structured domain, it is easy to see that for a given `nnodes` and `ndims`, the default MDC minimizes the surface area of a sub-domain. In three dimensions, thus, the sub-domains are as close to a cube as possible when the topology is the default MDC. It is inter-

esting to note that any permutation of the individual sizes in the `dims[]` array returned by the default MDC minimizes this surface area. Thus, for a default MDC for `nnodes=24` i.e. `dims[0]=4,dims[1]=3,dims[2]=2`, a combination such as `dims[0]=3,dims[1]=4,dims[2]=2` also minimizes the surface area of the sub-domain. If in 3-D we denote the product of three dimensions as $D_x \times D_y \times D_z =$ `dims[0]` $\times$ `dims[1]` $\times$ `dims[2]`, then all 6 combinations, namely, $4\times3\times2$, $4\times2\times3$, $3\times4\times2$, $3\times2\times4$, $2\times4\times3$ and $2\times3\times4$ minimize the surface area. Regardless of the data layout supported by a language, the function `MPI_DIMS_CREATE()` return the same value. Thus, the default MDC using the Fortran version of the function `mpi_dims_create()` also returns a topology of $4 \times 3 \times 2$ with `nnodes=24`. It is important to note that the default MDC only minimizes the surface area of cubic domains and may or may not minimize the surface area of non-cubic domains.

Different MPI implementations use different heuristic algorithms to implement the default `MPI_DIMS_CREATE()` function. The aim of all these heuristics is to produce a balanced partition but the interpretation of a balanced partition is debatable and researchers have found that implementations such as MPICH, MVAPICH2 and OpenMPI can produce weak and strong violations of the MPI specification [60].

We now outline and illustrate the working of the heuristic algorithm used by OpenMPI for implementing the default `MPI_DIMS_CREATE()` convenience function. The algorithm is outlined in Figure 2.4. The algorithm takes as input a given number of processes (say `nnodes`) and dimensions (say `ndims`), and after determining the prime factors of `nnodes`, it then distributes these factors using a greedy heuristic into `ndims` bins. As a final step, the bins are sorted in a descending order to obtain the default decomposition (as output in the `dims[1...ndims]` array).

We illustrate the algorithm with the help of two examples. The first example is for `nnodes=24` and `ndims=3` for which the default `MPI_DIMS_CREATE()` returns $4 \times 3 \times 2$. Using Figure 2.4, the value of `sqnnodes`=5, space allocated for the array `factors`=5 and the space allocated for the array `bins`=3. The prime factors of `nnodes=24` are $2 \times 2 \times 2 \times 3$ and thus the array `factors` contains 2, 2, 2, and 3, with an empty trailing array element. Since there are a total of 4 factors, `nfactors=4`. The number of elements in array `bins` is 3 as `ndims=3` and they are all initialized to one. Since the minimum value in the `bins` array is one initially, the element at position one in the `bins` array i.e. `bins[1]=bins[1]*factors[nfactors]=1*3=3`. We now decrement `nfactors` by one and again find the minimum of `bins[1...ndims]`. This minimum is now found at position two in the `bins` array and hence `bins[2]=1*2=2`. In the third assignment step, `bins[3]=1*2=2`. Now the `bins` array contains 3, 2, and 2 at positions 1, 2 and 3, respectively. Clearly, the next minimum value in the `bins` array is at position 2 and 3. We can choose any of these values but we choose the lowest possible index to break the tie. Thus,

Require: `nnodes`: number of processes, `ndims`: number of dimensions, `dims[1...ndims]`: output array containing processes in each dimension

1: sqnnodes $\leftarrow \lceil\sqrt{nnodes}\rceil$
2: factors $\leftarrow malloc(\lceil \lg_2(nnodes)\rceil \times sizeof(int))$
3: bins $\leftarrow malloc(ndims \times sizeof(int))$
4: $i \leftarrow 1$
5: **while** $nnodes\%2 = 0$ **do**
6: $\quad factors[i++] \leftarrow 2$
7: $\quad nnodes \leftarrow \frac{nnodes}{2}$
8: **end while**
9: $j \leftarrow 3$
10: **while** $j <$sqnnodes **do**
11: $\quad$ **while** $nnodes\%j = 0$ **do**
12: $\quad\quad factors[i++] \leftarrow j$
13: $\quad$ **end while**
14: $\quad j \leftarrow j + 2$
15: **end while**
16: **if** $nnodes \neq 1$ **then**
17: $\quad factors[i++] \leftarrow nnodes$
18: **end if**
19: nfactors $\leftarrow i - 1$
20: Initialize $bins[1...ndims] \leftarrow 1$
21: **while** nfactors > 0 **do**
22: $\quad$ Find minimum i such that $bins[i]$ is minimum
23: $\quad$ bins[i] $\leftarrow factors[nfactors] \times$ bins[i]
24: $\quad nfactors \leftarrow nfactors - 1$
25: **end while**
26: $dims \leftarrow$ SORT(bins)

Figure 2.4: Default `MPI_DIMS_CREATE()` algorithm used by OpenMPI

after assigning the fourth remaining factor in the array `factors`, the value at `bins[2]` becomes `bins[2]=2*2`. We now sort the bins in the descending order and assign it to the `dims` array to obtain the decomposition $4 \times 3 \times 2$ as the final decomposition. It is to be noted that this is a balanced decomposition. The second example below shows how this greedy heuristic approach fails to obtain a balanced decomposition when `nnodes=72` and `ndims=2`.

For `nnodes=72` and `ndims=2`, the `factors` array is allocated space for $\lceil \lg_2(nnodes) \rceil = 7$ elements. The number of elements in the `bins` array is now two as `ndims=2`. The prime factorization of 72 yields $2 \times 2 \times 2 \times 3 \times 3$. Since the number of prime factors is 5, hence `nfactors=5`. Since both `bins[1]` and `bins[2]` initially contain a one, we choose `bins[1]` as the minimum element and assign `factors[nfactors]=factors[5]` (i.e. `bins[1]=bins[1]*factors[5]=1*3`) to it. Thus, `bins[1]=3`. Now, the minimum element is `bins[2]` which also becomes 3 after carrying out `bins[2]=bins[2]*factors[4]=1*3=3`. For assigning `factors[3]`, we again choose `bins[1]` as the minimum and hence `bins[1]` becomes `bins[1]=3*2=6`. Carrying out the same procedure again, `bins[2]=3*2=6`. Now we are left with only one factor i.e. `factors[1]=2` and both `bins[1]=bins[2]=6`. Thus, we choose `bins[1]` as the minimum and it is multiplied with `factors[1]` to obtain `bins[1]=6*2=12`. After sorting the `bins` array (and copying it to the `dims` array) the final output in the `dims` array becomes 12×6. This is where the optimality of the balance is violated as another decomposition 9×8 exists where the difference between the first and the last dimension is smaller than in the decomposition 12×6. Thus, it is not necessary that the algorithm employed by OpenMPI (or any other implementation) will always yield the most balanced decomposition.

The function `MPI_DIMS_CREATE()` only helps to specify the number of processes in each dimension but does not actually create the Cartesian Topology. The function `MPI_CART_CREATE()` is used to create the Cartesian Topology. The C syntax of this function is shown in Listing 2.2. This takes as input the old MPI communicator `comm_old` (for example `MPI_COMM_WORLD`), dimension of the topology `ndims`, processes in each dimension through the array `dims[]`, periodicity in each dimension through the array `periods[]`, a boolean value `reorder` to permit or not permit reordering of ranks and outputs the new Cartesian communicator `comm_cart`. If the topology is periodic in a certain dimension, then the last process is followed by the first process in that dimension. If reordering of ranks is allowed in the new Cartesian Topology then the ranks of the processes in the new communicator maybe changed from the ranks of processes in the old communicator.

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
2 const int periods [], int reorder, MPI_Comm *comm_cart)
```

Listing 2.2: `MPI_Cart_create()` function

2.4.6 Sub-domains

Domain Partitioning results in the creation of multiple sub-domains, each of which can be assigned to a particular MPI process. In a distributed architecture where each sub-domain resides with a separate process, *ghost cells/halo data/guard cells* must be introduced to exchange the data in the address space of a neighbouring process for stencil computations [37, 52, 61]. Generally these ghost cells form a part of the sub-domain but they can be separate buffers as well. For structured stencil problems, e.g., solving a PDE on a unit cube with MPI using a Cartesian topology, each process has a maximum of 6 neighbours if a 1-element deep ghost zone and a 7-pt stencil is used. The depth of the ghost zone can be increased to carry out multiple iterations of updates before the next round of communication begins. With a 4 element deep ghost zone, 4 iterations can be performed at the cost of some redundant computations but the neighbouring data has to be exchanged with a (maximum of) 26 neighbours (faces, edges and vertices of a cube/cuboid) [62]. An estimated 100% memory increase for 32^3 sub-domains and 50% increase in 64^3 sub-domain for a 4-element deep ghost zone has been reported in the literature [62]. Further, it is not necessary that each process will contain an equal number of sub-domain mesh points, especially sub-domains resulting from dividing domains which have dimensions of the form $(2^l + 1) \times (2^m + 1) \times (2^n + 1)$ (in 3-D) [59, 63]. This creates a certain amount of load-imbalance between processes. Another type of load imbalance arises in domains where the work done per-grid point is variable. An example of the latter category is a *Dirichlet-Neumann* [25] boundary value problem where a boundary point adjacent to the Dirichlet boundary has to perform less compute work as compared to a boundary point which is immediately adjacent to the Neumann boundary. We discuss the structure of the sub-domains and the terminology associated with them thoroughly in Chapter 4.

2.4.7 Overlapping Communication with Computation

To hide the communication latency in a multicore cluster, the communication can be overlapped with computation after analyzing the dependency of computation elements on communication elements [64]. The MPI standard provides APIs to several non-blocking versions of point-to-point and collective operations which return the control immediately to the application and progress the communication engine in the background [24]. The non-blocking calls are completed by calling `MPI_Wait()` or `MPI_Waitall()`. Generally applications consist of two kinds of work: independent work that does not generate data for communication and dependent work, the computation of which depends on the communicated elements. Potential for communication overlap in large scale scientific applications has been explored in [64] with the conclusion that fine-grained overlap is not necessary as the main opportunity for overlap is provided by the independent work. In general, communication is carried out asynchronously using the network adapter [65]. A multithreaded model for overlapping has been proposed in [65] as opposed to the non-blocking operations in MPI and the *Operating System bypass* mechanism employed by

Infiniband [66]. Some implementations make progress without further calls to the MPI library whereas others progress only when the MPI library is re-entered (by calling `MPI_Wait()` or `MPI_Test()`) [67]. The multithreaded approach mentioned above was implemented in *FiTMPI* and its design was based on creation of a *worker thread* when `MPI_Init()` is executed. The worker thread continuously polls for a network event and sends/receives any pending data. It is important to note that it is generally the low-level data transfers on the network that can be overlapped with computation and not the process of packing or unpacking of data. In principle though, the MPI implementation should be able to overlap the packing/transfer/unpacking of data with the support of the underlying communication mechanism [68–71].

2.5 Multigrid

Multigrid [25, 37, 63, 72] methods are hierarchical algorithms used to optimally solve certain sparse linear systems of equations having N unknowns in $\mathcal{O}(N)$ time. They are based on the idea of using grids of decreasing mesh resolution [63, 72, 73]. Iterative schemes [25, 37, 63, 72] such as Gauss-Seidel, weighted Jacobi (ω-Jacobi) etc., can remove high frequency error components very effectively, known as *smoothing*, but decrease the low frequency error spectrum very slowly, thus producing an unacceptable convergence rate for large numbers of unknowns. These low frequency error components can be represented as relatively high frequency components on coarser grids [25,59,63]. *Standard coarsening* reduces the number of points by one-eighth in 3-D from the immediate finer grid level, i.e. coarsening is done in all dimensions [37, 59, 63]. When these iterative schemes are applied on the coarser grid, they filter out these high frequency errors and speed up the overall convergence. In general, the smooth or low frequency error modes are associated with large *Eigenvalues* and the high frequency error components are associated with small Eigenvalues of the iteration matrix. These smoothing properties of certain iterative methods and the equivalent system of equations at various levels, i.e. coarser grids, form the basis of Multigrid [59]. A vast repository about Multigrid can be found on-line [74] along with a huge list of references in a file named `mgnet.bib`. Multigrid finds a particular use in *Computational Fluid Dynamics* (CFD) where it has been used to solve problems such as viscous flow around the aircraft and fluid flows in industrial machines [75].

Multigrid can be viewed as a recursive algorithm and is best expressed in the form a *2-grid* algorithm/coarse grid correction algorithm [25, 37, 59, 63, 72]. A 2-grid algorithm works by applying a few iterations of the smoother (ω-Jacobi or Gauss-Seidel), on the finest grid, calculation of a *residual*, *restricting* these residuals to the coarse grid, solving an equivalent linear system of error equations exactly on the coarse grid to approximate the error, *interpolating* the error solution to obtain a better approximate of the solution at the fine grid level and repeating the same procedure until a desired convergence is achieved at the fine level [25,59,63]. This scheme is explained in detail in Chapter 6. This 2-grid scheme when repeated recursively

on coarse levels gives rise to the Multigrid algorithm. Typically the *pre-smoothing* (ν_1) and *post-smoothing* (ν_2) iterations of the smoother vary between one and three for most practical problems [59]. Depending on the order of the traversal between grids, two common types of cycles are categorized as *V-cycles* and *W-cycles* [25, 63]. The shape is dictated by a parameter called the *cycle index* (γ), which determines the number of times the recursive Multigrid algorithm is called at a particular coarse grid level. Thus, $\gamma = 1$ produces a V-cycle and $\gamma = 2$ produces a W-cycle (where each coarse grid level is solved twice in an approximate manner) [59].

A method called the *Full Approximation Scheme (FAS)* may be used when the discretization operator is non-linear. This is called so because a full approximation of the solution at the coarsest grid is solved instead of solving only for the error [63, 76]. Another method called *Newton-Multigrid* is also used in non-linear settings and a comparison of these two methods appears in [77]. When the coarse grid is used recursively to approximate the initial guess on the fine grid, it gives rise to the concept of *nested iteration* [63]. Nested iteration when combined with the recursive Multigrid technique gives rise to *Full Multigrid methods (FMG)*. FMG usually starts on the coarsest grid, solves it accurately, interpolates the solution to the finer grid and then applies a *Coarse Grid Correction (CGC)* scheme or Full Approximation Scheme (FAS) cycle before further interpolating to the next finer level [59, 63].

2.5.1 Type of Multigrid methods

Multigrid methods are broadly classified as Geometric or Algebraic Multigrid methods [63, 75]. Algebraic Multigrid uses no geometric information regarding the grid on which the PDE or any other problem is solved and thus they can be better called Algebraic Multilevel methods rather than Algebraic Multigrid [75]. Though the flexibility of Algebraic Multigrid is unparalleled, the higher throughput of Geometric Multigrid in terms of unknowns solved per second makes it extremely attractive [78]. A discussion of Algebraic Multigrid is beyond the scope of the thesis but a gentle introduction can be located in [63]. The classical Multigrid method refers to the Geometric version and we discuss it exhaustively in Chapter 6 of the thesis.

2.5.2 Parallelization and Coarser Grids

Parallelization introduces a bottleneck when coarser grids in Multigrid are visited due to the low ratio of computation to communication. This problem of inefficient solution on coarse grids does not exist in serial Multigrid codes [11]. Communication aggregation and vertical traffic avoidance do not offer substantial benefits at coarser levels [62]. Further, for very large core counts, it is the coarsest grid which contributes to the maximum percentage of run-time [79] as the time spent in `MPI_Waitall()` increases. Researchers have explored the possibility of vertical and horizontal communication avoidance at coarser levels and found them to be ineffective [80].

When a large number of processors (or cores) are present, the coarsest grid can be solved in two standard ways. The first method is to agglomerate the coarse grid points from every processor onto a single processor and then solve the problem. Two constraints exist for a single processor solve. The complete coarse grid problem should be able to fit into the memory of a processor and the solve time should be optimal. The second method is a generalization of the first method where the coarse grid points from all processors are collected on a subset of processors and the problem is again solved in parallel. The first approach incurs zero communication cost (excluding the cost of agglomeration and transfer of the solution after solving) whereas the second one has lesser communication cost as compared to solving the coarsest grid problem on all the processes [11, 81]. Tasks from processes can be aggregated onto a subset of processes (agglomeration) or the combined task copies can be solved on different subsets of processes (redundant approach) [82]. The redundant approach also embeds in itself a resilient approach i.e. in case of a failure of a node in a subset, the result does not need to be re-computed.

Scalability of the coarsest level solvers is an extremely important issue [62]. The coarsest level solver maybe a direct solver [11] such as *MUMPS* (Multifrontal Massively Parallel sparse direct Solver [83]) or *SuperLU* (SupernodalLU) [84] in both Geometric and Algebraic Multigrid. Coarsest level iterative solvers can vary depending on the problem being solved, i.e. from a constant number of relaxations at the coarsest level to implementing an Algebraic Multigrid solver. As an example, in a comparison based study, the truncated V-cycle was terminated when the coarse level contained a 4^3 domain and twenty-four iterations of the Red-Black Gauss Seidel method were performed at the coarsest level [62]. Researchers have preferred the direct solvers as compared to an aggregation of the coarse grid problem on Blue Gene/P systems which has a number of cores of the order of 3×10^5 [85]. These direct solvers are very difficult to implement as a stable pivot choice is needed [86] and have sub-optimal efficiency. An appreciable number of unknowns can be kept at the coarsest level and a highly parallel solver such as *Chebyshev semi-iterative* solver or unpreconditioned Conjugate Gradient method can also be used [11]. Researchers have made attempts to make a rough estimate of the coarsest grid solve using Conjugate Gradient method with a heuristic $\frac{\sqrt[d]{N}}{2^{l-1}}$, where d is the number of dimensions, N is the number of unknowns and l is the level of the coarsest grid. The obtained coarsest grid was then solved using this CG approximation [79, 85].

In our experiments with parallel Geometric Multigrid, we also fix the number of Jacobi iterations at the coarsest level such that the number of V-cycles does not increase. To fix the iterations, we first solve the coarsest grid problem to a high degree of accuracy and note the number of V-cycles. We then remove the global communication calls (`MPI_Allreduce()`) at the coarsest grid level and fix the coarsest grid iterations to the smallest number such that the number of V-cycles does not increase. To find this least value, the coarsest grid iterations are systematically decreased, until a point is reached where the V-cycles start increasing.

2.6 Adaptive Mesh Refinement (AMR)

The accuracy of the approximate numerical solution of a PDE can be increased by increasing the resolution of the mesh, i.e. decreasing the grid spacing. Since the error in the solution may be undesirably higher in only certain regions, increasing the grid resolution locally in such regions is a more efficient strategy rather than a global increase in the resolution. Thus, the mesh obtained after discretization can be refined locally depending on the error, geometric "interestingness" of the solution or any other relevant parameter. This technique is known as *Adaptive Mesh Refinement* (AMR) [87, 88]. The main goal of AMR thus, is to obtain a desired accuracy of solution with the least possible mesh points. This also implies an optimal use of computational resources. AMR automatically adds mesh points to regions where a greater resolution is desired and removes points from regions where a low resolution solution will suffice [89]. Although AMR is complex to implement, it is extremely useful for applications involving a large gradient change, phase change, discontinuities, and shocks. Further physical examples include high *Reynolds* number flows interacting with solid objects, chemically reacting flows, cosmology simulations (resolution is twelve orders of magnitude) and combustion problems [90].

2.6.1 Structured and Unstructured AMR

AMR can be used for both structured (SAMR) and unstructured meshes (UAMR). UAMR are often based on Finite Element discretizations of unstructured meshes but due to indirect memory references, its implementations on cache-based architectures remain inefficient. SAMR uses logical rectangular grids refined spatially and temporally - categorized either as *patch-based* or *tree-based*. The main advantage of SAMR is the ease with which the neighbours of a mesh point can be decoded, in general, simply through array indices. Since the identification of a neighbouring mesh point is straightforward, the efficiency of the method is expected to be high and thus SAMR is used in applications with strict time constraints [89]. In tree-based schemes, grid elements are stored using *k-way* trees with at least a pointer to the parent and an array of pointers to its children. Additionally, some metadata such as the element type (if multiple geometries are allowed), refinement level, a boolean value to distinguish between a boundary element/non-boundary element etc., is also stored. Further, the leaves of the tree are the active elements and elements are generated upon refinement [91]. The tree-structure demands higher storage space and thus it is non-trivial to decide the splitting of this hierarchical data structure which forces additional interprocessor communication [90] to exchange splitting information. Each node of a tree can contain a single cell or a contiguous block of elements (represented using arrays). The latter gives rise to *block-structured* AMR where even if a single cell within a block is refined, the entire block is refined. Block-structured AMR can be implemented for both patch-based and tree-based schemes. In the orthogonal approach of refining a single cell the advantage is a much more flexible refinement but the disadvantage is the indirect memory references [1]. Small grids in complex applications such as AMR are not recommended because

of the increased metadata, increased ghost cells and associated computations and copying of data between different levels.

2.6.2 Software Packages for SAMR

Some notable software packages for parallel Structured AMR (SAMR) are: *Chombo* [2], *BoxLib* [92] (both from Lawrence Berkeley National Laboratory), *PARAMESH* [1] (NASA) and *SAMRAI* [93] (Lawrence Livermore National Laboratory). A detailed survey of block-structured AMR can be found in [94]. PARAMESH uses a tree-based approach while the other three use a patch-based approach. Since load balancing is a critical issue, several algorithmic approaches such as *Space-Filling Curves* (SFCs), greedy algorithms, sensitivity analysis and *Knapsack* problems have been explored [90]. As an example, PARAMESH uses the *Peano-Hilbert* SFC [1] for load-balancing and BoxLib can either use a Knapsack strategy or SFC.

In a study conducted on scaling Chombo to thousands of cores [95], researchers found the influence of OS to be the performance bottleneck rather than the hardware or application code. The migration from *Catamount* micro-kernel to Compute Node Linux caused a decrease of 10% performance in an AMR benchmark due to complex interactions between Linux *libc* heap management and the memory hierarchy. Since it is difficult to interpret weak scaling in AMR, replication scaling was used to take a hierarchy of grids and data points for a fixed number of cores and replicated for higher concurrencies [95]. The affinity of data and threads (called geographical locality) has been stressed for a good performance of AMR as data and work need to be re-partitioned dynamically.

2.6.3 BoxLib

BoxLib [4, 19] is a parallel, multiscale, multiphysics, patch-based AMR framework for structured grids written in C++ and Fortran90. We use and describe BoxLib in detail in Chapter 5. BoxLib uses a properly nested hierarchy of grids but not based on a tree structure i.e. there is no unique parent-child relationship between grids at two adjacent levels. The smallest unit of abstraction is a *Box* and boxes at each level are distributed independently of the boxes at the next level. BoxLib is the basis of several massive codes such as *MAESTRO* [96] and *CASTRO* [97]. Unfortunately, the BoxLib library is now deprecated but a new framework called *AMReX* [98] targeted at Exascale and similar to BoxLib has been released.

The major computational intensity in BoxLib lies in two types of computations: (i) Pointwise evaluation i.e. expressions of the form $\bar{\phi}_{i,j,k} = \phi_{i,j,k} + k(fx_{i,j,k} + fy_{i,j,k} + fz_{i,j,k})$ and (ii) Stencil evaluations i.e. expressions of the form $\bar{\phi}_{i,j,k} = k\phi_{i,j,k} + m(\phi_{i\pm a,j,k} + \phi_{i,j\pm a,k} + \phi_{i,j,k\pm a})$ where a is some scalar offset [19]. In a comparative study of Hybrid parallelism using a combination of OpenMP and MPI, the division of the entire index range of the set of boxes owned by

a process to the set of threads (*Tiling*) outperformed the strategy of dividing each box among the set of threads (*Striping*) [19] by a factor of `5.6x`. Each tile can only belong to a unique box and thus the tile index space is a subset of the box index space. The strategy of assigning one box to one thread has the disadvantage of leaving some threads idle if the number of boxes per MPI process are less than the number of threads. It is to be noted that tiled code has a significant effect on stencil computations but little/no effect on point-wise computations [19]. Application of loop tiling along with improved loop vectorization resulting from simplification of loops using loop fission in *Nyx* - a hybrid application for cosmological simulations - improved the performance by an order of magnitude on the *Intel Xeon Phi Knights Landing* processor [19]. Tiling in the context of BoxLib exposes more parallelism and reduces the working set size of threads [99]. There is no language support for tiling but manually tiling loops and element loops are introduced to loop over tiles and individual elements, respectively [99]. Determining the size of the tile for BoxLib kernels also remains an important research question. Shifting the burden of tiling from the application programmer to compilers has always been an aim of researchers [6, 100–102].

Regional tiling is a hierarchical scheme in which a grid represents a rectangular index space, a contiguous division of the grid represents a region and a logical division of the index space of a region represents a logical tile. Thus, a grid can be made up of multiple contiguous regions and logical tiles are just index space divisions of the regions which can be varied on a loop-by-loop basis. A special case is *Logical tiling* in which each grid consists of a single contiguous region. While creating tiles, the length of the tile in the contiguous dimension is left uncut [6, 12, 99]. BoxLib uses OpenMP for threading and tiling allows it to use coarse-grained threading instead of fine grained loop-level threading. Specifically, the OpenMP parallel `do` loops are placed around tiles and not individual loops [99].

2.6.4 Error Estimation

There are multiple ways in which errors can be measured. In general when solving a PDE using numerical methods, the actual solution is not known and thus the accuracy of the solution needs to be estimated without full knowledge of the actual solution. For a system of linear equations of the form $Au = f$, we can define the residual r as $r = f - Au^k$, where u^k is the k^{th} iterate of the approximated (computed) solution. Clearly, when the approximated solution $u^k = u^*$, where u^* is the true solution, then $r = f - Au^k = f - Au^* = 0$. Further, since the residual is a vector $\in R^m$, we use some *norm* to check if sufficient accuracy has been attained to stop the simulation. A norm is a mapping from a vectors $u \in R^m$ to the set of non-negative real numbers [34]. The two most common norms are described below.

1. *Infinity* or *Max*-norm: The infinity norm is denoted by $||.||_\infty$ and is defined as the maximum absolute value of the components of the vector i.e. $||e||_\infty = \max_{1 \leq i \leq m} |e_i|$. A bound on

the infinity norm implies that no component in the vector is more than the max-norm.

2. *2-norm*: The 2-norm of a vector e having m components is defined as

$$||e||_2 = \sqrt{\sum_{i=1}^{m} e_i^2}.$$

In the current work we use the test problems to investigate the performance of various domain partitions and since their solutions are known to us, we may choose to use the norm of the error vector calculated from the actual solutions to stop the simulation when sufficient accuracy has been obtained. We use this methodology to terminate our simulations in AMR and use the residual 2-norm in Multigrid as the stopping criterion. In some cases, we choose to fix the number of iterations for performance comparisons. However, the accuracy measurement remains a valuable asset to verify the correctness of our implementations. Errors can be estimated *a priori* or *a posteriori*. In the context of AMR, the *a priori* error estimates based on fundamental error analysis of discretization methods and geometry are insufficient in the presence of sharp changes in the solution or singularities [103]. These are insufficient in the sense that they provide information only on the asymptotic error behaviour and assume that the solution is regular. Thus, *a posteriori* error estimates based on the computed solution are needed to select the regions for further refinement [104]. It is to be noted that we do not use AMR in the traditional way i.e. we fix the refinements at the beginning of the simulation and keep them fixed. This treatment is sufficient to serve our performance studies. Traditionally, AMR starts on a coarse mesh and after an *a posteriori* error estimation selects regions for further refinement. The coarsening and refinement of regions continue till sufficient accuracy is obtained. A detailed discussion of error estimation is beyond the scope of the current work.

2.7 Cache Memory

The memory unit of modern computer systems consists of memories of different speeds and sizes. The most common memory hierarchy in order of decreasing sizes, increasing speeds, increasing cost per byte and decreasing distance from the processor consists of the *hard disk*, the *main* memory (RAM), the *cache* memories (L3, L2 and L1) and the *register* memory [5]. Figure 2.5 shows the typical memory hierarchy of a server system along with the typical size and access times. We shall collectively refer to the L1, L2 and L3 caches as the cache hierarchy. The typical number of processor cycles to access the L1, L2 and L3 caches are approximately 1 - 2, 5 - 10, and 10 - 20, respectively [105] (though these numbers may vary depending on the system).

Although the historical increase in processor clock frequencies has stalled in recent years due to power constraints, the mismatch in the rates at which the processor computes and the main memory delivers data necessitates the introduction of the cache hierarchy. Caches exploit

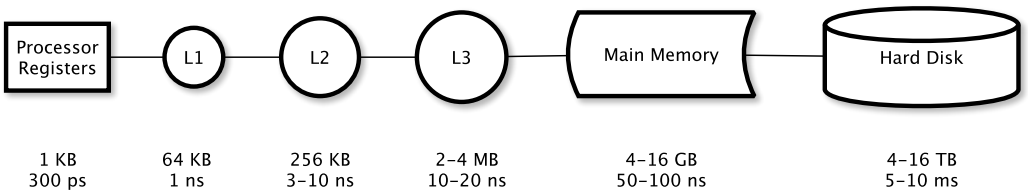


Figure 2.5: Typical memory hierarchy with size and access times in a server system (reproduced from [5])

the principles of *spatial locality*, i.e. data in the vicinity of the data being used is most likely to be accessed, and *temporal locality*, i.e. data which was accessed will be accessed again [5, 105]. Generally the L3 cache contains a copy of the data contained in the L1 and L2 cache and this is termed as the principle of *inclusion*. This principle is also followed by the main memory and the disk storage. There are generally two types of L1 cache: *Instruction* cache (L1i) and the *data* cache (L1d) but the L2 cache is *Unified* (stores both instruction and data). The L3 cache is also Unified, Inclusive and *shared* among several cores in a multi-core system.

A cache-miss results when the data requested by the processor is not found in a cache and thus, data is fetched from the lower levels of the cache hierarchy or the main memory. As can be seen from Figure 2.5, the lower the memory level, the higher the access time and thus the aim is to minimize the cache-miss rate (or maximize the *cache-hit* rate). For reasons of efficiency and spatial locality, a cache miss results in fetching multiple words and not just a single word from the lower levels. This group of words is called a *cache-block* or *cache-line*. For example, instead of fetching a single `double` word of 8 bytes on a cache miss, 8 `double` words are fetched from the main memory. Thus, a typical cache-line size is 64 bytes (i.e. 8 `double` elements or 16 `float` elements). A contiguous collection of blocks in the cache memory is called a *set* and the fetched cache-line from the memory can be placed anywhere in this set. The cache memory can contain many such sets. Such a cache is said to be of *n-way Set Associative* type. In the extreme case where a single set spans the full cache memory, the cache is said to be *Fully Set Associative* as the cache-line can be placed anywhere in the cache. On the other hand if this set consists of only a single cache-line, i.e. $n = 1$, the cache is said to be *Directly Mapped* as there is only one location where the incoming cache-line can be loaded. In other words, a Directly mapped cache has a single block per set and a Fully Associative cache only has a single set. Fully Associative caches are generally used as special purpose caches such as *Translation Look-aside Buffers* (TLBs) [105]. Further, the data in the cache and main memory must be kept consistent. If the data is just being read then the memory is consistent with the cache. The problem occurs when data is written and two policies are used for memory consistency. A *Write Through* policy updates the cache and also the main memory. A *Write*

Back policy only updates the cache but delays writing to the memory for some later point in time. When writing data to the memory, the data can be copied from the cache to a buffer and then written to the memory. A large wait time can result if any incoming cache-line has to wait for some cache-line in the cache to be written directly to the memory. Thus, introducing a buffered scheme prevents full latency times and is used by both Write Through and Write Back.

As mentioned above, it is desired that cache-misses be minimized. The cache-miss rate is defined as the fraction of cache accesses which result in a cache-miss. Similarly the cache-hit rate is defined as the fraction of accesses which result in a hit. Three types of cache-misses have been identified by the *3C's* model [5]:

1. *Compulsory miss*: A compulsory miss results every time a cache block is requested for the first time and is not in the cache memory.

2. *Capacity miss*: When the working set is so large that it just cannot be contained in the cache memory, a capacity miss occurs. Thus, two things must hold true for a capacity miss. First, the cache must be full and, second, the processor must request data that is not in the cache. A cache-block or line must be evicted from the cache memory and hence the cache-miss due to the requested block is categorized as a capacity miss.

3. *Conflict miss*: For a non-fully associative cache, two blocks can map to the same address and hence the first cache block must be evicted from the cache. This results in a conflict miss. A conflict miss can occur even when the cache is not full i.e. the incoming data can theoretically fit into the cache but due to the constraint of mapping to a particular set, it evicts another block.

We can infer from the 3C's model above that for a fully associative cache, only Compulsory and Capacity cache-misses can occur. In a cache-miss rate study where the cache size was varied from 8 KB - 512 KB and the set associativity varied from 1 to 8, the range of Compulsory, Capacity and Conflict cache-misses as a percentage of the total cache-misses was found to be $\approx$ 0.1 - 1.1%, 66 - 100%, and 0 - 35%, respectively [5]. Multicore processors add a fourth type of cache-miss called a *Coherency cache-miss* that are caused by eviction of cache blocks in order to maintain cache coherency across multiple cores.

When a cache-block is evicted from a set in the cache, there are policies to choose the evicted block. Various policies such as the *Random* policy, the *Least Recently Used* (LRU), *Least Frequently Used* (LFU) and *First In First Out* (FIFO) are used [5,105]. The LRU policy chooses the block which was not accessed for the longest period of time. The logic behind using it is that a block which has not been accessed till now will have the lowest probability of being accessed again in the future. In practical implementations, a *pseudo LRU* algorithm approximates the behaviour of the LRU algorithm by associating one bit with each block in a set. Thus, if a cache is 4-way set associative then each set has 4 bits associated with it.

Whenever a block is accessed in a set, the corresponding bit is turned on. When all the bits in a set are in the on state, they are all turned off except for the bit corresponding to the block which was most recently accessed [5]. Thus, when a block is to be evicted, the replacement algorithm can choose from any block for which the bit is in the off state (hence multiple blocks may be available for replacement out of which one can be chosen randomly).

Cache optimization techniques can be broadly grouped into two categories: Hardware based and Software based. An optimization technique such as Prefetching can fall into both the categories. The technique of Prefetching involves fetching data or instructions based on patterns of access into the cache in order to speculatively reduce future cache-misses. Prefetching can be implemented in both hardware as well as software. Special hardware prefetch units can detect strided accesses and keep tables for detecting such patterns. Software prefetching is generally implemented by the compiler by inserting software prefetch instructions after analyzing the access pattern. Prefetching is not a silver bullet and can lead to performance deterioration as well by fetching data which may not be needed, by interfering with cache block replacement policies and by increasing capacity cache-misses etc [106].

2.8 Stencil Codes: Metrics and Optimization

Stencil computations are classified as memory-bound as compared to compute-bound because the memory bandwidth limits their performance rather than computations. To quantify the memory-boundedness of stencil codes, we describe two performance metrics. The first of these is *Arithmetic Intensity* (AI) or *FLOPS/byte*, which is the ratio of Floating Point Operations (FLOPS) to the bytes fetched from the main memory/caches [107]. A lower value of AI indicates memory-bandwidth limited kernels, such as the ones found in Sparse Linear Algebra applications [54]. As an example, consider the weighted Jacobi iteration or smoother:

$$v_{i,j,k} = \omega \times u_{i,j,k} + \bar{\omega} \times (u_{i,j,k+1} + u_{i,jk-1} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i+1,j,k} + u_{1-1,j,k} + H \times f_{i,j,k}). \quad (2.48)$$

Equation (2.48) has 3 multiplication and 7 addition FLOPS. Assuming the data-type is `double`, the memory in bytes that is accessed in Equation (2.48) above is $9 \times 8 = 72$ bytes. It is only the array accesses (such as $v_{i,j,k}$ or $u_{i+1,j,k}$) that are counted as the constant values including ω, H and $\bar{\omega}$, can be stored in processor registers. The theoretical AI of the code above can be calculated as the number of FLOPS divided by the number of bytes that are accessed. Thus, AI $= \frac{10}{72} = 0.14$. Typically, the maximum AI for stencil codes is 1 FLOP/byte [108]. *Operational Intensity* (OI) is a term related to AI which signifies the data movement between caches and the main memory rather than between caches and the processor [54]. It is usually expressed as FLOPS/DRAM byte, with the word DRAM differentiating it from AI. Unfortunately, in practice, because of various NUMA effects and behaviour of modern cache systems, computation of

AI or OI is not a straightforward process [108].

The *Roofline model* is a visual model which provides insight to programmers and designers to better optimize floating point computations [54]. The roofline gets its name from two lines: a horizontal line which illustrates the peak Floating Point performance (a hardware limit) and a diagonal line which denotes the maximum memory bandwidth (in GBytes/sec) for a varying operational intensity. The diagonal line is plotted using the STREAM [109] benchmark and at a varying operational intensity, i.e. STREAM is run at various values of operational intensity and the value is plotted. The angle that the diagonal line makes with the horizontal axis depends on the scales chosen to plot the graph. Further, the advantage of the Roofline model is that it needs to be calculated only once for a multicore system and not once per a computational kernel. On drawing a straight vertical line from the operational intensity axis, if it hits the roof (horizontal line) it means the performance is computation bound and if it intersects the diagonal line - the application is memory traffic bound. Further, the X-coordinate of the *ridge point* (intersection of diagonal with roofline) gives the minimum operational intensity at which peak floating point performance can be obtained. Thus, it is preferable to have the ridge point as far to the left as possible so that even kernels with a very small operational intensity can also achieve the theoretical maximal FLOPS. Further, additional rooflines such as ILP (Instruction Level Parallelism), software prefetching and SIMD (Single Instruction Multiple Data) can be added to the Roofline model. The maximal limits for these can be obtained by running appropriate benchmarks [54].

General cache optimization techniques can be applied to Stencil codes. There have been several efforts to optimize and exploit spatial and temporal principles of the cache memory hierarchy to bridge the gap between the fast processor speed and the comparatively slower memory access times [5, 12–16]. Researchers advocate fetching a higher fraction of data from the higher levels of memory such as registers and L1 cache while reducing the fraction of data fetched from lower levels such as L3 cache and main memory [15]. The major source of cache-misses are nested loops which access the same data repetitively. Data access optimizations are transformations that change the pattern in which data is accessed in the loops to exploit temporal locality [105]. Transformations such as loop *skewing*, loop *peeling*, loop *unroll*, loop *interchange*, loop *fusion* (or jamming), loop *fission*, and loop *blocking* (or tiling) help to make better use of caches and expose available parallelism [105,110]. *Cache tiling/blocking* techniques have been heavily researched and they aim at bringing a sub-domain of data into the cache instead of traversing the entire domain in a single iteration [5,15,16]. The effectiveness of these cache tiling/blocking techniques in modern microprocessors has decreased due to advances in compiler technology and increasing size of on-chip caches [80].

Fusion techniques have been researched in Red-Black Gauss-Seidel (in 2-D, 5-pt stencil)

methods, a variant of Gauss-Seidel, to combine the update of red and black points in a single sweep by updating red points in row i followed by black points in row $i-1$. In the same context, a blocking technique allows multiple updates of a red/black point i.e. re-using cache across multiple time-steps by multiply updating red points in rows i and $i-2$ and black points in rows $i-1$ and $i-3$ [15]. A 2-D blocking technique using a parallelogram shape sweeping through the grid has been proposed as an improvement to the simple blocking technique [15]. Further, the red and black points for unknowns and the corresponding right-hand side values can be stored in different arrays to reduce the traffic between various cache hierarchies, although the total traffic to the main memory remains the same [111].

Initial ground-breaking work proposed the use of partial 3-D blocking for 3-D loops which maximizes the size of the dimension which has continuous data [6]. Analytical cost models for cache tiling fail to address the difference between load and store operations [16]. Further, cache conflict misses occur when the data is read from and written to different grids represented by multi-dimensional arrays in the memory as in the case of Jacobi updates [17]. These cache optimization techniques also interfere with automatic optimization techniques implemented in the hardware and software in the modern microprocessors. These automatic techniques can be called streaming techniques and SIMD (Single Instruction Multiple Data) instructions (also called vectorization) and prefetching fall under it. Researchers have explored specific hardware optimizations along with software optimizations to enhance performance for specific platforms such as the *IA64* (Itanium Architecture) [111]. In order to maximally reap the benefits of parallelism on specific CPUs, explicit SIMDization has to be implemented [80].

Microbenchmarks including the *Stanza Triad* (STriad) and *Stencil Probe* have been created that attempt to act as a proxy for modelling the prefetch behaviour of the actual program [13, 16]. These benchmarks do not account for the packing or unpacking times and the changing latency in the context of using *derived datatypes* in the MPI implementations [48, 61]. Researchers have used hardware performance counters such as cache-misses, Translation Look-Aside Buffers (TLB) misses, mispredicted branches, hardware prefetches, and regression analysis to predict the performance of stencil codes [18]. *Cache oblivious/transcendental* [112] algorithms have been proposed which ignore the hardware characteristics of caches as opposed to Cache aware algorithms which use the cache specifications to minimize cache-misses. The idea behind every memory optimization is to minimize the data accesses between every access to the same memory location [15]. The *ExaStencils* project encourages a *Domain Specific Language* (DSL) for generation of stencil codes which range from an abstract mathematical description to highly optimized code for a particular platform [113]. Further, it stresses the fact that there are a variety of stencils and switching from one form of the stencil to another form is non-trivial in terms of the coding effort. Several domain specific stencil initiatives exist that have different goals such as autotuning, applying cache obliviousness and adding abstractions

to a high level language [113].

2.9 Summary

Partial Differential Equations (PDEs) are widely used to model natural phenomena. It is very difficult to solve them analytically and hence they are discretized by popular methods such as the Finite Difference (FDM), Finite Element (FEM), Finite Volume (FVM) among others and then solved numerically using direct or iterative methods. The FEM is a very powerful and flexible discretization scheme that can be applied to complex domains but in this thesis we concentrate only on the finite difference discretization of linear, second order Elliptic PDEs on regular, structured domains. The finite difference discretization of such equations gives rise to a Sparse system of linear equations which may be solved using point iterative methods such as Jacobi, $\omega-$Jacobi and the Red-Black Gauss Seidel (RBGS). The application of Jacobi/$\omega-$Jacobi (or RBGS) to update the solution at a mesh point after discretization results in a fixed geometrical pattern called a Stencil. A Stencil uses a weighted average of data neighbours to update the solution at a mesh point and a 7-pt, 19-pt or a 27-pt stencil is frequently used in 3-D space.

With the aim to speed-up the numerical solution of a PDE, multicore processors are used. Parallel computing involves the division of a problem into sub-problems and mapping them onto multiple execution units which then simultaneously solve the sub-problem, communicating/synchronizing as and when needed. Domain partitioning or domain decomposition, the first step in parallel computing, is the process of dividing the largest shareable data-structures among cooperating processes. For a load-balanced problem, the orthodox approach to domain partitioning aims to minimize only the total communication volume. For domain level distributed parallelism involving structured domains, this is accomplished by using the default `MPI_DIMS_CREATE()` function of the Message Passing Interface (MPI). MPI is the de-facto standard for programming distributed memory systems, although it can be used to program shared memory systems as well. This aforementioned approach to domain decomposition is followed when Elliptic PDEs are solved numerically on structured meshes.

Stencil-based codes are limited in performance by the memory bandwidth, thus necessitating an optimal use of the cache memory hierarchy for optimal performance. The overlap of communication with computation in a parallel numerical solution of PDEs incurs additional cache-misses as the sub-domain is divided and updated separately as two regions: the inner computational kernel that does not require any data from other sub-domains and the planes at the surface that require data from other processes.

Due to the unacceptably slow convergence rate of these iterative methods when the number of mesh points is large, a multilevel algorithm such as Multigrid may be used. Geometric Multi-

grid is an optimal $\mathcal{O}(N)$, multilevel, multiplicative, iterative method used for solving Elliptic PDEs. Multigrid uses a hierarchy of grids of decreasing sizes and carries out various on-grid and inter-grid operations, such as smoothing, restriction, interpolation and error correction, to accelerate the convergence to the solution. It is challenging to optimize parallel Geometric Multigrid due to the low computation to communication ratio at coarser levels.

Adaptive Mesh Refinement (AMR) is another multilevel technique which allows increasing the mesh resolution in a local region of the global mesh based on error estimates or geometric importance. Multiple parallel frameworks exist that implement AMR and abstract away the communication/synchronization among multicores. BoxLib is a massively parallel, multiphysics, multiscale AMR framework written in C++ and Fortran90 that we use in this thesis. The importance of AMR and Geometric Multigrid in Scientific Computing is indicated by the presence of a large literature base detailing their use in a multitude of application areas and amplified by that fact that they are indispensable candidate algorithms for Exascale. The current thesis focuses on optimizing domain partitioning for stencil-based single and multilevel methods in parallel settings.

Chapter 3

Test Platform: Hardware and Software

This chapter describes in detail the test platform that we use for our experiments. There are two test platforms used - *ARC2* and *ARC3* facilities at the University of Leeds. Although in the broad sense they have similar architecture and software ecosystems, at the finer levels they differ in some aspects. We first describe the terms related to parallel architecture that we use frequently in this chapter and then describe the hardware of each of these facilities. The description of software is divided into two parts. The primary category is that of the compilers and MPI implementations. This category is described separately for each of the facilities but the secondary category consisting of performance profilers and visualization tools does not carry such distinction.

3.1 Architecture

The most common multiprocessor architecture in use today is the *Symmetric Multiprocessor* (SMP). An SMP consists of multiple processing units called *cores* which access the same shared memory. It is called symmetric because all cores have access to the same centralized memory, even if the memory is distributed into multiple modules. Since the cores have the same uniform latency to the memory in an SMP, it is also known as a *Uniform Memory Access* (UMA) multiprocessor [5]. Figure 3.1 shows the schematic diagram of an SMP or UMA multiprocessor.

If the number of cores sharing the centralized memory is increased beyond a certain number, the memory system will not be able to keep up with the bandwidth demands and the increased latency. Thus, an alternative design physically distributes the memory and is known as a *Distributed Shared Memory* (DSM) architecture. A DSM architecture is shown in Figure 3.2 where an interconnection network allows the SMP's to access the physically distributed memory.

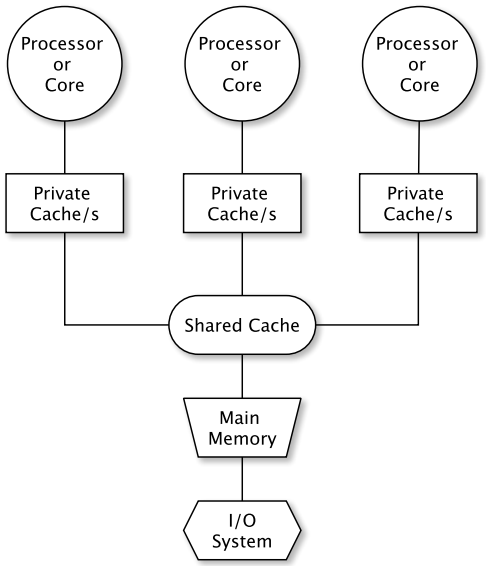


Figure 3.1: Symmetric Multiprocessor (SMP) or Uniform Memory Access (UMA) multiprocessor, each processor or core has uniform latency to main memory and a shared cache.

Since accessing the local memory and non-local memory have different latencies, the DSM architecture is also known as a *Non-Uniform Memory Access* (NUMA) architecture. In both SMP and DSM architectures, *threads* communicate using a shared address space and thus even if the memory is physically distributed, any thread can access any memory location in any module. Currently, multiple multiprocessors are combined to create a compute *node*. The memory in this node is distributed shared memory, i.e. each multiprocessor can access the physical memory attached to the other multiprocessor using a shared address space. Since the local memory has lower latency as compared to the memory attached to the other processors, the node is sometimes called a NUMA node. Each of the processors in a node is housed in a *socket* or a *processor chip*. Thus, a multiprocessor chip consists of multiple cores, each multiprocessor is housed in a chip or socket, and there are generally multiple sockets in a node. These nodes can be connected together with a high speed network such as *Infiniband* [66] to create a *cluster*. Two cores in two different nodes in such a cluster cannot access the physically distributed memory directly without using some form of software *protocols*. Generally *Message Passing* protocols are used for communication in such clusters.

Programming models are experiencing a paradigm shift to efficiently utilize the underlying shared-distributed High Performance Computing infrastructure. For maximal benefit the programming models and their implementations must fully exploit the underlying machine topology and minimize the memory footprint. The most common hybrid parallel programming

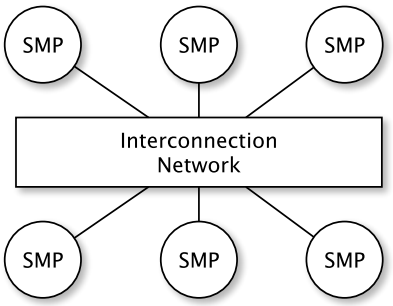


Figure 3.2: Distributed Shared Memory (DSM) or Non-Uniform Memory Access (NUMA) architecture where the SMP's can access the distributed shared memory through an interconnection network, non-local memory access is non-uniform

model which matches the underlying hierarchical hardware is a combination of MPI [48] and OpenMP [50] - MPI being the de-facto standard for distributed memory programming and the latter being the same for shared memory programming [56]. Thus, between a pure message passing approach and a pure virtual shared memory approach for clusters of shared memory nodes interconnected by high speed networks, the hybrid approach i.e. using pure MPI for inter-node and OpenMP within a node promises a better mapping to the cluster architecture.

3.1.1 ARC2

The ARC2 (Advanced Research Computing 2) [114] cluster is a *CentOS 6* based HPC (High Performance Computing) facility. Servers and storage are *HP* based, with each *HP BL460 blade* consisting of a single compute node. Each compute node consists of *2 Xeon E5-2670 Sandy Bridge* processors, each with 8 compute cores (base clock frequency 2.6 GHz, Turbo 3.3 GHz), 16 GB shared memory per processor, thus, making it a total of 32 GB per compute node. The memory is a DDR@1600MHz and achieves a peak memory bandwidth of 102.4 GB/sec per node. Each processor is housed in a socket and has two *QPI* (Quick Path Interconnect) [7] links, with each link running at 16 GB/sec in each direction simultaneously [7]. Each socket or chip forms a NUMA region as the time taken to access the non-local memory is different from that of the local memory. The cluster has a total of 190 blades consisting of 190 nodes or 380 processors - a total of $190 \times 2 \times 8 = 3040$ compute cores. The hierarchy of elements is as follows. Each a *blade* contains a single node, there are multiple blades in a *shelf* and there are multiple shelves in a *rack*. The network that connects the computes nodes is a *QDR* (Quad Data Rate) *Connect-X*, delivering 40 Gbit/sec to the compute blades and storage.

The L1d and L1i cache are 32 KB each, L2 cache is 256 KB (Unified) and 8 cores in a socket share the *Last Level Cache* (LLC) or L3 of 20 MB. L1d and L2 have a cache-line size of 64 bytes and associativity of 8, while L3 has the same cache-line size but an associativity of 20.

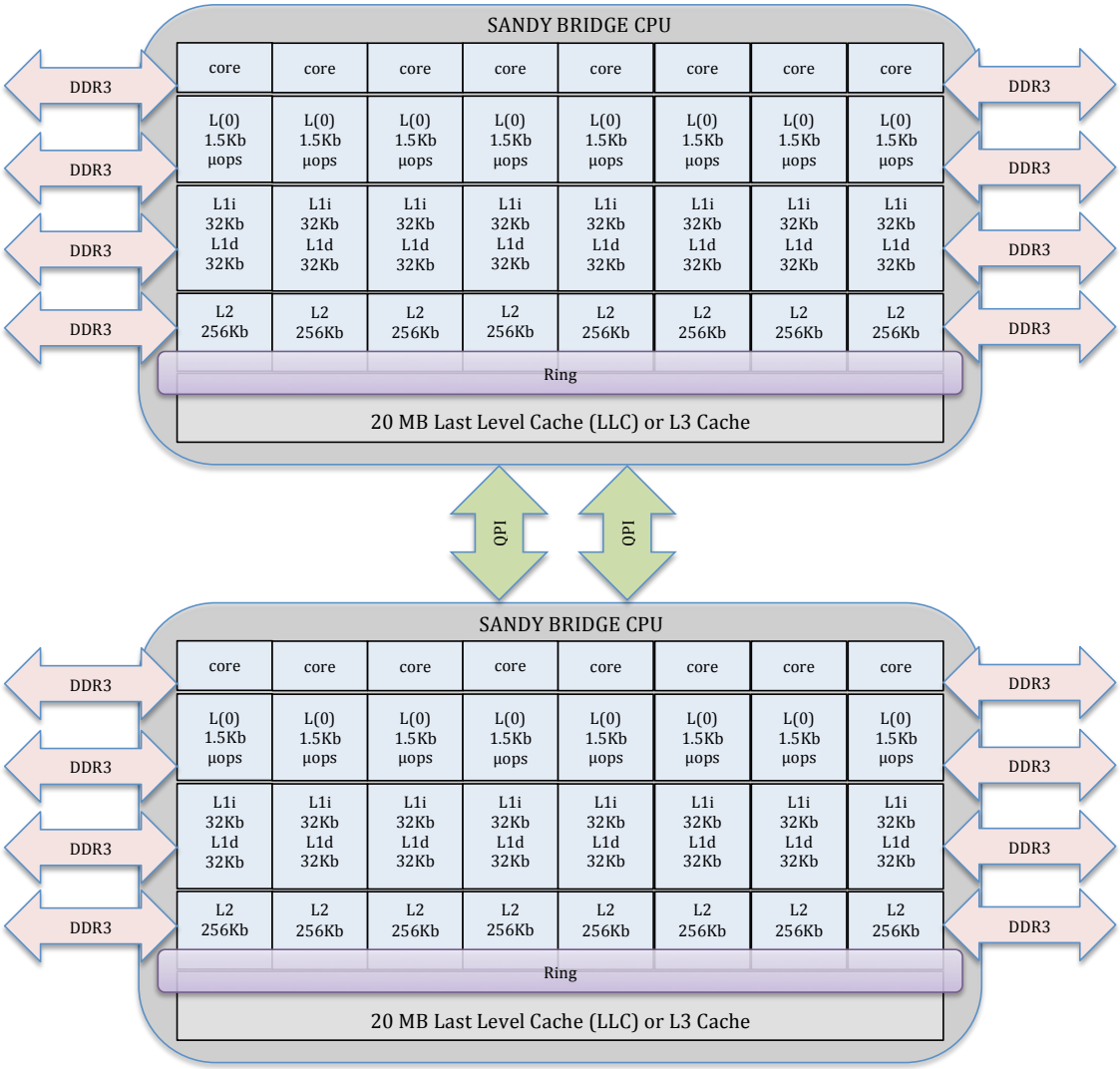


Figure 3.3: Memory hierarchy of an E5-2670 CPU processor and Quick Path Interconnect (QPI)

A schematic diagram depicting the memory hierarchy of a node is shown in Figure 3.3. I/O, PCIe buses, etc., have been omitted in this high level diagram. Table 3.1 shows a summary of important features of this experimental test-bed at the level of a core, processor and node [7].

3.1.1.1 Theoretical FLOPS

The *Intel Sandy Bridge* architecture in ARC2 implements a 256-bit AVX (*A*dvanced *V*ector e*X*tensions) instruction set. Thus, the vector registers are of length 256 bits, i.e. capable of containing either $\frac{256}{sizeof(float)\times 8} = \frac{256}{4\times 8} = 8$ single precision (SP) floating point values or 4 double precision (DP) floating point values. We use *FLOPS* as an acronym for *Floating Point Operations* and not Floating Point Operations per second. Whenever we want to refer to the latter in this thesis, we refer to it as *FLOPS/sec*. In one clock cycle, the Sandy Bridge can achieve 1 floating point Multiplication and 1 floating point addition for AVX-FP_High (256 bits). Thus, considering DP floating points, a total of 4 (Mul) + 4 (Add) = 8 FLOPS can be achieved in a single cycle. Similarly for SP floating point, a total of 8 (Mul) + 8 (Add) = 16 FLOPS can be achieved. The base frequency at which a core runs on ARC2 is 2.6 GHz. Since Hz means cycles per second (unit of frequency), this translates to $8FLOPS \times 2.6 \times 10^9 \frac{cycles}{sec} = 20.8$ double precision GFLOPS/sec per core. For single precision FLOPS, we can double 20.8 to get 41.6 single precision GFLOPS/sec per core. Since there are 8 cores per node, we obtain a theoretical FLOPS rate of $20.8 \times 8 = 166.4$ DP FLOPS/sec or 332.8 SP FLOPS/sec per processor (or socket or CPU). As a single node consists of 16 cores i.e. two sockets or multiprocessors or chips, we need to double the FLOP rate obtained previously to obtain 332.8 DP FLOPS/sec or 665.6 SP FLOPS/sec (per node).

3.1.1.2 Theoretical Memory Bandwidth of ARC2 node

The memory is DDR3@1600MHz and there are 4 channels per socket. The total memory bandwidth per processor then is equal to: $\mathcal{B}$ = 8 bytes × 1600 MHz × 4 channels = 47.68 GB/sec. But Intel specifies a bandwidth of 51.2 GB/sec for the same. The reason is that to convert the $\mathcal{B}$ above into Giga Bytes, 1 KB is approximately taken as 1000 bytes and not 1024 bytes and 1 MB is taken as 1000 KB and not 1024 KB. Similarly, 1 GB is taken as 1000 MB and not 1024 MB. Hence, calculating the $\mathcal{B}$ in such a way produces a value of 51.2 GB/sec - a value specified by Intel [115]. Clearly, for two sockets or a single node this is multiplied by two to get a memory bandwidth of 102.4 GB/sec.

3.1.2 ARC3

The ARC3 (Advanced Research Computing 3) facility is a *CentOS 7* based HPC (High Performance Computing) facility at the University of Leeds. ARC3 has 252 nodes of 24 cores each (standard nodes). A node is made up of two *Intel Xeon Broadwell E5-2650v4* processors (12 cores per CPU or socket or processor). Thus, the total number of cores is 6048. The clock rate

Table 3.1: ARC2 Features: Core, Processor and Node characteristics [7]

ARC2 Core Features	
Base Frequency	2.60 GHz
Turbo Frequency	3.3 GHz
SP FLOPS/cycle	16 (8 Mul + 8 Add)
DP FLOPS/cycle	8 (4 Mul + 4 Add)
SP FLOPS/sec	41.6 GFLOPS/sec
DP FLOPS/sec	20.8 GFLOPS/sec
L0 microperations cache	1.5K micro-ops
L1 cache size	32 KB (L1i) + 32 KB (L1d)
L2 cache size	256 KB (Unified)
HyperThreads/core	2
ARC2 Processor Features	
Processor Code Name	Intel Sandy Bridge-EP (Xeon E5-2670)
No. of cores	8
SP Peak FLOPS/sec	166.4 GFLOPS/sec
DP Peak FLOPS/sec	332.8 GFLOPS/sec
L3 cache size	20 MB (Shared and Inclusive)
L3 cache network	Ring
Memory type	4 channels DDR3 - 2 DIMMS per channel
Memory speed	1600 MHz
I/O controller	On-chip
PCI Lanes	40 Integrated PCIe 3.0
PCIe 3.0 Speed	8 GT/sec
ARC2 Node Features	
Number of Processors (Sockets)	2
Main memory	16 + 16 = 32 GB
Total HyperThreads	16 + 16 = 32
Inter-Socket QPI Links	2
QPI Frequency	8.0 GT/sec
SP FLOPS/sec	665.6 GFLOPS/sec
DP FLOPS/sec	332.8 GFLOPS/sec

Table 3.2: ARC3 Features: Core, Processor and Node characteristics (standard nodes only)

ARC3 Core Features	
Base Frequency	2.20 GHz
Turbo Frequency	2.90 GHz
L1 cache size	32 KB (L1i) + 32 KB (L1d)
L2 cache size	256 KB (Unified)
HyperThreads/core	2
ARC3 Processor Features	
Processor Code Name	Intel Broadwell EP (Xeon E5-2650 v4)
No. of cores	12
L3 cache size	30 MB (Shared and Inclusive)
L3 cache network	Ring
Memory type	4 channels DDR4 - 2 DIMMS per channel
Memory speed	2400 MHz
I/O controller	On-chip
PCI Lanes	40 Integrated PCIe 3.0
PCIe 3.0 Speed	8 GT/sec
ARC3 Node Features	
Number of Processors (Sockets)	2
Main memory	64 + 64 = 128 GB
Total HyperThreads	24 + 24 = 48
Inter-Socket QPI Links	2
QPI Frequency	9.6 GT/sec

for non-AVX instructions is 2.2 GHz and that for the AVX instructions is 1.8 GHz. The total memory per node is 128 GB and is arranged as 8 modules of 16 GB each ($\approx$ 5.3 GB per core). The Last Level Cache (LLC) memory is 30 MB per processor and is shared between 12 cores. The L1i/L1d cache is 32 KB and L2 cache is 256 KB (Unified) for a core. The cache-line size is 64 bytes for all the caches. The set associativity is 8 for L1/L2 and 20 for the shared, inclusive L3 cache. There is additional hardware on ARC3 which we do not utilize in our experiments and hence is not described.

3.1.2.1 Theoretical Memory Bandwidth of ARC3 node

The memory is DDR4@2400MHz and there are 4 channels per socket. The total memory bandwidth per node then is equal to: $\mathcal{B} = 8$ bytes $\times$ 2400 MHz $\times$ 4 channels $\times$ 2 sockets = 150 GB/sec. This gives a value of 75 GB/sec per socket which is 1.8 GB/sec less than what Intel states [116]. If we take a kilo to be 1000 instead of 1024, then $\mathcal{B} = 153.6$ GB/sec per node or 76.8 GB/sec per processor and matches the the bandwidth that Intel specifies [116].

3.2 Software

We group the software into two categories. The first category consists of the language *compilers* and MPI implementations. This is a primary category which we describe separately for ARC2 and ARC3 clusters. The second category consists of performance profiling and visualization tools which we do not describe separately for ARC2 and ARC3.

3.2.1 ARC2 Compilers and MPI Implementations

ARC2 uses the CentOS release version *6.9* (`$lsb_release -a`) and the kernel version 2.6.32-696.18.7.el6.x86_64 (`$uname -a`). Since we develop our programs in the C language, we use the *Intel* C/C++ compiler for compilation. There are multiple versions of the Intel icc compiler present on ARC2, managed using the module environment. Various module numbers for the Intel compiler are: intel/13.1.3.192 (default), intel/15.0.0, intel/16.0.2, intel/17.0.1 and the very recently installed intel/18.0.2. In our experiments, we use intel/16/0.2 and intel/17.0.1 but not intel/18.0.2 due to the unavailability of the latter during the course of the project. The corresponding icc compiler versions are (`$icc --version`): icc (ICC) 13.1.3, icc (ICC) 15.0.0, icc (ICC) 16.0.2, icc (ICC) 17.0.1, and icc (ICC) 18.0.2. The same output is obtained using `$mpicc --version` as well, as the MPI implementation internally uses the underlying C/C++ compiler.

There are multiple MPI implementations installed on the ARC2 cluster, namely, OpenMPI 1.6.5, multiple versions of Intel MPI and Mvapich2/1.9. Out of these we only use the OpenMPI 1.6.5 implementation. Our reasons for choosing this implementation are multiple. First, this seems to be the most popular choice in published literature. Second, it was designed with the goal of supporting Infiniband [66]. Third, just like *Mvapich2* - a derivative of *MPICH2* [117], it is publicly available. According to [118], all OpenMPI versions up till 1.8 have been either declared as retired or ancient. We use an updated version of OpenMPI (version 2.0.2) when using the latest ARC3 cluster and its details are described in the next section. As of writing this thesis, the current stable version seems to be OpenMPI v3.0 [118].

3.2.2 ARC3 Compilers and MPI Implementations

ARC3 uses the CentOS release version *7.4* (`$lsb_release -a`) and the kernel version 3.10.0-693.11.6.el7.x86_64 (`$uname -a`). There are various Intel C/C++ compilers, each activated by choosing the respective module, namely, intel/16.0.2, intel/17.0.1 and intel/18.0.2. The names of the respective icc compilers can be derived using `$icc --version` after loading the appropriate module and is the same as the name of the modules mentioned above. There are multiple GNU modules on ARC3, namely, gnu/6.3.0 and gnu/7.2.0. We use the C compiler gcc 6.3.0 on ARC3 to show the compiler independence of our model. Out of the multiple OpenMPI implementations on ARC3, we use OpenMPI 2.0.2 as the other implementation, namely, OpenMPI 2.1.3, was not available throughout the course of the project. In the Intel

MPI flavour we make use of Intel MPI 2017.1.132. Further, we conduct some experiments with Mvapich2/2.2 to further test the independence of our model from MPI implementations.

3.2.3 Other Tools

We use performance profiling tools to capture the cache-misses and other relevant performance metrics. *TAU* [119] (Tuning and Analysis Utilities) is a tool that can be used for profiling, tracing and sampling an application. It can be used with serial codes as well as parallel codes utilizing MPI, OpenMP and pthreads. The general steps consist of preparing the application for instrumentation, generating a profile and then examining the profile using a command line or a graphical tool. The graphical tool called *Paraprof* produces a visualization of the metrics captured and helps to identify the bottlenecks in the application. By default a single metric is collected, i.e. the time spent in different phases of execution but additional metrics such as cache-misses/hits, Floating Point operations, etc., can be configured using the `TAU_METRICS` environment variable. One file per process is generated and if multiple metrics are specified, each one is written to a different directory starting with the identifier MULTI_. The `TAU_MAKEFILE` environment variable specifies what kind of a parallel program is to be profiled. We only make use of pure MPI programs but the options can include hybrid programs using MPI and OpenMP as well. TAU can be used without recompiling the program though it is recommended that the program should be recompiled using a script file provided by TAU, namely, `tau_cc.sh` for C programs and `tau_f90.sh` for Fortran programs. Without recompilation, the executable provided by TAU can be placed directly with the command used to execute the parallel program. As an example `$mpirun tau_exec <prog>` is a completely valid instrumentation for an MPI program. TAU internally uses the *PAPI* (Performance API) [120] interface for recording various metrics. The commands `$papi_avail` lists the various possible hardware counters supported by environment or architecture and `$papi_choose_event` checks whether the counters specified are compatible with each other.

Scalasca [121] is another performance analysis tool that works by instrumenting, analyzing and then examining the profile/trace. The `scalasca` module uses another measurement infrastructure called *Score-P* [122]. Score-P can be used with other profiling tools as well, such as TAU [119]. Scalasca supports profiling of MPI, OpenMP, and Hybrid MPI+OpenMP programs as well as programs written in CUDA (Compute Unified Device Architecture). The profiling results can be visualized using the *Cube* tool. An advantage of Scalasca over TAU is that the former shows the load-imbalance, late-sender and late-receiver scenarios explicitly, thus, it helps identify performance bottlenecks directly. We use Scalasca with BoxLib to capture the cache-misses of various sub-domain shapes as BoxLib does not interface seamlessly with TAU.

VisIt [123] is a visualization tool that we use to plot the results from *BoxLib*. It is a distributed, open-source, visualization tool that can run on Desktop computers to HPC clusters

having 10^5 cores. VisIt offers a GUI with a wide variety of operators and mathematical manipulations that can be applied to visualizations. As an example, levels in adaptively refined meshes can be coloured or the local refinement in 3-D meshes can be plotted with a wire-mesh. It offers features such as slicing, rotating, and creating a video, among many others.

Chapter 4

Cache-aware Domain Partitioning

With the ubiquitous appearance of multicore processors, a natural step is to parallelize the simulations to minimize the time to produce meaningful results (or increase the accuracy of the results obtained in a given execution time). It is challenging to optimize the process of parallelization due to overheads such as data movement, mismatch in the speeds of the processor and memory, data dependency constraints, algorithmic inefficiencies, loose coupling of software with hardware etc., among many others. As mentioned in Chapter 1, our research broadly lies at the intersection of Parallel Computing and numerical methods for the solution of PDEs. We attempt to optimize their solution on multicore systems by creating a novel technique for Domain Decomposition/Domain Partitioning - the first step in parallel computing which consists of distributing data to individual cores of a multiprocessor system. We provide an insight into why the orthodox approach of domain partitioning based on minimizing the communication volume is not generally the optimal solution. We create and experimentally validate a new model for domain partitioning based on the minimization of cache-misses. Cache-misses are the major performance bottleneck in serial computing and our research focuses on connecting them to domain partitioning in parallel computing. To the best of our knowledge, such a relationship stands unexplored in the literature. With this macroscopic view of our research, we now delve into the details.

4.1 Introduction

Partial Differential Equations (PDEs) [21,124] lie at the heart of numerous scientific simulations depicting physical phenomena. It is very difficult, if not impossible, to solve them analytically and thus, they are discretized and solved numerically [22]. *Discretization* of the problem can be achieved by using, amongst others, the *Finite Difference (FDM), Finite Element (FEM)* or

the *Finite Volume (FVM)* methods [125]. A detailed description of FDM and a brief overview of FEM, FVM and other discretization schemes was provided in Chapter 2. To recollect, Finite Difference Methods are a numerical approximation method to estimate derivatives of any order and can be obtained using *Taylor's* theorem [22]. We only use the Finite Difference method in the current and subsequent chapters but we expect the results to hold for other local forms of discretization as well. Iterative methods such as the Jacobi, weighted Jacobi (ω-Jacobi), Gauss-Seidel, Red-Black Gauss-Seidel (RBGS) etc., can be used to update the solution at various mesh points after discretization (see Chapter 2). A fixed geometrical shape called a *Stencil*, is used to define the approximate solution at each mesh point using a weighted average of the solution at some fixed neighbouring mesh points. As illustrated in Chapter 2, a 7-pt, 19-pt and 27-pt stencils are the most commonly used stencils in 3-D. As the number of mesh points become larger, the time to solution increases. Parallel computing is used to decompose/divide the domains (grid) into sub-domains (sub-grids) and reduce the time to solution by letting the processor cores work independently on sub-problems, exchanging data when needed. In this chapter we consider only structured 3-D domains and decompose them with divisions/cuts parallel to the Cartesian Axes.

The parallelization of such simulations introduces additional performance penalties in the form of local and global synchronization among cooperating processes. Domain decomposition, the first step in parallel computing, partitions the largest shareable data structures into sub-domains and attempts to achieve perfect load balance with minimal need for communication. This chapter aims to introduce, develop and validate an alternate strategy to achieve optimal domain decomposition/partitioning for structured 3-D stencil-based PDE discretizations. This new strategy uses the minimization of cache-misses at the sub-domain level as the basis for obtaining optimal domain partitions/decompositions. We further, logically divide the sub-domain into three parts, namely, the *Independent Compute* (IC) - a part which does not require data from other processes for computation of a full iterative update, the *Dependent Planes* (DP) - a part which requires data from other processes for updating the solution, and the *Ghost Layer* (or *Halo Layer*) which acts as a buffer for the incoming data from neighbouring processes. Up to now research efforts to optimize spatial and temporal cache reuse for stencil-based PDE discretizations have considered sub-domain operations after the domain decomposition has been determined [6,12–14,126]. We derive a heuristic that minimizes cache-misses at the sub-domain level through a cache-directed analysis to predict families of high performance domain decompositions of structured 3-D grids. Our approach and strategy thus connects a true single core parameter (i.e. cache-misses) to a true multicore parameter (i.e. domain decomposition) - an aspect which to the best of our knowledge has no associated literature. The analysis is followed by appropriate experiments to demonstrate the efficacy of our high level model. The chapter concludes by emphasizing the need to re-examine the orthodox approach of domain decomposition for stencil-based PDE discretizations due to the tightly-coupled, evolving software and

hardware ecosystem of multicore processors.

4.2 Motivation and Contribution

Traditionally and universally domain decomposition or domain partitioning has been a function of minimizing communication volume only. Thus, the aim has only been to reduce the number of elements which are exchanged by multiple cores when subdividing a problem into sub-problems. This is achieved by using the *default* `MPI_Dims_create()` function in the *C* language or `mpi_dims_create()` in the *Fortran* language, as outlined in the MPI specification [48]. When considering structured 3-D domains, this approach results in cubic or nearly-cubic sub-domain shapes so as to minimize the surface area of the sub-domain. Thus, the approach minimizes the volume of communication elements which are needed by neighbouring MPI processes for updating the solution at the mesh points. Due to the increase in network capacity, reduction in transmission times, growth of Vectorization units requiring contiguous streams of data, and the very slow improvement in latency, the software and hardware ecosystem has changed since the original development of MPI. Packing and unpacking of data can contribute a high percentage of the total cost of transmitting data and thus needs to be examined in terms of cache-misses as the latter are the major factor in contributing to the overall computation time. We thus base our approach on quantifying cache-misses for various domain decompositions and selecting those that minimize the cache-misses. Our model, described later in this chapter, attempts to quantify this concept. The results of our experiments further strengthen our motivation and our efforts to continue looking beyond the orthodox approach of solely minimizing the communication volume. We make the following contributions in this chapter.

- An in-depth analysis and worst-case prediction of read/write cache-misses due to the local computations in the Independent Compute (IC) kernel and the Dependent Planes (DP), along with packing/unpacking cache-misses involved in the communication of data.

- Build a high level mathematical model using the *cache-line length* and the contiguity of data to quantify the cache-misses.

- Show that the inferences derived from the mathematical model are oblivious of the cache-line length and are based on the data layout only.

- Prediction of high performance families of virtual process topologies.

- To emphasize that a hand-coded optimization at sub-domain level can interfere with *compiler* optimizations.

- Predict and demonstrate that, given the same amount of data in an X/Y/Z-plane (Dependent Planes), communication of Z-planes is the most expensive.

- Examine the relationship, and build a bridge between, the most important *Serial Control Parameter* (SCP), i.e. cache-misses, and the first *Parallel Control Parameter* (PCP), Domain Decomposition.

4.3 The Problem

The introduction of parallel program design standards for implementation of *Application Programming Interfaces* (API), combined with advancements in the hardware of shared and distributed memory machines, has instigated researchers to make a variety of efforts to redesign, reimplement and optimize existing algorithms. Designing a parallel program consists of several steps. To take advantage of the several CPU cores available in a parallel computer, an existing problem must be partitioned and assigned to these cores, which then simultaneously execute instructions on the part of the problem assigned to them. While partitioning/decomposing, the focus can either remain on computations/functions or data [52]. Processes may communicate with a proper subset of processes (local communication) or with all other processes (global communication) for exchanging relevant data needed for the purpose of solving their sub-part of the complete problem [10]. Figure 4.1 shows the division of a *vertex-centered* $N_x \times N_y = 5 \times 5$ problem with *Dirichlet* boundaries (see Chapter 2) being represented by red balls. The number of internal mesh points at which the solution is to be computed is then 4×4. The entire domain is decomposed into four sub-domains, each having a local size of 4×4 including ghost/halo cells (2×2 excluding these). The ghost cells either represent the boundaries, if the process has no neighbouring process in a particular direction, or these ghost cells can act as buffers to store the incoming data coming from neighbouring processes. These *sub-domains* or *sub-grids* can be assigned to the cores that are available for computation in different ways. For example, if there are four available cores then a single sub-domain is assigned to each core and if there are only two cores then each core is assigned two sub-domains. In this work we follow the former strategy, i.e. we only assign a single sub-domain to each core. It can be seen from Figure 4.1 that the structure of the sub-domains is different from that of the domain. We elaborate and logically classify the different parts in the sub-domain at appropriate points as we proceed.

It is to be borne in mind that processors are unaware of the logical structure of the global problem and thus, the programmer's view of the problem can be totally different from the processor's view of it. For example, while a programmer thinks in terms of a problem as 1-D, 2-D or 3-D arrays, the physical layout of the data of an array in the memory of a processor is always linear. It is the programmer's responsibility to wisely choose an appropriate decomposition which maximizes the overall performance of the application.

As described in Chapter 3 two types of standard hardware architectures exist today to solve such partitioned problems in parallel. A shared memory machine (SMP) lets each process run-

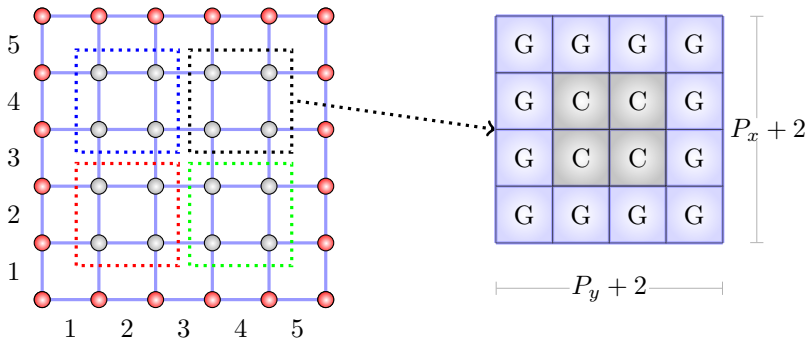
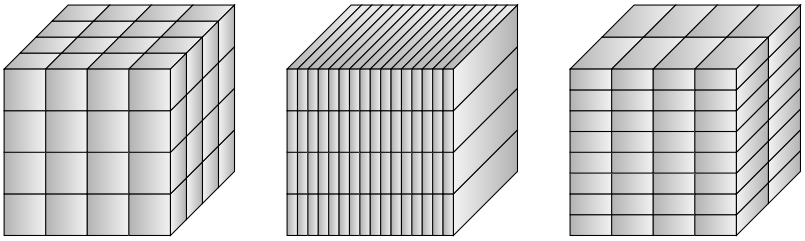


Figure 4.1: A Vertex Centered (VC) problem of size $N_x \times N_y = 5 \times 5$, having 4×4 internal mesh points is partitioned among 4 cores. The result is a $(P_x + 2) \times (P_y + 2) = (2 + 2) \times (2 + 2)$ sub-domain with 4 original 'C' cells and added ghost layer cells 'G'.

ning on any core access the complete memory in the system by using a global address space [61]. Processes in such machines communicate by writing to/reading from the shared memory. Orthogonally, a distributed architecture does not share memory between processors, and processes communicate by passing messages to each other. MPI (Message Passing Interface) [48] is the de-facto standard for programming distributed memory machines. Though MPI uses a message passing mechanism, it can also be used on shared memory machines i.e. the programming model need not match the underlying physical hardware. Hybrid architectures consisting of several shared memory nodes interconnected by a high speed network such as Infiniband [66] have become a norm.

The volume of an object in the physical world naturally maps to a 3-D data structure. When this 3-D data structure is divided into 3-D sub-domains and mapped to different processor cores, it imposes a geometrical arrangement of the processes as well. This geometrical arrangement of processes is termed a *Virtual Process Topology* [48] and the MPI standard specifies various *Cartesian Topology* functions for realizing such a topology. Functions such as `MPI_Dims_create()`, `MPI_Cart_create()` and `MPI_Cart_coords()` etc., help in specifying and creating an *n-dimensional* virtual process topology. The `MPI_Dims_create()` and the `MPI_Cart_create()` functions were explained in detail in Chapter 2. It is not necessary to use these functions for the programmer to visualize the topology and an n-dimensional topology of processes can be implemented by hand, but the use of such functions is recommended as they have been optimized by popular implementations of MPI such as OpenMPI [127] or MPICH [117].

For a given processor count, a spatial domain can be divided in several ways. Given 64 cores, a total of 28 virtual process topologies exist and the default MPI process topology returned by `MPI_Dims_create()` is $4 \times 4 \times 4$. Figure 4.2 shows 3 possible 3-D decompositions out of the 28



(a) $4 \times 4 \times 4$ (b) $4 \times 16 \times 1$ (c) $8 \times 4 \times 2$

Figure 4.2: Domain decompositions corresponding to three virtual process topologies

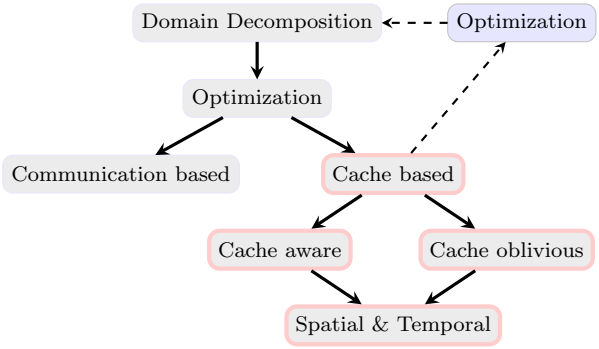


Figure 4.3: Traditional optimization (solid arrows), our approach (dashed + solid arrows)

possible topologies (decompositions) for 64 cores. Each 3-D sub-domain can be mapped to a unique CPU core and thus each process running on a CPU core has well-defined neighbours (i.e. those responsible for neighbouring sub-domains) whose MPI ranks can be uniquely determined.

The traditional criteria for deciding the domain decomposition consists of balancing the load on homogeneous processors and minimizing the volume of communicated data among them. Three levels, namely, local computations not requiring any communication, computations requiring data from neighbouring sub-domains (i.e. local communication) and global computations (i.e. requiring data from all processes) have been identified. Global communication generally has the largest effect on the performance of the parallel algorithm [10] and is to be avoided wherever possible.

There has been a stupendous increase in the computing power of processors/cores and capabilities of high-capacity interconnects. Performance optimizations can be done at several levels - beginning with domain decomposition at the macro-level and then optimizing the particular decomposition at the micro-level. With optimizations in the stacks of distributed programming paradigms, along with the advances in hardware, several optimizations in high performance parallel methods for implementing stencil codes have been explored. This idea is illustrated in Figure 4.3. These optimizations are majorly aimed at reducing the cache-misses [5] after a

domain decomposition has been carried out [6,12,14,16,18]. The process decomposition is generally aimed at reducing the total communication between processes. Thus, most applications choose this topology as the default for process decomposition. The work under investigation here attempts to predict the best family of topologies which automatically reduce the number of cache-misses in each sub-domain. Referring to Figure 4.3, our final objective is to optimize a sub-domain naturally using an efficient domain decomposition and further, encourage the use of sub-domain level optimizations.

4.3.1 Notation and Reference Figure

While simulating Finite Difference Methods [22] in 3-D, we represent the size of the input problem as $N_xN_yN_z$, where (N_i+1) is the number of mesh points in direction i and $i = x, y, z$. The number of internal points (i.e. unknowns in the terminology of Finite Difference Methods) is then $N_i - 1$. In the case of pure Dirichlet boundary conditions the outermost points in a 3-D domain form the boundary in our problem and have a prescribed value. Hence, for Dirichlet problems, we have a system of linear equations in $(N_x-1)(N_y-1)(N_z-1)$ unknowns which may be solved using an iterative scheme such as unweighted Jacobi, weighted Jacobi, Gauss-Seidel etc. Stated concisely, the above discussion formulates a *Boundary Value Problem* (BVP) [22,37] in a structured 3-D domain which is solved using a 7-pt stencil (say) in FDM to simulate a linear Elliptic PDE.

For parallel processing, these points (vertex unknowns) in each direction must be divided into sub-domains and mapped to individual processes running on independent cores (see Figure 4.1). Without any loss of generality, and to make the inferences and discussion simpler, we assume $N_x = N_y = N_z = N$. The number of processes or cores = P and any regular Cartesian domain decomposition must satisfy $D_xD_yD_z = P$, where D_i is the number of cuts/divisions in the i^{th} dimension for $i = x, y, z$. The number of mesh points (i.e. unknowns) assigned to each process is then $P_xP_yP_z$, where $P_i = \frac{N_i-1}{D_i}$ and $i = x, y, z$. Since the domain has been partitioned, sub-domains will require data from neighbouring sub-domains for stencil calculations. To store data from adjoining sub-domains, extra space is allocated to each sub-domain on each core. This data is typically called ghost data/ghost points/halo data [61]. Thus, the actual 3-D domain size allocated to each process = $(P_x + 2)(P_y + 2)(P_z + 2)$ due to ghost data/halo data, and we say that the ghost layer depth is one. We note that there will be processes which will have no neighbour in a particular direction. Such neighbours are called NULL processes and MPI has a constant named `MPI_PROC_NULL`[1] that may be used for representing them [48].

A process will need to pass between 0 to 6 planes of data, depending on the number of neighbour processes it has. Each sub-domain can be seen as being composed of three layers. The outermost layer stores the ghost data/ halo data and is not a part of the actual data that

[1]The value of this constant used by MPICH [117] is -1 whereas OpenMPI 1.6.5 [127] defines it as -2.

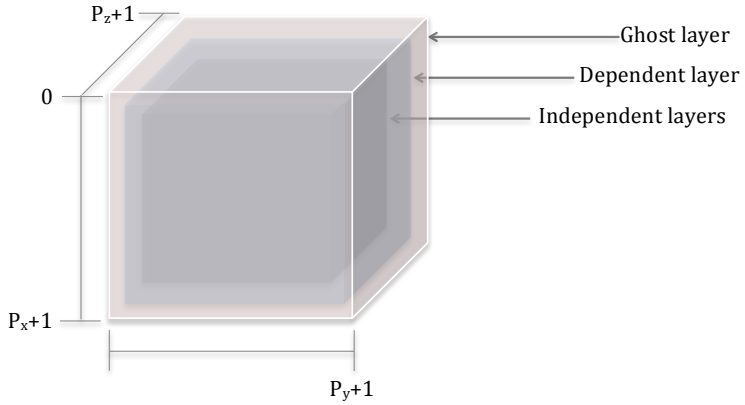


Figure 4.4: A 3-D sub-domain having an Independent Compute (IC) layer, Dependent Planes (DP) layer and Ghost/Halo layer, indexes of the sub-domain dimensions including the ghost layer are shown

the process contains but is necessary to store the data communicated by neighbouring processes. Hence, each process uniformly has 6 ghost layers to store data received from a maximum of 6 possible adjoining neighbours. There is no need for a ghost layer in a direction in which the neighbour is a NULL process i.e. no process. In such cases the ghost layer can act as a boundary layer and can be used to specify the boundary value (as in a Dirichlet Boundary Value Problem). The second layer is the Dependent layer - a layer which needs data from neighbouring processes to carry out stencil calculations. This has been appropriately named as a Dependent layer as it is dependent on neighbouring processes for stencil computations. We address the third layer as the Independent layer, and as the name suggests, it needs no data from neighbouring processes for computation of each iteration of the solution update algorithm. This layer also forms the computational kernel as it generally contains many more mesh points than the dependent layers. The various dimensions (indexes) can be seen in Figure 4.4 which also shows the three basic layers for a 3-D sub-domain: Independent layers which form the core computational kernel, Dependent layers which require data from other processes for updating elements in them and finally ghost layers to hold data from neighbouring processes.

A 7-point stencil in 3-D is illustrated in Figure 4.5. The central point is updated by the weighted average of six of its neighbours (two neighbours in each direction). These iterative solution algorithms then move to the next point, where the solution is updated using the same stencil, continuing until the whole domain under consideration is covered. The stencil in Figure 4.6 shows the same stencil along with directions and with the assumption that the central point has an index of (i, j, k). When considering the *Row-major* order (described later in this section), the data points at indexes $(i, j, k-1)$, (i, j, k) and $(i, j, k+1)$ are contiguous in memory. Similarly when considering a *Column-major* order (described later in this section), the data points at indexes $(i-1, j, k)$, (i, j, k) and $(i+1, j, k)$ are contiguous in memory.

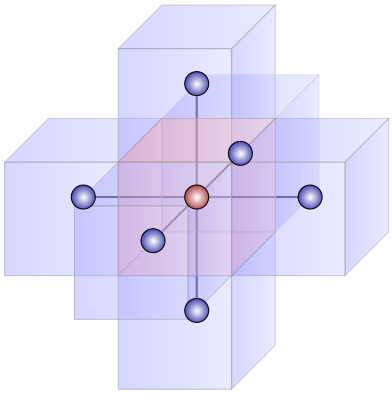


Figure 4.5: 7-pt Stencil for updating the central red point

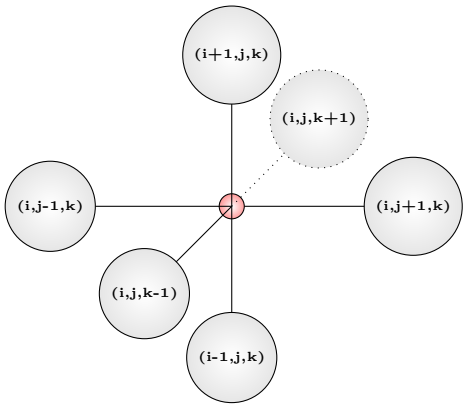
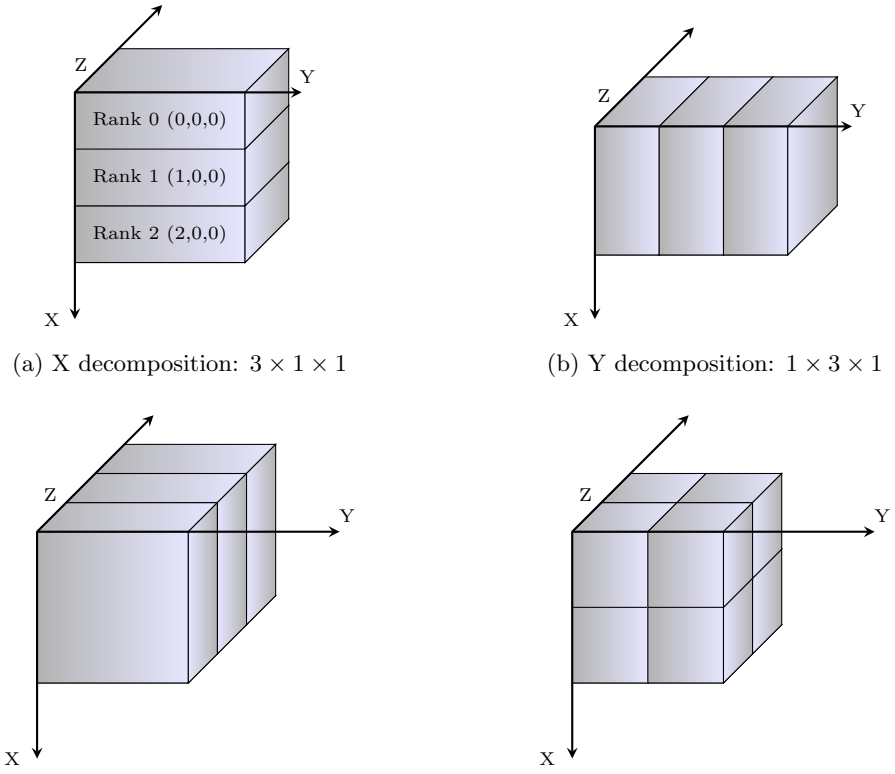


Figure 4.6: A 7-point stencil in 3-D. The central point is updated according to prescribed weights associated with, and values of the neighbouring points.

The total independent calculations done by each process at each solution iteration, i.e. the number of elements which do not depend on data from other processes, is: $(P_x - 2)(P_y - 2)(P_z - 2)$. The maximum total data contained in planes communicated by processes is $2P_yP_z$, $2P_xP_z$ or $2P_xP_y$ for the X, Y and Z planes, respectively. Please note that this is an upper bound on the data as there exist decompositions where data less than this upper bound can be sent depending on the number of neighbours which maybe NULL. The value $2[(D_x-1)(N_y-1)(N_z-1)+(D_y-1)(N_x-1)(N_z-1)+(D_z-1)(N_x-1)(N_y-1)]$ represents an upper bound on the total data elements communicated by all processes.

Figure 4.7 shows an example domain and the Reference axes with selected decompositions. The upper YZ plane is called X_UP and the lower YZ plane is called X_DOWN. The left XZ plane is called Y_LEFT and the right is called Y_RIGHT. The XY plane closer to the reader is called Z_TOWARDS_U and the plane farther away from the reader is called Z_AWAY_U. The



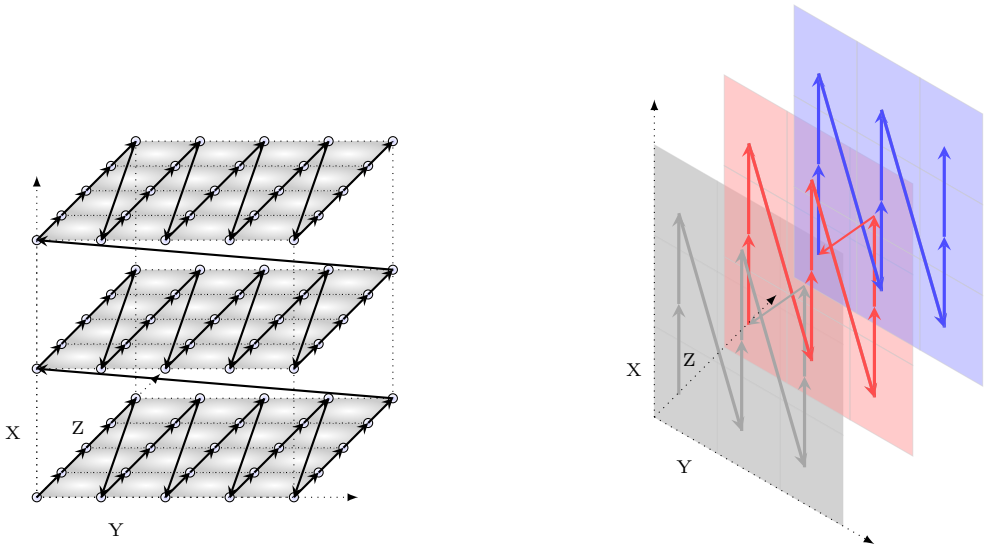
(a) X decomposition: $3 \times 1 \times 1$

(b) Y decomposition: $1 \times 3 \times 1$

(c) Z decomposition: $1 \times 1 \times 3$

(d) Decomposition: $2 \times 2 \times 2$

Figure 4.7: Process Grid Decomposition and Coordinate Axes (a) Shows process ranks in X decomposition with MPI process coordinates (b) Only Y direction is decomposed (c) Only Z direction is decomposed (d) General decomposition in all 3 directions



(a) 3-D data layout: Z direction - contiguous data

(b) Data layout where data is contiguous in the X-direction (Column-major order)

Figure 4.8: Row-major and Column-major data layout

coordinate axes shown in Figure 4.7 are in the direction of the coordinate axes assumed by the MPI function `MPI_Cart_coords()`. This function returns the process coordinates of processes in an *n-dimensional* space. Thus, for a topology of $2 \times 2 \times 2$ when $P = 8$, the ranks have the following process coordinates: Rank 0 (0,0,0), Rank 1 (0,0,1), Rank 2 (0,1,0), Rank 3 (0,1,1), Rank 4 (1,0,0), Rank 5 (1,0,1) Rank 6 (1,1,0), and Rank 7 (1,1,1). The fastest changing index is Z and the slowest changing index is X when looping through a 3-D MPI process decomposition - this also matches the Row-major data storage in C language when looping through a 3-D array.

Figure 4.8a shows the layout of data in a 3-D array. The data points are contiguous along the Z-axis and this is what constitutes a Row-major order layout. A language which supports such an order is the C language and we use the C language for all our implementations in this chapter. The contiguity of data points (drawn as circles) is shown by means of continuous black lines in Figure 4.8a. Figure 4.8b shows the Column-major order in which the fastest changing index is the X-index and this data layout is supported by a language such as Fortran. Although we illustrate both the data layouts here for completeness, we use the Row-major order in this chapter to quantify the cache-misses in the sub-domain. It can be noted that the final inferences derived from the model remain independent of the data-layout. The independence comes from the fact that the Z-direction in the Row-major order is analogous to the X-direction in the Column-major order and the X-direction in the former is equivalent to the Z-direction of the latter.

4.4 Creating a Model for Prediction

We focus on establishing a relation between minimizing cache-misses and domain decomposition by considering the internal layout of data of a sub-domain and the cache-line size. Our high level analysis allows us to ignore contention of shared resources, processor architecture, cache-line replacement policies - factors that contribute to cache-misses but are extremely difficult to quantify because of the multitude of interactions between contending processes on a multicore node. We always decompose along Cartesian Axes directions, i.e. perpendicular cuts along X, Y and Z dimensions (block partitions) and start the analysis by considering the planes consisting of near-to-boundary values (Dependent Planes - see Figure 4.4).

As discussed in Chapter 2, data from the main memory to the cache memory is transferred in terms of cache lines. A cache-line consist of multiple words. Multiple words from the main memory or low levels of cache are transferred for reasons of efficiency [5]. Thus, a cache-line or a cache block is the smallest unit of data that can be transferred from the main memory. For the purpose of the discussion that follows, we assume that the length of the cache-line in bytes is denoted by $\mathcal{L}$ or *cache_line_size*, the size of the data type is denoted by $\mathcal{D}$ (4 bytes for `float` or 8 bytes for `double`) and the number of words per cache-line is denoted by $\mathcal{W}$ i.e. $\frac{\mathcal{L}}{\mathcal{D}} = \mathcal{W}$. Our test platforms ARC2 and ARC3 (described in detail in Chapter 3), both have a cache-line length ($\mathcal{L}$) of 64 bytes. Thus, when considering a single precision (FP) `float` data type of $\mathcal{D} = 4$ bytes, $\mathcal{W} = \frac{\mathcal{L}}{\mathcal{D}} = 16$ floats can be transferred from the main memory to the cache memory in a single cache-line. When considering a `double` precision floating point data, 8 doubles can be contained in a cache-line. Although we include the cache-line length in our model (thus making it cache-aware), our inferences remain independent of the cache-line length (thus, cache-oblivious). This combination of cache-awareness and cache-obliviousness leads us to classify our model more precisely as a *Quasi-cache-aware* model.

Considering a Row-major order (see Figure 4.8a) and a 7-pt stencil, the minimum number of cache lines which can contain 3 contiguous data elements in the Z-direction, 2 non-contiguous data elements in the X-direction and 2 non-contiguous data elements in the Y-direction is 5. This is because, a single cache-line can contain 3 contiguous elements in the Z-direction, two cache lines are needed for the non-contiguous Y data neighbours and similarly two cache lines are needed for the X data neighbours. We will assume that the sub-domain is sufficiently large so that a single cache-line is unable to store all the data elements contained between two directly opposite ghost data points. At any point in time, while updating, we deal with 3 planes and assume at least 5 dedicated cache lines.

Require: Sub-domains with set Dirichlet boundary
  while Not converged **do**
    `MPI_Irecv` (ghost data)
    `MPI_Isend` (next-to-boundary data)
    Update (see Figure 4.10) interior independent values using 7-pt stencil
    `MPI_Wait` ( )
    Update next-to-boundary values using 7-pt stencil
    `MPI_Allreduce` (convergence test)
  end while

Figure 4.9: High level iterative parallel PDE solver, e.g. Jacobi

4.4.1 Parallel Numerical Solution of a Discretized PDE

The problem that we solve is abstractly illustrated in Figure 4.9. After dividing the data structures using domain decomposition, the approximate numerical solution at various mesh points is updated using an iterative method. As mentioned before, the sub-domain consists of an interior region which does not require data from other processes (Independent Compute - IC), a region called Dependent Planes (DP) that requires data from other processes and a ghost region, which is simply a buffer region to store incoming data. Since the IC can be updated independently of the data from other processes, computation can be overlapped with communication using the non-blocking point-to-point functions (`MPI_Isend() and MPI_Irecv()` shown in Figure 4.9) specified in MPI. After the data has been received (after `MPI_Wait()` in Figure 4.9) from other processes, the Dependent Planes are updated. It is important to note that separating the update of the Dependent Planes and Independent Compute introduces additional cache-misses as the data points in the DP are not accessed in continuity with the data points in the IC. At the same time, not overlapping the communication with computation generally incurs a performance penalty. With an increase in the number of cores in a node and size of the network, overlapping communication with computation has become the norm. After updating the IC and DP, the overall convergence is tested (e.g. through a global norm, requiring a global reduction operation). For the purpose of scaling studies we can fix the number of iterations and remove the convergence test (and associated global communication). It is to be noted that uniform single level meshes are generally not used to solve a PDE but they form the basis of multilevel methods such as Adaptive Mesh Refinement (see Chapter 5) and Multigrid (see Chapter 6).

For updating the solution at a mesh point we use the unweighted Jacobi point iterative algorithm as shown in Figure 4.10 (where `alpha`=$\frac{1}{6}$). The array elements in this code can directly be mapped to the 7-pt stencil. Figure 4.10 shows that two 3-D arrays are required for the Jacobi algorithm and the update of the solution at the data point (i, j, k) in the array `new` is done with the old values of the solution stored in the 3-D array `old`.

```
new[i][j][k]=alpha *
              (old[i-1][j][k]+old[i+1][j][k]+
               old[i][j-1][k]+old[i][j+1][k]+
               old[i][j][k-1]+old[i][j][k+1]);
```

Figure 4.10: Unweighted Jacobi iteration kernel, `alpha`=constant, `new` and `old` are 3-D data arrays

Table 4.1: Model Assumptions: Logically classified assumptions in deriving the model

Logical Class	**Assumption in Derivation**
PDE	Elliptic, second order and linear
Boundaries	Dirichlet
Domain	Cubic
Mesh	3-D structured mesh, $N_x = N_y = N_z$
Decompositions	Parallel to Axes
Discretization	Finite Difference
Stencil	7-pt
Iterative Method	Unweighted Jacobi
Data Layout	Row-major
Data Type	Single Precision Floating Point (FP)
Sub-domain	one-per-core, one element deep ghost layer
MPI process	one-per-core, no threads
Computation and Communication	Overlapped
Cache-Size	Problem size $\gg$ any level of cache
Cache Hierarchy	All levels merged into a single cache
Prefetching	Ignore
Temporal Locality	No
Spatial Locality	Yes

4.4.2 Reiterating Assumptions

Before we proceed to deriving the high level mathematical model quantifying the cache-misses for the sub-domains and extracting inferences from it, we consolidate the assumptions and categorize them logically as shown in Table 4.1. We again revisit these assumptions towards the end of the current chapter to expand upon the generality of the model. These assumptions help the model to remain high level and abstract but at the same time provide sufficient insight to appreciate the complex relationship between cache-misses and domain decomposition.

4.4.3 Dependent Planes

In this section we model the approximate cache-misses using the *unweighted Jacobi algorithm* and the 7-pt stencil for the Dependent Planes (DP). As mentioned earlier, there are three pairs of planes, namely, the X-planes (X_UP and X_DOWN), the Y-planes (Y_LEFT and Y_RIGHT) and the Z-planes (Z_AWAY_U and Z_TOWARDS_U). Since the two types of X, Y and Z planes are symmetrical, we do not separately calculate the cache-misses for each of them but rather treat them as a single X, Y or Z-plane. As we assume a Row-major order, the Z-plane is perpendicular to the direction of data contiguity while the other two planes are parallel to this direction. Thus, qualitatively we expect the number of cache-misses in the Z-plane to be higher as the data points comprising it are not contiguous in the memory. Again, qualitatively we expect the X-plane and the Y-plane cache-misses to bear a symmetrical expression as both the planes have contiguous data along the Z-direction. Equipped with our assumptions and a qualitative idea, the sections below describe in detail the derivation of the cache-misses for the Dependent Planes.

4.4.3.1 Z-Plane

As mentioned above, the Z-plane is the plane which is perpendicular to the direction in which data is contiguous. This plane has the greatest effect on the running time as no dimension has contiguous data here. In a 3-D domain decomposition and using a 7-pt stencil, 2-D layers of data must be passed to the neighbouring processes. There are three costs associated with the planes.

1. *Packing cost*: The data from the Dependent Planes is packed in the sending process explicitly by the user or implicitly by the MPI implementation. When using an explicit one dimensional application level buffer, the data from the specific Dependent Plane in the 3-D array must be read and copied to the 1-D buffer array. This leads to read cache-misses while reading the 3-D data structure and write cache-misses when writing into the one dimensional application buffer. The other method is to define an MPI Datatype, such as `MPI_Type_subarray()`, and let the MPI implementation do the packing implicitly. This again incurs cache-misses when the data is transferred from the 3-D application array to the MPI buffer. We choose to ignore the cost of writing into the MPI buffer or the user defined one dimensional array and concentrate only on the cache-misses while reading the 3-D user array. Ignoring the cost of cache-misses while copying the application buffer into the MPI buffer can be justified by highlighting that when using the *Rendezvous* protocol (see Appendix A), the application buffer can be directly manipulated for communication purposes by the MPI implementation.

2. *Unpacking cost*: At the receiving end, the neighbouring process explicitly unpacks the data from the MPI buffer to the 3-D application buffer or the MPI implementation implicitly unpacks it at the address of the specified location. This writing of data into the 3-D user

array is accompanied by cache-misses. There may also cache-misses when data is read from the MPI buffer but we choose to ignore these cache-misses and concentrate only on the former (see Appendix A).

3. *Update cost*: Finally, we must update the value of the unknowns in the Dependent Planes, which in turn depends on the data stored in the ghost layers as received from the neighbouring processes. This cost is expected to be much more than the cost of packing and unpacking as not only the mesh point but also its neighbours are accessed.

As mentioned above, we choose to ignore the write cache-misses when non-contiguous data in the planes is being written to a contiguous network buffer by the MPI implementation at the sender side. Similarly, we do not incorporate the read cache-misses when a contiguous network buffer is being read by the MPI implementation to unpack its contents into the non-contiguous application buffer at the receiver side. We extend this discussion with the aim to support our choice of not taking into account these cache-misses and enumerate our reasons as follows:

1. Our model is a high level model and we avoid low level implementation details.

2. The contiguous network buffer offers high spatial locality, both when writing into (sender side) or reading from (receiver side) it. Thus, the contiguous access, in practice, is the ideal scenario for minimizing cache-misses. Since we expect negligible cache-misses due to a linear access, we do not incorporate these cache-misses into our model.

3. In principle, an MPI implementation can transfer the non-contiguous data to the receiver without copying it to an intermediate contiguous buffer [68]. Thus, there exists a possibility that the non-contiguous data present in the Dependent Planes can be directly communicated to the receiver by the MPI implementation if the underlying communication mechanism supports it [69].

4. The direct transfer of non-contiguous data performs well when there are dense blocks of contiguous data [70]. The sending of data in the X and Y plane perfectly aligns with this case as they contain P_z contiguous data-points (discussed in sections that follow). The problem lies in the Z-plane where the contiguous blocks consist of only a single data point (hence, extremely sparse). Thus, for the Z-plane, it make sense to accumulate the data into a contiguous buffer before sending it to the destination.

5. The `vader` BTL (Byte Transport Layer) in OpenMPI (versions 1.7 and later) supports direct loads/stores in the address space of the process from/to other processes on the same node. Thus, instead of a copy-in/copy-out mechanism, it follows a zero-copy mechanism. A zero-copy mechanism still involves a single copy and should be interpreted to mean that there are no intermediate copies involved. The `vader` BTL can be configured with `xpmem`, `cma` or `knem` kernel modules to achieve this zero-copy mechanism. `xpmem` is a Linux kernel module that allows processes to export memory regions and other on-node

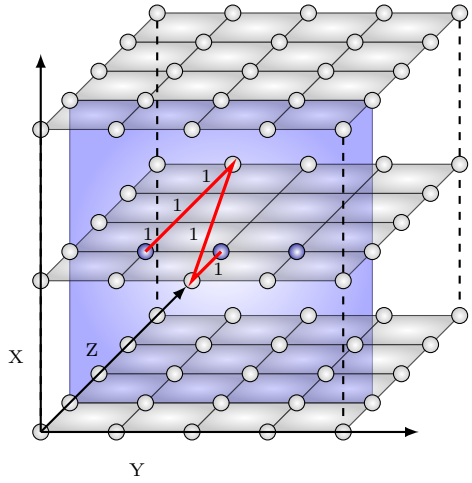


Figure 4.11: Dependent Z_TOWARDS_U (blue shaded vertical rectangle), adjacent points distance (thick solid red line $\approx P_z$) and boundary (unshaded circular points).

processes to attach to this memory region to perform direct loads/stores. `cma` stands for Cross-Memory Attach and is a Linux kernel such as `knem` that requires calls into the kernel to transfer data between intra-node processes [71].

In light of the discussion above, we choose to ignore the costs associated with the writing and reading of the contiguous network buffer managed by the MPI implementation. We, however, acknowledge the need for modelling such costs when moving towards a low level model incorporating architectural parameters and implementation details. We further believe that incorporating these costs in our high level model will not alter the formulation methodology or the inferences from the model.

Using a 1-element deep ghost layer, 2-D data from the Z-plane is packed implicitly (using `MPI_Type_subarray()` in our implementation) in the sending process and sent to the receiver. It may be noted that, as opposed to the data being packed explicitly by the user, implicit packing by MPI exposes greater parallelism. This parallelism becomes available in the application as now the computations can be overlapped with packing/unpacking as well, in addition to the transmission of data on the network. As mentioned above, while packing, read-misses (reading from user array) become significant and while unpacking, write-misses (writing to the user array) become significant. Both read cache-misses (while reading the `old` 3-D array, see Figure 4.10) and write cache-misses (while writing into the `new` 3-D array, see Figure 4.10) are significant when updating an element using its neighbouring elements.

Figure 4.11 shows the update of a Z-plane. The near-to-boundary points (in blue) have a minimum distance of P_z between them and hence do not represent contiguous data. When a

data point is updated, the cache logic tries to exploit spatial locality. With reference to updating the mesh points on the Z-plane, the greater the value of P_z, and smaller the length of the cache-line, the lesser the probability that the next needed element will be found in the cache. Assuming $P_z + 2 > \frac{cache\_line\_size(64)}{sizeof(FP)}$ for large problem sizes and only a single line is fetched upon a cache-miss, there is a cache miss for a write or read on every element of the Z-plane. Hence, the probability of a write-miss/read-miss is $\frac{P_x P_y}{P_x P_y} = 1$. Although our model does not take into account the prefetch (see Table 4.1), as long as the number of prefetched elements remain less than $P_z + 2$, there is a cache-miss on every read and write of a mesh point on the Z-plane.

For lending completeness to the discussion, we present an example to illustrate what is meant by a small scale problem but discard its presence in future discussions. As an example suppose the input problem size is $N_x N_y N_z = 161 \times 161 \times 161$ and the total number of cores is $P = 16$. If we assume an MPI Cartesian topology of $D_x D_y D_z = 1 \times 1 \times 16$, then $P_z = \frac{N_z - 1}{D_z} = \frac{161-1}{16} = 10$. Clearly, $P_z + 2 = 12 < 16$ and hence there is a probability that more than one data element in the Z-plane will be contained in a single cache-line as the length of the cache-line is 16 single precision floating point elements. Thus, the probability of a cache-miss even without prefetch is less than one when accessing adjacent data points in the Z-plane. We, however avoid such cases and for all practical purposes assume a large sub-domain size i.e. $P_z + 2 > \frac{cache\_line\_size}{sizeof(FP)}$. Table 4.2 shows various relevant parameters for Z-planes. The maximum as well as the minimum distance between two adjacent points on the Z-plane is $\approx P_z$ (we can ignore the two ghost points at the two boundaries of the sub-domain if $P_z >> 2$). As explained previously, the total read-misses/write-misses in packing/unpacking the Z-plane is $P_x P_y$ for a large P_z. Assuming a sufficiently large P_z again, there are 2 read-misses in accessing each of X and Y direction mesh points and 1 read-miss in the Z direction while updating a single mesh point of the Z-plane. Hence, there is a total of 5 cache read-misses in updating one element, making it a total of $5P_x P_y$ misses for the entire Z-plane (see Table 4.2). It is to be noted that the value of parameter $\mathcal{W}$ (words per cache-line) in Table 4.2 is $\frac{\mathcal{L}}{\mathcal{D}} = \frac{64}{sizeof(FP)} = \frac{64}{4} = 16$ as we consider only single precision floating point data in the current chapter.

4.4.3.2 X-Plane

The X-plane is the plane which lies at the top and bottom of the sub-domain. Both X_UP and X_DOWN have contiguous data in the Z direction (blue points in Figure 4.12). Irrespective of the value of P_z, the gap between the last element updated in the Z direction and the first next element on the X-plane is always two (two ghost data points). The various parameters for X-planes are shown in Table 4.3. It can be noted that while writing only one value is being accessed (see LHS of Figure 4.10) and hence the X-plane is being accessed in a linear manner. The same does not hold while reading as the 7-pt stencil accesses immediate data neighbours in all three directions.

Table 4.2: Z-Plane: Relevant parameters for Z-plane showing total size, distance between two adjacent elements, cache-misses in packing (reading)/unpacking (writing) and updating an element amongst others.

Description	**Value**
Total elements	P_xP_y
2 element gap	$P_z + 2 \approx P_z$, if $P_z >> 2$
Probability cache write-miss	1, if $P_z + 2 > \mathcal{W}$
Total cache write-misses (update/unpack)	$\approx P_xP_y$, if $P_z + 2 > \mathcal{W}$
Probability cache read-miss	1, if $P_z + 2 > \mathcal{W}$
Total update cache read-misses	$5P_xP_y$ if $P_z + 2 > \mathcal{W}$

Table 4.3: X-Plane: Relevant parameters for the X-plane showing total size, the maximum gap between two adjacent elements, read/write cache-misses in packing/unpacking and update

Description	**Value**
Total elements	P_yP_z
Max. 2 element gap	2
Probability of cache write-miss	$1/\mathcal{W}$
Total cache write-misses (unpack/update)	$P_yP_z/\mathcal{W}$
Probability of a cache read-miss	$1/\mathcal{W}$
Total update cache read-misses	$\frac{5}{\mathcal{W}}P_yP_z$

All updates proceed in the Z direction where data is contiguous and hence after a cache-write miss, data would be fetched into the cache according to the cache-line size. Thus, there is a cache write-miss after every $\mathcal{W}$ elements ($=\frac{\mathcal{L}}{\mathcal{D}}$) as we ignore the Prefetch. Further, there are 5 cache read-misses every $\mathcal{W}^{th}$ element, making a total of $\frac{5}{\mathcal{W}}P_yP_z$ cache read-misses for the entire plane in the worst case (see Table 4.3). For our model/implementation in the current chapter, $\mathcal{W} = 16$ for $\mathcal{L} = 64$ bytes and single precision floating point data ($sizeof(FP) = 4$ bytes).

4.4.3.3 Y-Plane

The planes Y_LEFT and Y_RIGHT have contiguous data in the Z direction but not in the X direction. The gap between the last updated element in a row (last element in the i^{th} line in the Z-direction) and the first element in the next row (first element in $(i+1)^{th}$) line) is $(P_z + 2)(P_y + 1) + 2$. This quantity represents the maximal gap between any two adjacent elements in the Y-plane. Table 4.4 shows relevant parameters for the Y-plane. Data here is contiguous in the Z-direction and hence there is a cache write-miss every $\mathcal{W}$ elements ($= \frac{cache\_line\_size}{sizeof(FP)}$ in the worst case), making the probability of a cache write-miss $\frac{1}{\mathcal{W}}$. The total cache write-misses are then $\frac{1}{\mathcal{W}}P_xP_z$. But unlike the constant maximum distance of 2 elements in updating the X-plane, the distance here is variable and depends on the Z and Y direction.

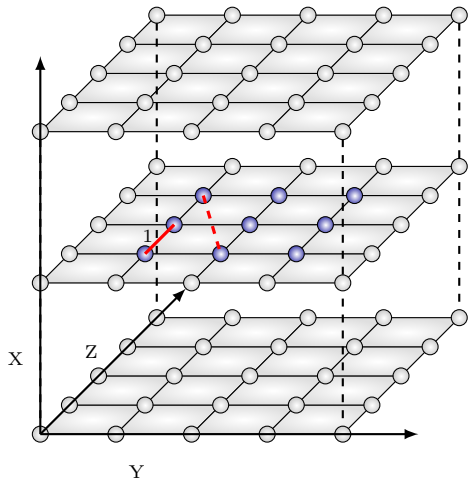


Figure 4.12: X-plane update: Data elements are contiguous (solid thick red line) except at boundary (dashed thick red line)

Table 4.4: Y-Plane: Relevant parameters for the Y-plane including its size, maximum gap between two adjacent elements, read/write misses in packing/update.

Description	**Value**
Total elements	$P_x P_z$
Max. 2 element gap	$(P_z + 2)(P_y + 1) + 2.$
Probability cache write-miss	$\frac{1}{\mathcal{W}}$
Total cache write-misses (unpack/update)	$(1/\mathcal{W})P_x P_z$
Probability cache read-miss	$\frac{1}{\mathcal{W}}$
Total update cache read-misses	$\frac{5}{\mathcal{W}}P_x P_z$

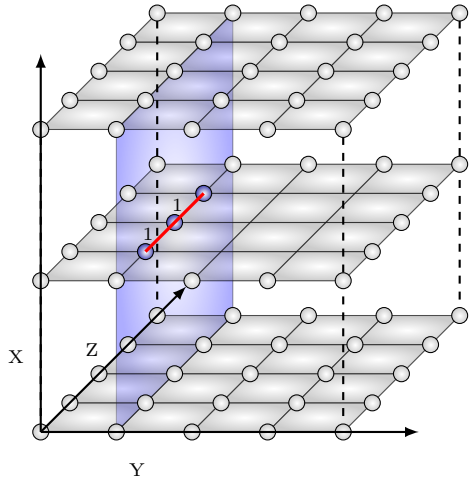


Figure 4.13: Dependent Y_LEFT plane (blue vertical shaded rectangle) and distance between two adjacent points (solid red thick line).

The higher the value of $(P_z+2)(P_y+1)+2$, the lower the probability that the fetched data will be available in cache while updating a Y-plane. Aside from the maximal gap between adjacent elements, the Y-plane is very similar to the X-plane as indicated by the similarity of Tables 4.3 and 4.4.

Figure 4.13 shows the Y_LEFT plane and a contiguous stream in the Z direction. We move along the contiguous Z direction to update the Y plane and hence the data for the next element is available if the gap between the current and next element is less than the size of the cache-line. Hence, the total cache read-misses is $\frac{5}{\mathcal{W}}P_xP_z$ (2 for X neighbours, 2 for Y neighbours and 1 for Z neighbours). If there is Prefetching involved then the packing of the same sized X-plane should perform better than a same sized Y-plane as there is a maximum constant gap of 2 between any two updated elements and there is higher probability that Prefetching will cover that gap of 2 for the X-plane instead of a gap of $(P_z+2)(P_y+1)+2$ for the Y-plane. However, as mentioned in the previous sections, we do not incorporate Prefetching in our model.

4.4.4 Independent Computation

Irrespective of the dimensions, the Independent Compute kernel (IC) has a maximum gap of 4 elements between the last updated element and the next element to be updated. This gap of 4 elements is made up of 2 data elements of the Dependent Planes and 2 data elements of the ghost layer (assuming a 1-element deep ghost layer - see Table 4.1). As P_z decreases and P_x and P_y increase, the total gap/unwanted elements will increase. A write-miss in the IC is expected only after approximately $\mathcal{W}$ elements (assuming no prefetch). Hence, the probability of a write-miss is approximately $\frac{1}{\mathcal{W}}$, which generates a total of $\frac{1}{\mathcal{W}}(P_x-2)(P_y-2)(P_z-2)$ cache

Table 4.5: Independent Compute (IC): Relevant parameters including the size, maximum gap between two elements, and read/write cache-misses in update.

Description	**Value**
Computational elements	$(P_x-2)(P_y-2)(P_z-2)$
Max. 2 element gap	4
Probability cache write-miss	$1/\mathcal{W}$
Total cache write-misses	$\frac{1}{\mathcal{W}}(P_x-2)(P_y-2)(P_z-2)$
Probability cache read-miss	$1/\mathcal{W}$
Total update cache read-misses	$\frac{5}{\mathcal{W}}(P_x-2)(P_y-2)(P_z-2)$

write misses. Since this is the same for all topologies, a uniform cache-miss rate is expected irrespective of the size of the cubic sub-domain but the total number of cache-misses is clearly a function of the size of the sub-domain. The case for cache read-misses is similar.

Table 4.5 shows relevant parameters for the IC kernel. Note that the IC kernel is the part of the sub-domain where computation can be overlapped with communication (see Figure 4.9) using the non-blocking communication routines in MPI. When the data is being packed by the communication progress engine, the cache is being used for two purposes: to bring in data for independent computations, and to bring in data from the dependent planes which are being packed if the neighbour $\neq$ `MPI_PROC_NULL`. Since the cache is now being used for both the purposes mentioned above, the cache miss rate is likely to go up because of cache pollution[2]. Similar is the case of unpacking of data if the MPI implementation decides to unpack it before `MPI_Wait()` is executed. If the data is unpacked at the point of executing the *wait* call, we can be sure that the IC core has already been updated. To discuss the communication progress engine of MPI [48] is beyond the scope of the current work and further, the progress engine is dependent on the MPI implementation itself.

4.4.5 Packing, Unpacking and Updating

In general, the number of cache write-misses for unpacking will be the same as cache read-misses while packing data. While updating data, the number of cache write-misses will be different from cache read-misses because of the pattern of data accesses in the 7-point stencil. Table 4.6 shows the total number of cache-misses in the worst case in terms of sub-domain dimensions P_x, P_y and P_z as predicted by our model when a plane is packed, unpacked and updated.

4.4.6 Minimization of Cache-Misses

We proceed to minimize the cache-misses that we derived in the previous sections. The total cache-misses for the three Dependent Planes using Table 4.6 and substituting $\mathcal{W} = 16$ (=

[2]Prefetched cache lines of the Dependent Planes may evict the cache lines from the Independent Compute in a real execution of the program.

Table 4.6: Plane Cache-Misses: read/write cache-misses in packing/unpacking/updating X, Y and Z-planes

Plane	**Pack read-misses**	**Unpack write-misses**	**Update read-misses**	**Update write-misses**	**Total**
Z-plane	P_xP_y	P_xP_y	$5P_xP_y$	P_xP_y	$8P_xP_y$
X-plane	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{5P_yP_z}{\mathcal{W}}$	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{8P_yP_z}{\mathcal{W}}$
Y-plane	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{5P_xP_z}{\mathcal{W}}$	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{8P_xP_z}{\mathcal{W}}$

$\frac{\mathcal{L}}{sizeof(FP)} = \frac{64}{4}$) can be written as:

$$S = 8P_xP_y + \frac{1}{2}P_xP_z + \frac{1}{2}P_yP_z = \alpha P_xP_y + \beta P_z(P_x + P_y), \tag{4.1}$$

where α and β are dependent on the length of the architecture-specific cache-line (here $\alpha = 8$, $\beta = \frac{1}{2}$). Our goal is to minimize this expression to obtain the least value of S. By equating $\frac{\partial S}{\partial P_x} = 0$ and $\frac{\partial S}{\partial P_y} = 0$, we obtain $P_x = P_y$ but this does not yield any relation to P_z. Since N is constant, the values of P_x, P_y and P_z are dependent on the values of D_x, D_y and D_z such that $D_xD_yD_z = P$, where P is the number of processes or cores. Clearly, we can find all possible combinations of D_x, D_y and D_z and hence in turn find all possible values of P_x, P_y and P_z by noting that $P_i = \frac{N-1}{D_i}$ for $i = x, y, z$. Thus, we can substitute these values of P_x, P_y and P_z into S to find all possible values of $\alpha P_xP_y + \beta P_z(P_x + P_y)$. We observe that the minimum value of S is obtained when $P_x = P_y$ and $P_z = N - 1$. Since $P_x = \frac{N-1}{D_x} = P_y = \frac{N-1}{D_y}$, we obtain $D_x = D_y$. The second condition $P_z = N - 1$ implies that $D_z = 1$ as $P_z = \frac{N-1}{D_z}$. Thus, our solution implies that for S to be minimum, we need $D_x = D_y$ and $D_z = 1$.

In the worst case when all six planes are sent, the volume of data is given by:

$$V = 2(P_xP_y + P_yP_z + P_zP_x). \tag{4.2}$$

Minimizing V in Equation (4.2) by manipulating $\frac{\partial V}{\partial P_x}$, $\frac{\partial V}{\partial P_y}$ and $\frac{\partial V}{\partial P_z}$, we obtain $P_x = P_y = P_z$. The intersection of conditions for minimization of the sum of communicated elements and minimization of cache-misses leads to a common condition $P_x = P_y$. This implies that $D_x = D_y$ when $N_x = N_y = N_z = N$. In the more general case where $N_x \neq N_y \neq N_z$, the ratio $\frac{(N_x-1)}{D_x} = \frac{(N_y-1)}{D_y}$ must be maintained.

As the problem size increases, the inner IC kernel increases faster than the surface area of the planes. For example, when the problem size increases 8 times, the independent computational

domain increases 8 times as compared to a 4 times increase in the surface area. Our derivation in the current Section is based on the assumption that the cache-misses due to the IC kernel should not be much larger than the sum total of cache-misses incurred by the planes. If this case is violated i.e. $\tilde{S} = \frac{(5+1)}{16}(P_x - 2)(P_y - 2)(P_z - 2) >> S$, then the optimal topology moves towards the topology given by the default `MPI_Dims_create()`. This does not mean that the topology determining optimal domain decomposition is always the one returned by `MPI_Dims_create()` but rather that the optimal topology will be found at a higher $D_z \leq D_{sz}$, where D_{sz} is the Z-dimension returned by the default `MPI_Dims_create()`. Since minimizing $\tilde{S}$ yields $D_x = D_y = D_z$ and minimizing S gives $D_x = D_y$, $D_z = 1$, thus $1 \leq D_{z\_optimal} \leq D_{sz}$. In other words, `MPI_Dims_create()` returns the upper limit of the search space of the highest performing topologies. Thus, in summary, whereas the orthodox approach of optimizing the domain decomposition by minimizing the communication volume suggests cubic sub-domains, our approach of minimizing the sub-domain level cache-misses suggests partitions which are close to 2-D partitions.

4.4.7 Interpreting the Model

Our model first quantifies the cache-misses for the Dependent Planes (DP) by taking into account only the cache-line size and ignoring Prefetching. This quantification (see Table 4.6) implies that for planes of equal sizes, the Z-plane incurs the maximum cache-misses. Thus, in the orthodox method of minimizing the communication volume, since the sub-domain is cubic (or close to cubic), the Z-plane incurs a much higher cost (cache-misses) while packing, unpacking and in updating of the solution as compared to the X/Y-planes. Our model which points to the topologies that minimize cache-misses at the sub-domain level suggests that the size of the Z-plane should be reduced. As a natural consequence of reducing the size of the Z-plane, the packing/unpacking and update times of the Z-plane reduces. This also means that the size of the X-plane and Y-plane increases but since there exists a contiguous data stream in the Z-direction for both these planes, the packing/unpacking efficiency is expected to be much higher than a similar sized Z-plane.

Laying emphasis on the cache-misses during sending and receiving of data logically divides the communication into two parts: a *latency* governed packing/unpacking of data and a *bandwidth* governed network transmission. Since the network bandwidth is improving at a much faster rate than the latency, our model also serves as an aid to identify the bottleneck in the transmission of data [128]. Latency is defined as the time that elapses between the issuance of a request for a memory value by the processor and the time that the first byte is transferred to it. Bandwidth is the speed at which the second byte and all the remaining bytes of data are transferred [129]. The processor cycles to access a lower level of memory level are always more than a memory that is closer to the processor. The more the cache-misses, the greater the probability that a higher number of memory accesses are generated (assuming lower levels

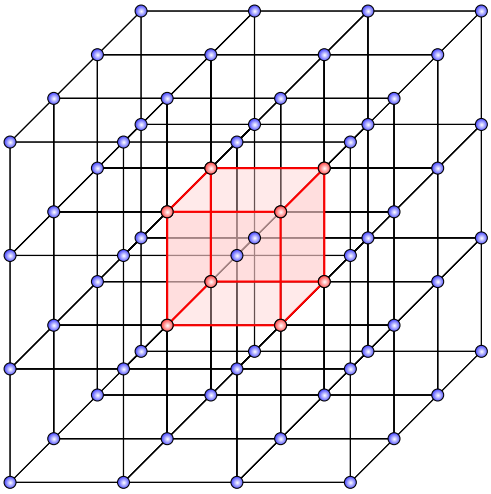


Figure 4.14: Test problem illustration, Vertex centered, domain $N_x \times N_y \times N_z = 3 \times 3 \times 3$, blue balls show Dirichlet boundaries and red balls show the unknowns

of cache also generate a miss). Thus, at a finer level it would not be incorrect to say that the packing/unpacking/update in the Z-plane is latency-bound whereas it is bandwidth-bound in case of X/Y planes. However, it must be noted that both latency and memory bandwidth play an important role in governing the transmission costs of the Dependent Planes. Specifically for the Z-plane, each data element fetched into the cache is part of a different cache-line and thus results in filling the cache with elements which are not used in packing/unpacking/updating. This can result in an increase in capacity misses and unnecessary eviction of useful data.

4.5 Test Problem

We implement a finite difference approximation to the Laplace equation $\nabla^2 u = 0$ in 3-D where $u = u(x, y, z)$. This equation as described in Chapter 2 is an Elliptic, linear, homogeneous PDE of order two, and we solve using Dirichlet boundary conditions for boundary $\partial\Omega$ $(u = 1)$. Implicit equations in $(N_x - 1)(N_y - 1)(N_z - 1)$ unknowns are created using a finite difference 7-point stencil on a unit cube $\Omega = (0, 1)^3$. Without loss of generality, and for simplicity, the simulation assumes that $(N_i - 1)\%D_i = 0$ for $i = x, y, z$. When $(N_i - 1)\%D_i \neq 0$, it produces a load imbalance and complicates an unbiased study of the effect of domain decompositions. Further, we always use an unweighted Jacobi computational kernel (see Figure 4.10) in 3-D for evaluation and discussions.

Figure 4.14 shows a vertex-centered $N_x \times N_y \times N_z = 3 \times 3 \times 3$ domain. There are two types of vertices: the vertices at the boundary (blue balls in Figure 4.14) represent the Dirichlet boundaries and the internal vertices (red balls in Figure 4.14) represent the unknown variables

where the solution is to be found. The total number of unknowns or *degrees of freedom* (dof) in this case are $(N_x - 1) \times (N_y - 1) \times (N_z - 1) = 2 \times 2 \times 2 = 8$. It is to be noted that in this particular illustration if the problem is solved on a single core then the Independent Compute (IC) zone is empty i.e., there are only Dependent Planes (DP). The reason for this is that since the sub-domain is the same as the domain here because of a single core, the sub-domain dimensions P_x, P_y and P_z are equal to $N_x - 1, N_y - 1$ and $N_z - 1$, respectively. Since the IC has a volume equal to $(P_x - 2)(P_y - 2)(P_z - 2) = 0$, the IC is completely empty. We consistently follow this convention everywhere but our main interest lies in much larger problems where the sub-domain dimensions are such that both the IC and DP are non-empty. It is to be noted that after every application of the iterative algorithm to update the solution, the Dirichlet boundaries are not updated in the vertex centered scheme as the physical boundaries coincide with the position of the grid boundaries. This is different from a *cell-centered* scheme which requires the Dirichlet boundaries to be updated after every iteration as they do not coincide with the physical boundaries.

4.6 Experimental Results

We carry out various experiments to test the validity and efficacy of our model derived in the previous sections. We only make use of pure MPI and assign a single sub-domain to a single core. When using a node, unless otherwise stated, all the cores in the node actively participate in the simulation. To make sure that all topologies or decompositions run on the same set of cores, we test all the topologies within the same execution of the program and thus once the batch job is assigned resources, all topologies are executed in the same run of the program. It is difficult to predict and quantify the effect of process placement as we have no control over the resources granted by the *SGE (Son of Grid Engine)* scheduler. Unless specifically mentioned, by default all our experiments in this chapter are carried out using the ARC2 (Advanced Research Computing 2) facility described in detail in Chapter 3.

The metric that we use to compare the relative performance of topologies is the time for execution and we explain its use in the next section. We logically proceed by testing our hypothesis on a single (16 core) node and then advancing to multiple nodes. In practice, stencil codes are heavily optimized using spatial and temporal methods for reducing cache-misses. Our aim in the current work is not to research tiling optimizations. We abstain from any discussion on temporal optimizations but devote appropriate space to spatial optimizations as the latter category is more commonly utilized in stencil-based codes. Our experiments then proceed to study *Strong Scaling* and testing the communication efficiency with *Weak Scaling*. At appropriate milestones, we quantify the cache-misses to test the inferences from our model. Towards the end of the experimental section, we deviate from utilizing all the cores of a node and weigh the increasing communication costs against the increasing computation performance

due to reduced contention of the shared cache memory. We then present a few representative results with a 19-pt stencil to test the applicability of our model but abstain from using a 27-pt stencil that is presented in Chapter 6.

4.6.1 Performance Metric

Throughout the thesis our performance metric remains the time for execution. Since we compare the performance of various topologies, it is appropriate to say that the time to execution is relative. Thus, for two topologies/decompositions/sub-domain shapes T_1 and T_2, if the time taken to execute T_1 i.e. $t(T_1)$ is less than that to execute T_2 i.e., $t(T_2)$ on a given experimental test-bed $\mathcal{E}$, then we say T_1 outperforms T_2. We do not directly compare the performance of some topology T on two different test-beds $\mathcal{E}$ and $\tilde{\mathcal{E}}$. For all experiments that we conduct, we can divide the time into two distinct parts: *Set-up* and *Solve*. For most experiments, it is the latter which is significant and our timing measurements thus focus on the Solve phase. Two extremely significant performance measurement methods for parallel computing are Weak Scaling and Strong Scaling. Weak Scaling measures the performance when both the problem size and number of cores increase but the problem size per-core remains constant. The main purpose of Weak Scaling is to measure the efficiency of communication and its effect on the performance of the program. In Strong Scaling the number of cores are increased keeping the problem size constant. Thus, the aim of Strong Scaling is to study if an ideal theoretical speed-up can be obtained.

4.6.2 Single Node

A single node of the ARC2 facility consists of 16 cores, with 8 cores in each socket. For 16 MPI processes or cores decomposed in 3-D, the cache equations yield an optimal decomposition of $4\times4\times1$ instead of the $4\times2\times2$ topology given by the default `MPI_Dims_create()` function. The performance of various topologies for 16 processes for a problem of size $257\times257\times257$ is shown in Figure 4.15. At this problem size each process or sub-domain consists of $\approx \frac{16777216}{16} = 1048576$ single precision data points (without ghost points) per 3-D array. This approximately equates to a storage of 4 MB per array and since the working set of the unweighted Jacobi algorithm consist of 2 arrays, a total of $\approx$ 8 MB memory is accessed. This size clearly exceeds the shared L3 cache-per-core which is $\approx$ 2.5 MB/core. This problem therefore fits our criterion for a large problem as the working set is not fully contained in the cache-hierarchy.

In a SMP (Symmetric Multiprocessor) node, communication takes place through shared memory (intra-socket) and hence the Infiniband [66] network is not used. OpenMPI 1.6.5 [127] uses the SM (Shared Memory) component which operates on a copy-in/copy-out strategy as opposed to a zero-copy scheme used by the KNEM[3] library [130, 131]. In our simulation, even

[3]KNEM is a high performance intra-node communication library which performs a direct copy of large

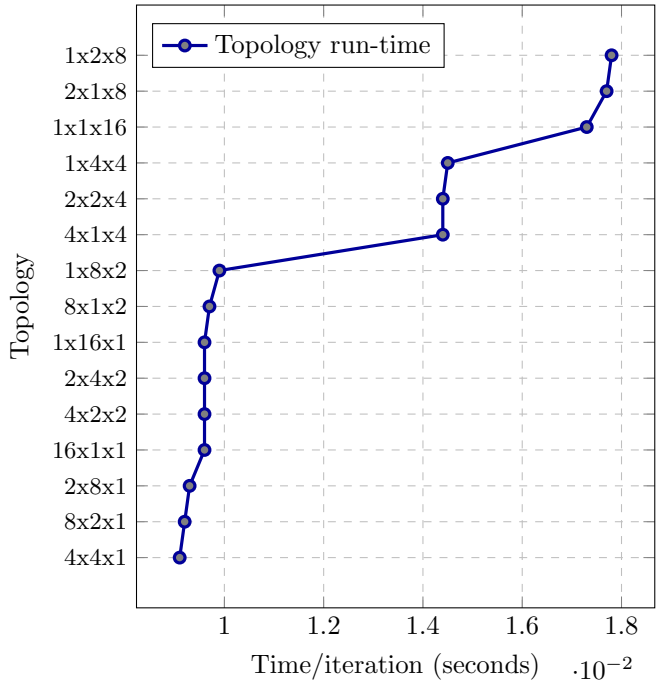


Figure 4.15: Time/iteration Vs Topology for 16 processes (single SMP node of ARC2) on problem size=257^3, $\approx$ 1048576 cells/process

when the neighbouring process of a process does not exist (i.e. is `MPI_PROC_NULL` according to MPI), we choose to send data to it. The MPI standard states that the operation of sending data to a `MPI_PROC_NULL` process completes and returns immediately [48]. Practically, this is where a Cartesian Topology is extremely useful as it gives a symmetric structure to the code by not differentiating between calls made to `MPI_PROC_NULL` processes and processes with valid ranks.

Points in Figure 4.15 can be visually grouped as families of topologies and hence at least three families can be observed. The performance gain for the best topology ($4 \times 4 \times 1$) over the topology minimizing communication ($4 \times 2 \times 2$) is approximately 4%, while compared to the worst topology ($1 \times 2 \times 8$) it is approximately 48%. It can be noticed that even topologies such as $8 \times 2 \times 1$, $2 \times 8 \times 1$ outperform the default `MPI_Dims_create()` (MDC or standard) topology which yields close-to-cubic sub-domains. For all the topologies which perform better than the communication volume minimizing topology the common factor is that their $D_z < D_{sz}$, where D_{sz} is the number of processes in the Z-direction in the standard topology. For all topologies having performance lower than the default MDC, the value of $D_z \geq D_{sz}$ (there is negligible difference between the performance of $1 \times 16 \times 1$ and $4 \times 2 \times 2$ and hence it is not counted as

messages (several kilobytes) between processes within the Linux kernel. KNEM is generic and offers asynchronous completion modes [130].

an exception). This observation is in good agreement with our model.

Using our model we search for the solution of $D_x D_y = P = 16$ and find that $D_x = 4$, $D_y = 4$ satisfies it such that $P_x = P_y$. It may be noted that it is not always possible to find $D_x = D_y$. When $D_x \neq D_y$, we find the closest D_x, D_y such that $D_x D_y = P$ holds while keeping $D_z = 1$. When D_x, D_y are found, we systematically consider the next best topologies to have the X and Y components as $2D_x$ and $\frac{1}{2}D_y$ or $\frac{1}{2}D_x$ and $2D_y$ while keeping $D_z = 1$. Applying this rule to Figure 4.15 we predict the next highest performing topologies to be $8 \times 2 \times 1$ and $2 \times 8 \times 1$, which coincides with the experimental values.

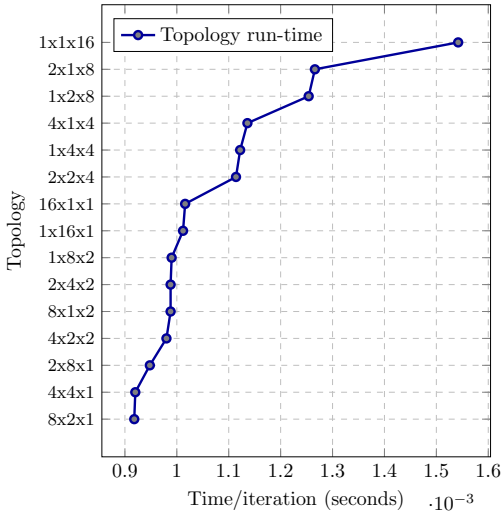
Figures 4.16a, 4.16b and 4.16c show the execution times of problems of sizes $129^3, 321^3$ and 513^3 on 16 cores of a single node of ARC2, respectively. The *Working Set Size* (WSS) of two arrays using the unweighted Jacobi method for these problems ranges from 1 MB to 64 MB. The following can be noted about Figures 4.16a, 4.16b and 4.16c:

- In no case is the default `MPI_Dims_create()` (MDC) topology of $4 \times 2 \times 2$ the optimal.
- As discussed above, the three predicted topologies, namely, $8 \times 2 \times 1$, $2 \times 8 \times 1$ and $4 \times 4 \times 1$ consistently outperform the MDC and other topologies.
- With an increase in the problem size, additional topologies, for e.g., $1 \times 16 \times 1$ and $16 \times 1 \times 1$ also outperform the MDC.
- Even though the problem of size 129^3 completely fits in the shared L3 cache, the MDC is still not the optimal.
- The percentage difference between the highest performing topology and the MDC is at least 6% for all the cases.

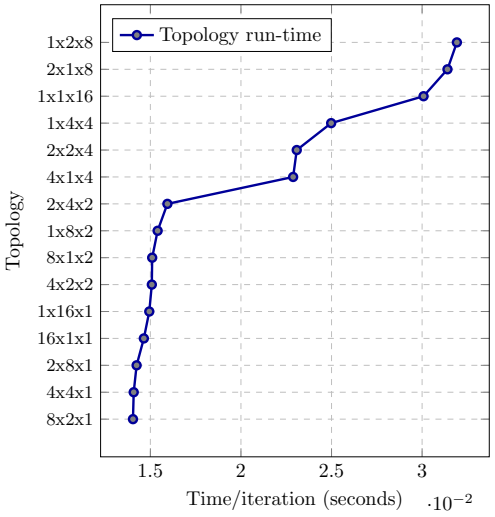
It is very difficult to optimize communication as compared to computation. For the computation example optimizations include loop unrolling, loop interchange, loop fission/fusion and tiling, etc. Optimizing communication can involve placement of neighbours of a process at close-by nodes, using specialized libraries such as KNEM [130] for reducing intra-node latencies, or techniques to reduce data copy time or combining small messages to be sent as a single message. The latter category, where communication mechanisms are optimized, are much harder to implement. Further, the gap between computation and communication suggests that this is a computationally intensive (and memory bandwidth limited) problem. Hence, it is likely to benefit more by choosing a decomposition that reduces cache-misses naturally than choosing a decomposition that reduces communication.

4.6.2.1 Compiler Optimization

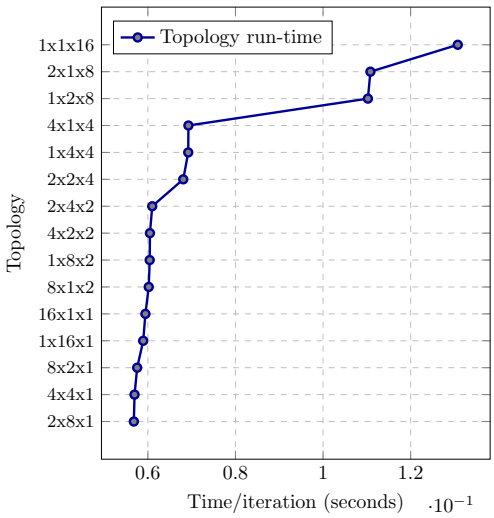
Modern compilers, the Intel compilers (16.0.2 and 17.0.1) in our case, do an excellent job at optimizing programs however it is often useful to hand-optimize codes unless this conflicts with



(a) $N = 129$, ≈ 131072 cells/process



(b) $N = 321$, ≈ 2048000 cells/process



(c) $N = 513$, ≈ 8388608 cells/process

Figure 4.16: Time/iteration Vs Topology for 16 processes (single SMP node of ARC2) and varying problem sizes

Table 4.7: Optimizations: Time per iteration with different compiler options for problem size=$161 \times 161 \times 161$ and cores=16

Compiler Optimization	**Time/iteration** (10^{-5} secs)
-O2	373
-O3	372
-O3 -xhost	384
-O3 -fp-model fast=1	361
-O3 -fimf-precision:low	370
-O3 -unroll4	374
-O3 -opt-prefetch=4	368
-O3, Tile Size=50, 2-D tiling, Rivera and Tseng [6]	394
-O2, Tile Size=50, 2-D tiling, Rivera and Tseng [6]	363

Table 4.8: Compiler Options: Brief explanation of various compiler options for the Intel C/C++ compiler

Compiler Option	**Description**
-O2	default optimization level, levels vary from -O0 to -O3, maximizes speed, includes automatic vectorization
-O3	includes -O2, additional loop and memory access optimizations, loop unrolling, loop blocking, scalar replacement
-xhost	uses most advanced instruction set on the host (such as AVX-256 or AVX-512)
-fp-model fast=1	sacrifices slight accuracy for speed, fast=2 also possible
-fp-model precise	stops optimizations which affect accuracy of floating point operations but allows FMA (Fused Multiply-Add)
-fimf-precision:low	sets low precision for math library functions to gain speed
-O3 -unroll4	sets the maximum number of times to unroll loops
-opt-prefetch=4	controls software pre-fetching, default is off

the standard compiler optimizations. Various compiler optimizations were tried in order to bring down the timing of the worst (theoretical) decomposition of $1 \times 1 \times 16$ with a problem of size $161 \times 161 \times 161$. We list the results in Table 4.7.

It can be noted from Table 4.7 that even with the `-O2` flag, the compiler generates almost optimal code. Further, the purpose of this experiment is not to search for the optimal tile at this problem size but to show that hand optimization may interfere with compiler optimization (`-O3` with Rivera and Tseng [6] 2-D tiling). A very brief explanation of the various compiler options is shown in Table 4.8. In this chapter, apart from the the `-O2` and `-O3` optimization levels, we do not use any of the additional options listed in Table 4.8. A detailed explanation is out of scope of the current work but can be found in the Intel C++ compiler documentation [132].

Table 4.9: Predicted and Actual cache-misses: Predicted Cache-Misses (PCM) and Actual cache-misses for Problem Size=161 × 161 × 161, Cores=16, Iterations=19353, Independent Compute Elements (ICE)=199712, PCM for ICE=62410

Topology	**PCM-planes**			**Total PCM**	**Observed Misses**	
	Z	X	Y		L1	L2
$16 \times 1 \times 1$	0	12800	0	1.45E+9	1.8E+9	4.0E+8
$1 \times 1 \times 16$	204800	0	0	5.16E+9	5.0E+9	1.4E+9
$1 \times 16 \times 1$	0	0	12800	1.45E+9	1.4E+9	5.3E+8

4.6.2.2 Cache-Misses

We try to predict the number of cache-misses using the estimates for cache-misses derived while creating the prediction model in Section 6.6. Table 4.9 shows the predicted cache-misses and the actual cache-misses. It can be seen that even when we don't incorporate Prefetching in our model, the predictions are fairly accurate. We combine the cache-misses of only the functions which have a significant contribution towards the total cache-misses. The profiler TAU (Tuning and Analysis Utilities) [119] was used to instrument the code and obtain the PAPI [120] (Performance Application Programming Interface) events such as `PAPI_L1_DCM` and `PAPI_L2_DCM`.

The Total Predicted Cache-Misses (Total PCM) in Table 4.9 is the sum total of the predicted cache-misses for the Dependent Planes (DP) and the Independent Compute (IC) kernel. The predicted cache-misses for the IC Elements (ICE) for all the topologies shown in Table 4.9 is equal i.e. 62410. The Total PCM can then be obtained by adding the PCM for ICE with the PCM-planes and multiplying by the total number of iterations (19353). In summary, the Total PCM = no. of Iterations × (PCM-planes + PCM-ICE). Table 4.9 shows that the Z decomposition is the worst, with maximum predicted cache-misses, whilst X and Y decompositions are exactly the same in our predictions. This serves as both verification and motivation for considering topologies such as $(2D_x)(\frac{D_y}{2})D_z$ and $(\frac{D_x}{2})(2D_y)D_z$. A subtle difference between X and Y decomposition is highlighted in Table 4.3 and Table 4.4 which relates to the maximum gap between the two elements. The observed L1 cache-misses for Y decomposition is less than the X decomposition although our predictions show that they should be equal. It is difficult to accurately predict the cache-misses due to Operating System fluctuations, interactions of various hardware components, hardware and software Prefetching policies and various other factors. But it can be seen from Table 4.9 that both X and Y decompositions perform significantly better than the Z decomposition. Further, the order of magnitude of the predicted and observed (actual) cache-misses is both 10^9 - instilling a good degree of confidence in our prediction scheme.

The predicted values in Table 4.9 are based on the estimated cache-misses in Table 4.5 (IC cache-misses) and Table 4.6 (DP cache-misses). The cache-misses in Table 4.5 are the estimated

cache-misses only for the IC of the `old` solution array (see RHS in Figure 4.10) in the unweighted Jacobi iterative method. Thus, when we add the IC cache-misses of the `new` solution array (see LHS in Figure 4.10), the total IC cache-misses become $\frac{(5+1)}{16}(P_x-2)(P_y-2)(P_z-2) = 74892$ for the problem of size $161 \times 161 \times 161$ (see Table 4.9). These IC cache-misses are the same for all three decompositions, namely, $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$. The estimated DP cache-misses in Table 4.6 and the predicted DP cache-misses in Table 4.9 are for a single plane only. It can be noted that the end processes (rank 0 and rank 15 processes) in the topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ communicate only a single plane. Hence, on an average, the DP cache-misses per process for $16 \times 1 \times 1$ is

$$\frac{1}{16} \times \left(14 \times 2 \times \frac{P_y P_z}{2} + 2 \times \left(\frac{6P_y P_z}{16} + \frac{P_y P_z}{2}\right)\right) = \frac{63 P_y P_z}{64} \approx 0.98 P_y P_z.$$

The expression above gives the average DP cache-misses per process for the X-decomposition as 25200. The total predicted cache-misses for $16\times1\times1$ is then $(25200+74892)\times19353 = 1.93\times10^9$. The Y-decomposition i.e. $1\times16\times1$ can be meted out the same treatment and hence the predicted cache-misses for this topology is also 1.93×10^9. For the Z-decomposition of $1 \times 1 \times 16$, the predicted IC cache-misses remain the same as that for the X/Y-decompositions i.e. 74892. The average DP cache-misses per process for the Z-decomposition can be calculated as

$$\frac{1}{16} \times (14 \times 2 \times 8P_x P_y + 2 \times (6P_x P_y + 8P_x P_y)) = \frac{252 P_x P_y}{16} \approx 15.75 P_x P_y.$$

The expression above gives the average DP cache-misses per process for the Z-decomposition as 403200. The total predicted cache-misses for $1 \times 1 \times 16$ is then $(403200 + 74892) \times 19353 = 9.25 \times 10^9$. This is much larger than the actual L1 cache-misses for the Z-decomposition which is 5×10^9. For the topology $1 \times 1 \times 16$ at a problem size of 161^3, the value of $P_z = 10$ and thus accessing a data point $(i, j, 1)$ on Z_TOWARDS_U brings the data element (i, j, P_z) on Z_AWAY_U into the cache. Hence, as an estimate we take into account the cache-misses only due to a single Z-plane and the predicted cache-misses for the Z-decomposition can be approximated as $\frac{9.25\times10^9}{2} = 4.63 \times 10^9$ - a value which is very close to the actual cache-misses for the Z-decomposition (see Table 4.9).

It should be noted that in practice both the Z-planes are sent/received and updated immediately after one another. Since the depth of the ghost layer is one in the derivation of our model and experiments, a cache-line which contains a point on the Z-plane will also contain the next immediate point that is at a distance of two on the other Z-plane (as data is contiguous in the Z-direction). Thus, if the point (i, j, P_z) on Z_AWAY_U is contained in a cache-line, there is an extremely high probability that the point $(i, j+1, 1)$ on Z_TOWARDS_U will also be contained in that cache-line. Bringing a Z-plane into the cache increases the probability of bringing the other Z-plane into the cache. It is important to note that accessing a single X or Y-plane does not increase the probability of bringing the other X/Y-plane into the cache since the distance

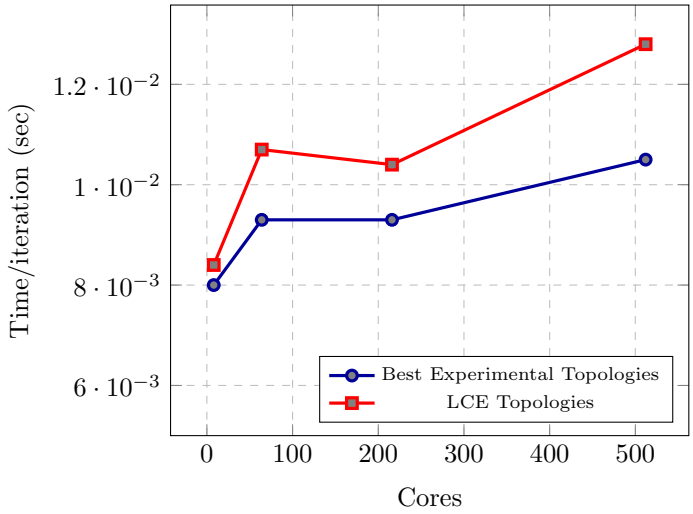


Figure 4.17: Weak Scaling for 8, 64, 216, 512 cores, Cells/core $\approx 10^6$, Iterations=10000, LCE (Least Communication Elements), best topologies ($4\times2\times1$, $16\times4\times1$, $6\times12\times3$ and $8\times32\times2$) Vs ($2\times2\times2$, $4\times4\times4$, $6\times6\times6$ and $8\times8\times8$), respectively.

between the two planes of each type is much larger as compared to the Z-planes. Interestingly, as a consequence of this behaviour that is specific to the Z-plane, the sequence of update of the Z-planes may reduce the total cache-misses incurred by the two Z-planes. Stated precisely, the Z_AWAY_U plane should be updated before Z_TOWARDS_U. In summary, although our predictions for cache-misses are fairly accurate, deviations are expected because our high level model does not take into account the low level architectural details of the cache-hierarchy.

4.6.3 Multiple Nodes

Inter-node communications take place via the Infiniband interface - leading to an increase in communication time due to an added message latency (hops) and increased data in-flight time. We further note that because of the difference in the number of communication elements between a topology which minimizes local cache-misses and a topology that minimizes communication elements specifically, the time gap between the execution of topologies for our experiment is expected to reduce when the communication time increases due to inter-node communication.

4.6.3.1 Weak Scaling

Figure 4.17 shows the results of a Weak Scaling test for 8, 64, 216 and 512 cores with $\approx 10^6$ cells/core. We refer to the communication minimizing topology as *LCE* (Least Communication Elements) and the cache-minimizing topologies as *LCM* (Least Cache-Misses). The solution time per iteration of the best topology for each core count and problem size is plotted against

that for the default `MPI_Dims_create()` (MDC) topology i.e. the LCE topology (or the *standard* topology). It can be seen that the cache-minimizing topologies outperform the communication minimizing topologies consistently and the gap in performance even tends to increase with an increasing number of cores. It can be noted that it was not possible to obtain all possible permutations of decompositions for 216 cores as our implementation assumes that $(N_i - 1)\%D_i = 0$ where $i = x, y, z$ (further explained in the discussion below). The execution run-times of various permutations of decompositions is needed to experimentally verify that the LCM topologies outperform the other topologies. It can be noted that the best experimental topologies can be predicted from our model but may not be the topologies which minimize cache-misses (theoretically). The problem sizes that we consider for Weak Scaling with 8, 64, 216 and 512 processes are $201^3, 401^3, 601^3$ and 801^3, respectively.

With $P = 8$ cores, keeping $D_z = 1$, we obtain the closest (D_x, D_y)=(4,2) or (2,4). Our observation matches with this prediction that the topology $D_x \times D_y \times D_z = 4 \times 2 \times 1$ outperforms the default MDC (or LCE) of $2 \times 2 \times 2$. We can go a level deeper to obtain the next topology i.e. $4 \times 2 \times 1 \rightarrow 8 \times 1 \times 1$ or $2 \times 4 \times 1 \rightarrow 1 \times 8 \times 1$ but the imbalance between the process dimensions D_x and D_y becomes higher (8 times) and this indicates a strong deviation from the cache-minimizing conditions that state that $D_x = D_y$ (theoretically). With $P = 64$ cores, we obtain an LCM of $8 \times 8 \times 1$ but going a level deeper i.e. considering $(2 \times 8) \times (\frac{8}{2}) \times 1$ and $(\frac{8}{2}) \times (2 \times 8) \times 1$, we obtain a topology which is the highest performing topology (i.e. $16 \times 4 \times 1$). With 216 cores, when we consider a decomposition of $18 \times 12 \times 1$ or $12 \times 18 \times 1$, the condition that $(N - 1)\%D_i = 0$ does not hold. Specifically at a problem size of 601^3 (as mentioned above), $(601 - 1)\%18 \neq 0$. Considering $D_z = 2$, we obtain a topology of $12 \times 9 \times 2$ but again $(601 - 1)\%9 \neq 0$ and hence we can only consider $D_z = 3$. The highest performing topology for $P = 216$ is also $6 \times 12 \times 3$. The other predicted LCM topology could have been $12 \times 6 \times 3$ but it is outperformed by the former. We discuss the choice between choosing $12 \times 6 \times 3$ and $6 \times 12 \times 3$ in Chapter 6 where we introduce the concept of *Working Plane Set Size* (WPSS). It can be noted that not being able to obtain decompositions such as $12 \times 18 \times 1$ or $12 \times 9 \times 2$ is a limitation of our implementation which is not designed to take into account a load-imbalance while partitioning the 3-D structured grid. When the number of cores is 512, the first topology that we consider is $16 \times 32 \times 1$ (and $32 \times 16 \times 1$). It is important to notice that the due to the increasing problem size and keeping $D_z = 1$, increases the communication volume generated by the X and Y plane. This is one disadvantage of our high level model as it does not take into account the problem size but only the number of cores. Considering topologies such as $64 \times 8 \times 1$ and $8 \times 64 \times 1$ does not solve this problem as the imbalance between D_x and D_y again increases. Thus, the idea is to increase the value of $D_z \leftarrow 2 \times D_z$ and to try to find values of D_x and D_y such that $|D_x - D_y|$ is minimized. Thus, we consider $D_z = 2$ and $D_x = D_y = 16$. Considering the variation of this topology as in other cases i.e. $(\frac{16}{2}) \times (2 \times 16) \times 2$ and $(2 \times 16) \times (\frac{16}{2}) \times 2$, we obtain the highest performing topology that can be experimentally verified. It is to be noted

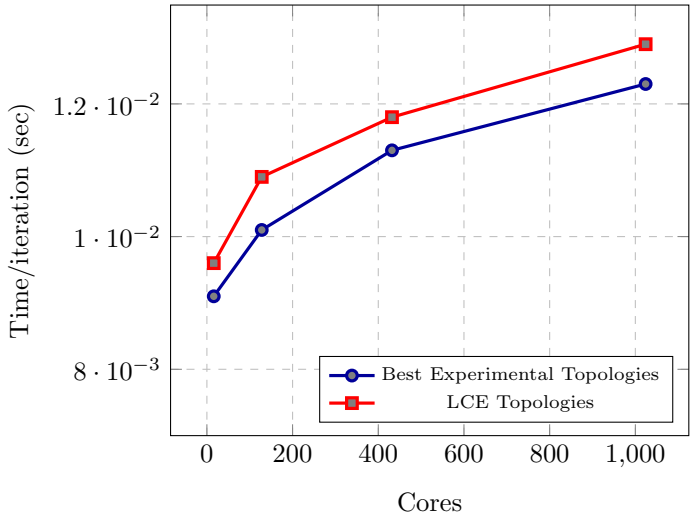


Figure 4.18: Weak Scaling for 16, 128, 432, 1024 cores, Cells/core=1048576, Iterations=10000, LCE (Least Communication Elements), best topologies ($4 \times 4 \times 1$, $16 \times 8 \times 1$, $12 \times 12 \times 3$, and $16 \times 32 \times 2$) Vs ($4 \times 2 \times 2$, $8 \times 4 \times 4$, $12 \times 6 \times 6$, and $16 \times 8 \times 8$), respectively.

that the default MDC (or LCE) gives $8 \times 8 \times 8$ as the topology for $P = 512$ cores. With $D_z = 2$, a value at which we obtain the optimal topology, is still four times less than $D_{sz} = 8$ (D_{sz} is the D_z for the default MDC or the LCE or the standard topology). Thus, it is not sufficient to minimize only the communication volume but to optimize the balance between minimizing cache-misses and minimizing communication.

Figure 4.18 shows the Weak Scaling between the two types of topologies for 16, 128, 432 and 1024 processors for a total of 1048576 ($\approx 10^6$) unknowns per core. The difference between this case and the previous case is that the number of cores is not a perfect cube and hence the default `MPI_Dims_create()` may return/returns $D_x \neq D_y \neq D_z$. A smaller gap between the two categories of topologies in the latter case is possibly because D_z is not the cube-root of the core count and hence generally less than D_x and D_y in the Least Communication Elements (LCE) case. If the processor count is a perfect cube then D_z ($= D_x = D_y$) grows exactly as $P^{\frac{1}{3}}$ for the LCE decomposition. This means that for a core count that is a perfect cube, the value of D_z returned by the default `MPI_Dims_create()` function is the highest. This in turn results in a much larger Z-plane than when D_z has a smaller value. As mentioned and verified previously, this leads to a deterioration in performance.

Since the process placement also plays a very important role when we venture out of the SMP, we show in Figure 4.19 the difference between two runs of the same problem size with identical number of cores but with random node allocation. The topology which minimizes communication i.e. $16 \times 8 \times 8$ has a variation of approximately 17% from the average execution

Table 4.10: Strong Scaling I: Strong Scaling for problem size=513^3, Iterations=500, t_{Best} is the minimum execution time, t_{MDC} is the execution time of default MDC

Cores	t_{Best} (sec)	t_{MDC} (sec)	**Best**	**MDC**	**WSS**(MB)
2	198.26	199.58	$1 \times 2 \times 1$	$2 \times 1 \times 1$	128
4	100.89	100.89	$2 \times 2 \times 1$	$2 \times 2 \times 1$	64
8	52.13	54.99	$2 \times 4 \times 1$	$2 \times 2 \times 2$	32
16	28.11	30.58	$4 \times 4 \times 1$	$4 \times 2 \times 2$	16
32	14.33	15.03	$8 \times 4 \times 1$	$4 \times 4 \times 2$	8
64	7.49	8.40	$8 \times 8 \times 1$	$4 \times 4 \times 4$	4
128	4.06	4.38	$8 \times 8 \times 2$	$8 \times 4 \times 4$	2
256	2.25	2.31	$8 \times 16 \times 2$	$8 \times 8 \times 4$	1
512	1.31	1.67	$8 \times 16 \times 4$	$8 \times 8 \times 8$	0.5

Table 4.11: Strong Scaling II: Strong scaling for problem size=1025^3, Iterations=500, t_{Best} is the minimum execution time, t_{MDC} is the execution time of default MDC

Cores	t_{Best} (sec)	t_{MDC} (sec)	**Best**	**MDC**	**WSS**(MB)
16	228.99	235.62	$1 \times 8 \times 2$	$4 \times 2 \times 2$	512
32	115.64	116.12	$2 \times 8 \times 2$	$4 \times 4 \times 2$	256
64	58.59	63.57	$4 \times 8 \times 2$	$4 \times 4 \times 4$	128
128	29.78	31.94	$4 \times 16 \times 2$	$8 \times 4 \times 4$	64
256	15.39	16.39	$8 \times 16 \times 2$	$8 \times 8 \times 4$	32
512	8.19	9.57	$8 \times 32 \times 2$	$8 \times 8 \times 8$	16

time. This shows that obtaining an optimal process placement is also important. A detailed discussion of topology mapping/process placement is outside the scope of this paper. The highest performing topology in both the runs (see Figure 4.19) is $16 \times 32 \times 2$ which is much closer to a 2-D domain partition than is $16 \times 8 \times 8$.

4.6.3.2 Strong Scaling

In Strong Scaling the problem size remains constant as the number of cores is increased. The computational time and the communication volume decreases as the problem size per core decreases but the communication time may increase due to the increasing distance between the cores. Table 4.10 and 4.11 show our Strong Scaling results for problems of sizes 513^3 and 1025^3 up-to 512 cores. The following observations can be made when strongly scaling a problem of size 513^3.

1. The execution on 2, 4 and 8 cores does not fully utilize the 16-core node of ARC2. Further, these 2, 4 and 8 processes are distributed in a round-robin manner by the scheduler i.e. mapped by socket so as to maximize the memory bandwidth per socket.

2. At no core count, except for $P = 4$ cores, is the default `MPI_Dims_create()` (or LCE or

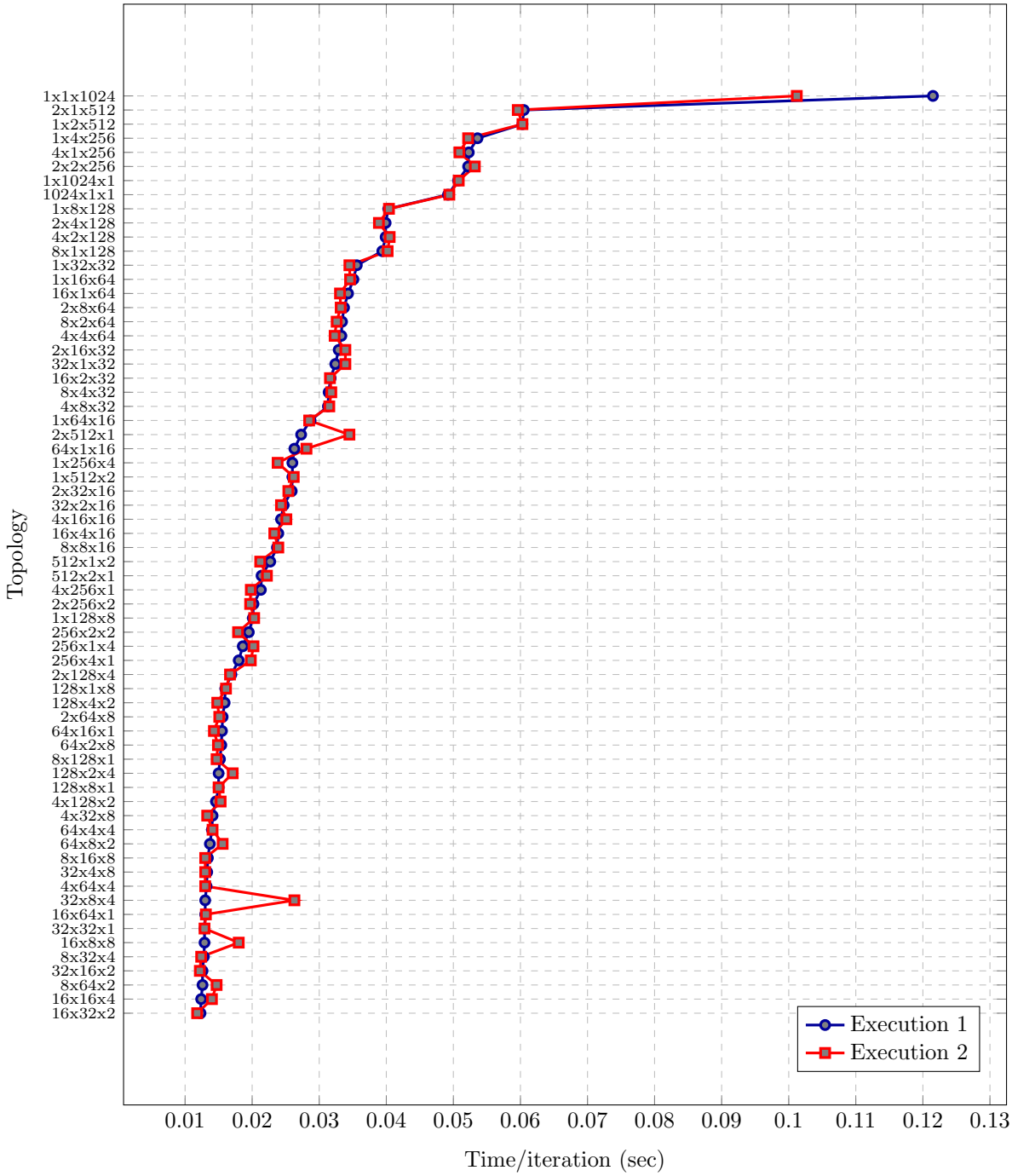


Figure 4.19: Topology Timings for two runs of Problem Size=1025^3, P=1024

standard topology), the optimal topology.

3. With two 3-D arrays in the unweighted Jacobi update, the Working Set Size (WSS) is 128 MB/core for $P = 2$ cores and decreases to 0.5 MB/core for $P = 512$ cores. The transition point occurs at $P = 128$ cores, where the WSS becomes 2 MB/core and starts fitting in the shared L3 cache per core (which is 2.5 MB/core).

4. At $P = 128$ and a WSS of 2 MB/core, since the data starts fitting into the shared L3 cache, the number of communication elements become a significant factor in determining the optimal partition and hence a topology such as $16\times8\times1$ or $8\times16\times1$ which communicates a maximum of 98304 ($= 2\times(64\times512+512\times32)$) elements is outperformed by a topology such as $8\times8\times2$ which communicates a maximum of 69632 ($= 64\times64+2\times(64\times256+256\times64)$) elements. It is to be noted that even when the default MDC at this core count i.e. $8\times4\times4$ communicates a maximum of 65536 ($= 2 \times (64 \times 128 + 128 \times 128 + 128 \times 64)$) elements, it still does not exhibit optimal performance. This can again be attributed to the better cache-efficiency of $8 \times 8 \times 2$. The topology of $8 \times 8 \times 2$ can be obtained from our model by setting $D_z = 2$ and keeping $D_x = D_y = 8$.

5. Even with $P = 512$ and $WSS = 0.5$ MB/core, the topology $8 \times 16 \times 4$ that communicates a maximum of 28672 ($= 2 \times (32 \times 128 + 64 \times 128 + 64 \times 32)$) elements, outperforms the default MDC of $8 \times 8 \times 8$ which communicates a maximum of 24576 ($= 2 \times 3 \times 64 \times 64$) elements.

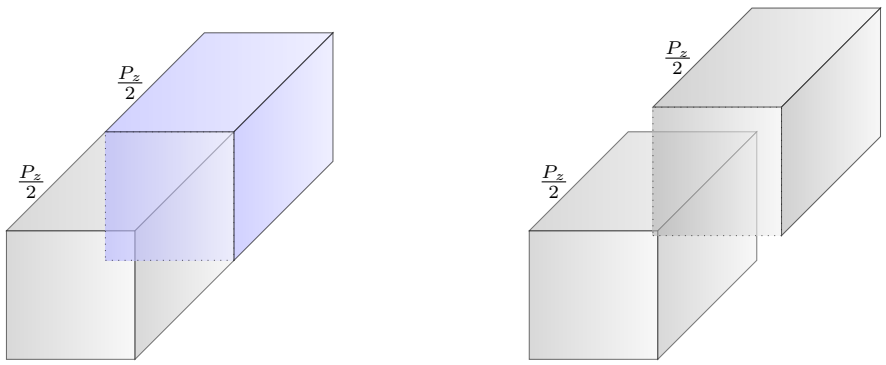
6. We conclude that when the WSS does not fit into the cache hierarchy, cache-misses are a much more significant factor in determining the optimal partition than the communication volume. As the problem size decreases and starts fitting into the cache hierarchy, the contribution of communication elements in determining the optimal domain partition increases but the cache-misses still influence the optimality. We again return to the discussion of WSS in Chapter 6 when we consider a hierarchy of grids of decreasing mesh spacing in parallel Geometric Multigrid.

Table 4.11 shows Strong Scaling on a problem of size 1025^3 with core counts ranging from 16 (16-core node) to 512 cores (32 nodes). This problem is 8 times larger than the previous problem of size 513^3. The following observations can be made for Table 4.11.

1. The WSS is approximately 512 MB per core at $P = 16$ cores and decreases to 16 MB at a core count of $P = 512$. The WSS thus, never fits completely into the cache-hierarchy.

2. The default MDC is not optimal for any core count at this problem size.

3. The optimal topology at $P = 16$ shifts to $1 \times 8 \times 2$ and outperforms the default MDC ($4 \times 2 \times 2$) by 2.81% even when the former communicates 33.34% more elements than the latter.



(a) Single sub-domain, two tiles of depth $\frac{P_z}{2}$ (b) Two sub-domains of depth $\frac{P_z}{2}$

Figure 4.20: Non-equivalence of tiled sub-domain and multiple sub-domains

4. At $P = 512$ and WSS of 16 MB/core the default MDC is outperformed by the topology $8 \times 32 \times 2$ by 14.42% even when the latter communicates 41.46% more elements than the standard topology of $8 \times 8 \times 8$. Thus, minimizing communication alone cannot be the sole criterion for obtaining the optimal domain partition.

5. For all the *Best* (topologies exhibiting minimum execution time) performing topologies, $D_y > D_x$. A plausible reason is that it decreases the Working Plane Set Size (WPSS) - a concept that we return to and discuss in Chapter 6.

6. The value of D_z for all the highest performing topologies increases to two at this problem size and we attribute it to the LRU (Least Recently Used) cache-eviction policy that purges data points before they can be reused. We discuss the effects of LRU in detail in Chapter 6, where we identify various factors affecting optimal sub-domain dimensions.

The experiments of Strong Scaling were performed without any cache tiling but it can be noted that a tile size of $\frac{N}{2D_z} \approx \frac{P_z}{2}$ in the Z-direction with P_z points is not the same as having no tiling with $\frac{N}{2D_z} = \frac{P_z}{2}$ data points. For example, a tile size of 512 in the Z-direction with $P_z = 1024$ is not equivalent to having no tiling with $P_z = 512$. Figure 4.20a and 4.20b illustrate this concept. The reason is that when the solution is updated at a depth of $\frac{P_z}{2}$ in the Z-dimension, the cache-logic in the tiled sub-domain prefetches data from the next tile and not the current tile. This situation cannot occur when the sub-domains are separate as the sub-domains are on separate processes.

Further, when we increase D_z, we trade-off an increase in the Z-plane update with a decrease in update in the independent computational kernel due to enhanced caching. An increase in D_z leads to a decrease in the value of P_z as $P_z = \frac{N-1}{D_z}$. This leads to an increase in the value of $P_x P_y$ and hence the size of the Z-plane increases - leading to an increase in the packing/unpacking/update time. As the size of P_z decreases, there is a greater probability that

the grid points in the cache are re-used again as generally caches typically implement a *pseudo* LRU (Least Recently Used) cache eviction policy. However, as the depth of the sub-domain (value of P_z) decreases, and the Z-plane size ($P_x P_y$) increases, there are an increased number of ghost points fetched while updating the Independent Compute (IC) kernel. This decreases the Cache-Line Utilization (CLU) as the ghost points are not used while updating the solution in the IC kernel. We discuss the LRU policy, CLU and other factors (and their effects on optimal sub-domain dimensions) in detail in Chapter 6. At this point, it would be correct to say that domain partitioning is dependent upon a multitude of Serial and Parallel Control Parameters (SCPs and PCPs) as mentioned in Chapter 1 and is not just a simple function of the communication volume (assuming a balanced load).

4.6.3.3 Communication Times of Planes

The data contained in the Dependent Planes must be sent to the neighbouring MPI processes. Within a single SMP (Symmetric Multiprocessor) node data can either be sent to a core within a socket (*intra-socket*) or across sockets (*inter-socket*). When multiple nodes are present, data can travel across nodes (*inter-node*) using the connecting network such as Infiniband. The destination node may be present in the same shelf as the source node (*intra-shelf* communication) or a different shelf (*inter-shelf* communication). The shelf itself may be present in the same rack (*intra-rack* communication) or a different rack (*inter-rack* communication). Thus, data can travel across a number of physical elements of the hierarchical network infrastructure. For simplicity and to limit the scope of the work we, at this point in time, discuss only intra-socket, inter-socket and inter-node communication, without considering shelves and racks. We revisit and elaborate this discussion in Chapter 6.

ARC2 uses a default `--bind-to-core --bysocket` intra-node process placement policy that assigns the first MPI process to the first core of the first socket, the second MPI process to the first core of the second socket, the third MPI process to the second core of the first socket and this pattern repeats until all MPI processes are assigned a core. For eight processes this is shown in Figure 4.21 where the MPI Cartesian Topology or decomposition is $2 \times 2 \times 2$. A line joining two different sockets i.e., the line joining blue balls and red balls in Figure 4.21, represents an inter-socket communication. An inter-socket communication does not take place through shared memory but through a high speed bus on Intel multiprocessor systems. In Figure 4.21 the red lines represent the Z-planes. Z-planes are thus communicated across sockets using the dedicated *Quick Path Interconnect* (QPI) [133] as opposed to a shared memory communication for intra-socket X and Y planes. The QPI is a multiple, point-to-point, low latency and high bandwidth bus connecting processors or other I/O/controller devices. It supports data speeds upto 25.6 GB/sec. It is used to access the remote memory in a multiprocessor system and also for cache-coherency across processors.

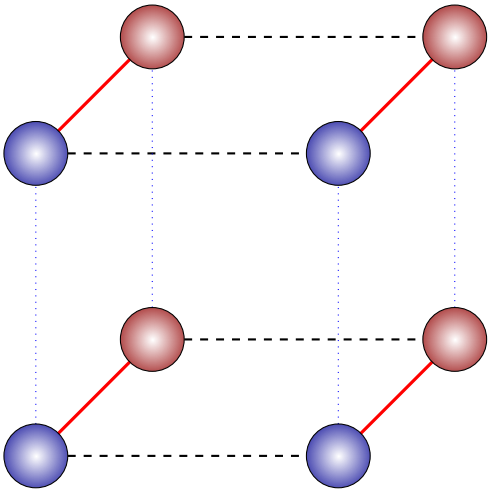
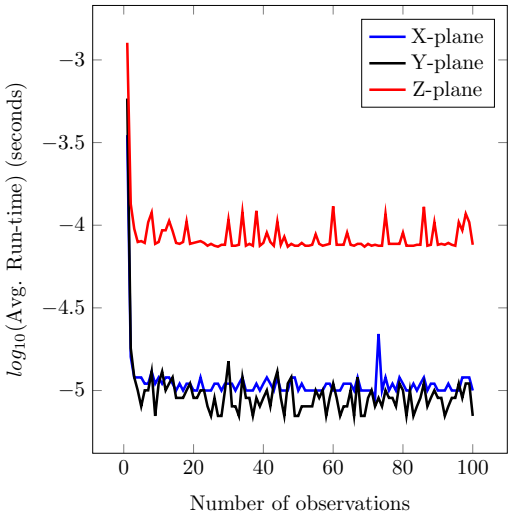


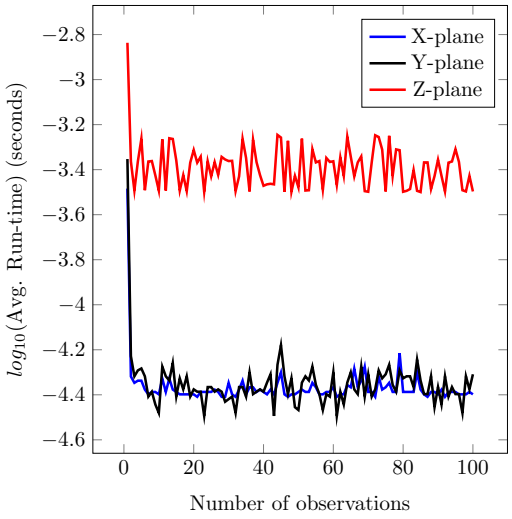
Figure 4.21: Cores in socket 0: blue balls, Cores in socket 1: red balls, Z-planes: very thick, red lines, Y-planes: thick, black, dashed lines, X-planes: thin, blue, dotted lines, Decomposition: $2 \times 2 \times 2$, QPI present on lines that connect different sockets, Mapping: `--bind-to-core --bysocket`

With 8 processes the default communication minimizing `MPI_dims_create()` function returns a decomposition of $D_x \times D_y \times D_z = 2 \times 2 \times 2$. With such a decomposition, a problem of size $129 \times 129 \times 129$ yields a sub-domain volume of $\frac{(129-1)(129-1)(129-1)}{2 \times 2 \times 2} = 64 \times 64 \times 64$ without the ghost cells. Thus, the size of each plane that is passed is $64 \times 64 \times 4$ bytes as we use a single precision float data type (`sizeof(float)`=4) here.

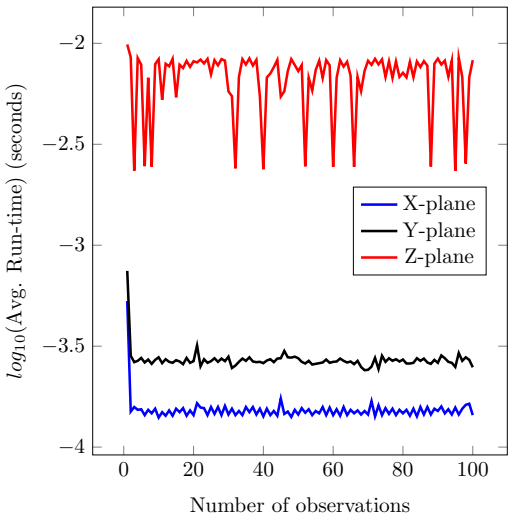
Figure 4.22 (*Log scale* on Y-axis) shows the average time taken by a single process to send an equal amount of data in the X, Y and Z planes. Even when considering larger problem sizes, the X/Y planes of similar dimensions always outperform the communication times of the Z-plane as shown in Figure 4.22. If we assume that the latency and bandwidth of the QPI is comparable to that for the shared memory then since the total amount of data remains the same for all three planes, the difference in timing must come (majorly) from the difference in packing times and not the transmission times. A comparison of latencies and effective bandwidth at different message sizes for a dual socket Intel Xeon 5160 can be found in [55]. The packing times are different as the data pattern to access individual data elements is different for the different plane types. This data access pattern is what gives rise to cache-misses when the data is non-contiguous. The topology chosen for this experiment was $2 \times 2 \times 2$ for 8 cores as it ensures an equal number of X, Y and Z neighbours for each process. As predicted in Table 4.6, the X/Y planes have an equal number of cache-misses but a higher packing efficiency is expected in the case of the X-plane as the maximum gap between data elements is only two (see Table 4.3). This does not hold true for the Y-plane as can be seen from Table 4.4. Thus, when sending equal sized, intra-socket X and Y planes to neighbouring processes, the X-plane is



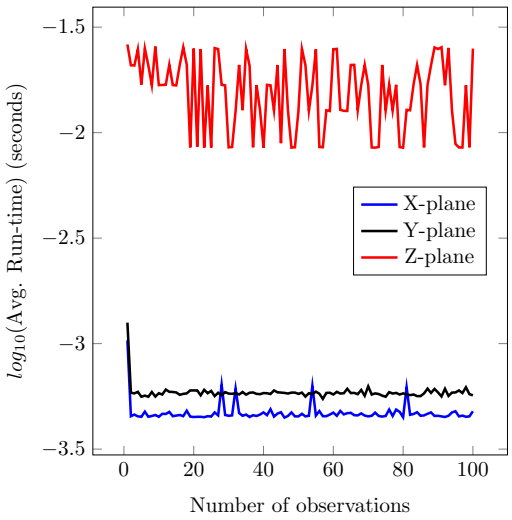
(a) Plane sizes of $64 \times 64 \times 4$ bytes



(b) Plane sizes of $128 \times 128 \times 4$ bytes



(c) Plane sizes of $256 \times 256 \times 4$ bytes



(d) Plane sizes of $512 \times 512 \times 4$ bytes

Figure 4.22: Average time taken to send X, Y and Z planes of same size with cores=8 (topology=$2 \times 2 \times 2$)

expected to take less time as compared to the Y-plane. This behaviour can be verified from Figure 4.22 which shows that the X-plane takes less time than the Y-plane for increasing data-sizes.

Figure 4.23 illustrates the same experiment for the communication times of equal sized X/Y/Z planes with 4 nodes (64 cores). The topology of $4 \times 4 \times 4$ here ensures that processes that are not at the boundary, have an equal number of X/Y/Z neighbour processes. Further, the total number of planes of each type is $3 \times 4 \times 4 \times 2 = 96$. The Y/Z-planes are sent to neighbour processes on the same node (intra-node) but X-planes travel across SMP's (inter-node using Infiniband). The Y-planes thus, take less time than X-planes on an average. The Z-planes still take more time than the X-planes, although the former uses QPI for communication. The major contributing component in the average timings of Z-planes is then due to the cache-misses incurring during its packing. Thus, for large but equal data sizes, inter-socket transmission of the Z-plane generally costs more than the inter-node transmission of the X-plane.

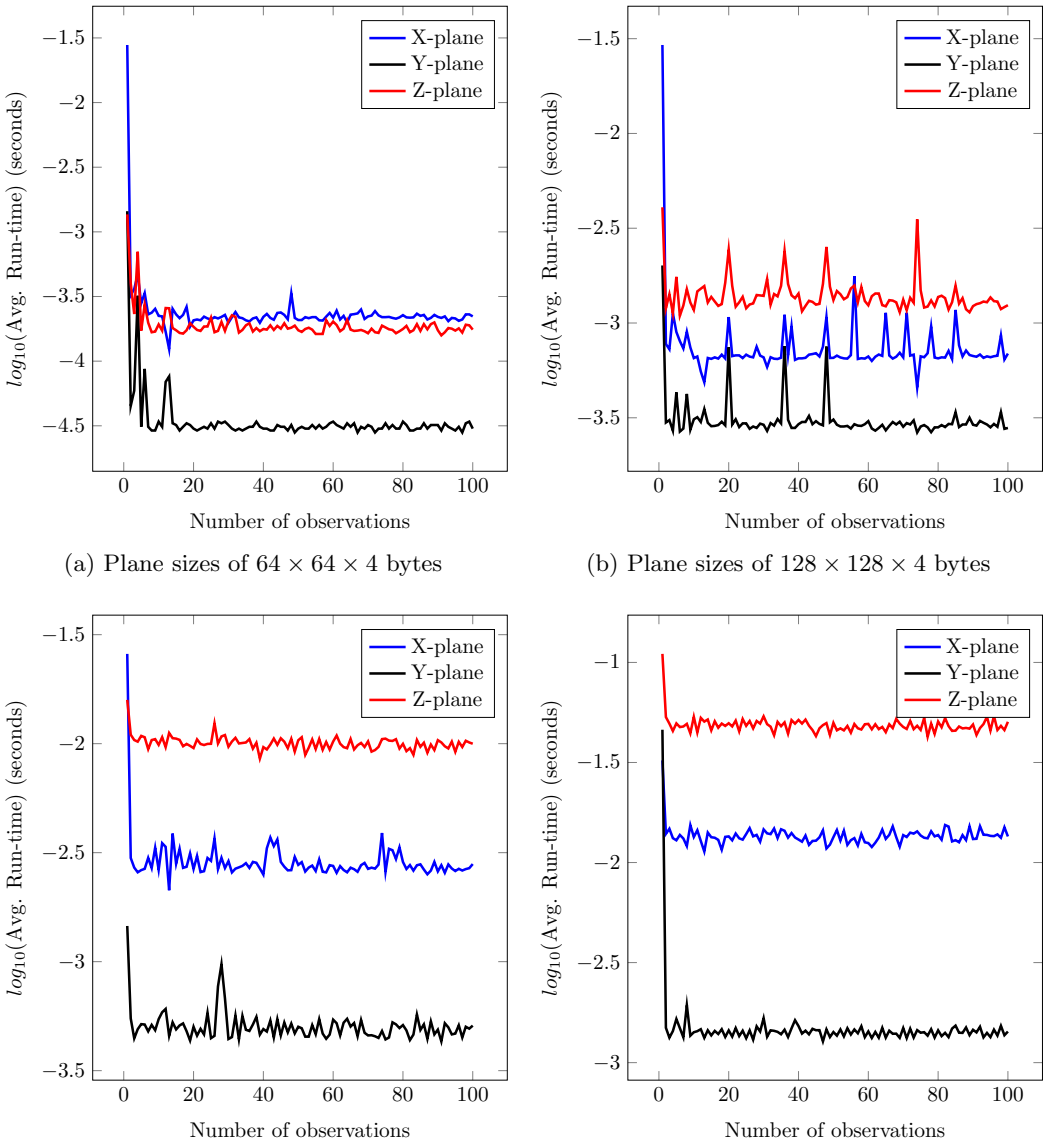
The MPI process-to-core mapping can be changed using process bindings in OpenMPI but we prefer to keep the default mapping and not venture into the field of process placement as it lies outside the scope of the current work. However, we do discuss another type of intra-node process placement `--bind-to-core --bycore` in Chapter 6 and show that our inferences hold for the latter process placement policy as well.

4.6.3.4 Planes Update Cache-Misses

When the Dependent Planes are updated after data is received from neighbour processes, both read and write cache-misses incur. These update cache-misses are shown in Table 4.6. Figures 4.24 and 4.25 show the cache-misses for two planes of sizes $64 \times 64 \times 4$ and $128 \times 128 \times 4$ bytes. As is predicted in Table 4.6, the cache-misses for the Z-plane are much higher than the cache-misses for the X/Y planes. Further, the X and Y plane update cache-misses are close to each other but the latter incurs higher cache-misses for all the plane sizes. This is due to a maximum gap of two ghost data points between the data elements of the X-plane (see Table 4.3). For the Y-plane this gap is much larger (see Table 4.4). Thus, in practice it is much easier for the cache-logic to prefetch the data for the X-plane as compared to the Y-plane. As mentioned, we do not take into account the factor of prefetch in our derivation of the model.

4.6.3.5 Increasing Bandwidth-per-core

When a node is completely utilized, the memory bandwidth per core is minimal as all the 8 cores of a socket on ARC2 share the same *Last Level Cache* (LLC) and the main memory module. Since simulation of a PDE using stencil based methods is a memory-bandwidth intensive procedure [12], we experiment with partial utilization of nodes. Though an under-utilization of resources, this can find a potential application in solving the coarsest grid (s) on a subset of processes in parallel multilevel methods such as Geometric Multigrid [59, 63]. Our experiments



(a) Plane sizes of $64 \times 64 \times 4$ bytes

(b) Plane sizes of $128 \times 128 \times 4$ bytes

(c) Plane sizes of $256 \times 256 \times 4$ bytes

(d) Plane sizes of $512 \times 512 \times 4$ bytes

Figure 4.23: Average time taken to send X, Y and Z planes of same size with cores=64 (topology=$4 \times 4 \times 4$)

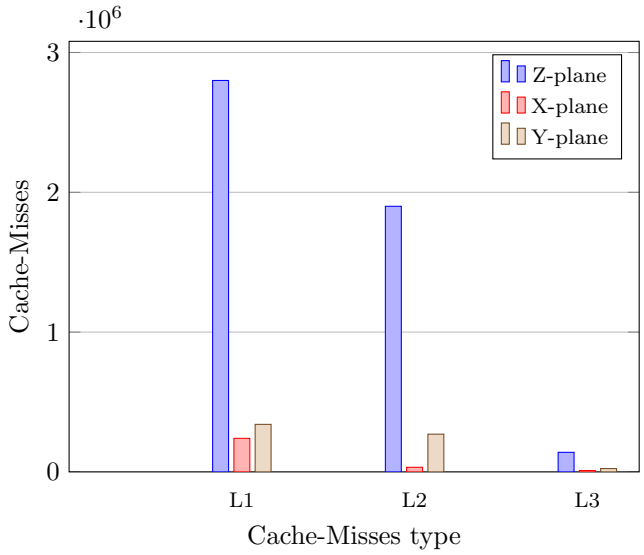


Figure 4.24: Cache-Misses for updating solution of Z/X/Y planes of equal sizes with Cores $P = 64$, planes of size $64 \times 64 \times 4$ bytes

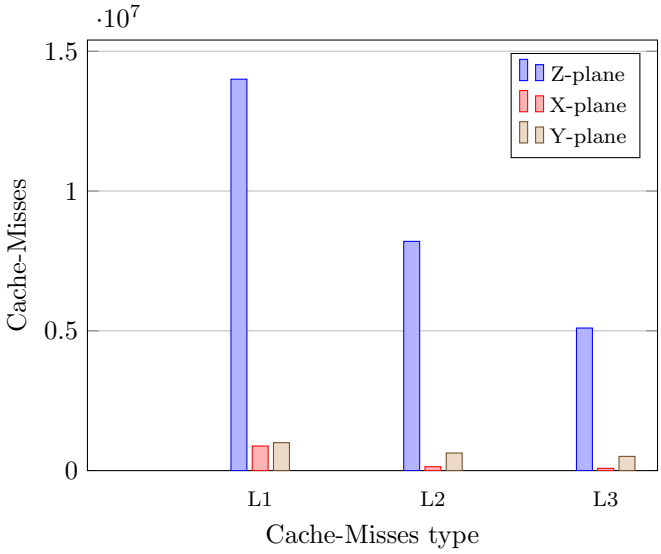


Figure 4.25: Cache-Misses for updating solution of Z/X/Y planes of equal sizes with Cores $P = 64$, planes of size $128 \times 128 \times 4$ bytes

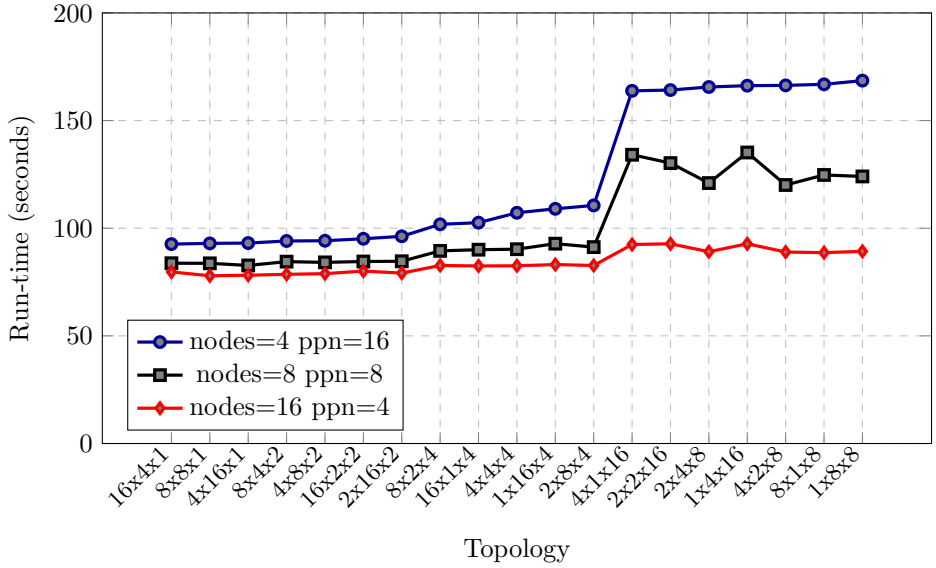


Figure 4.26: Topology Timings for 64 cores, Problem Size=401 × 401 × 401, Iterations=10000, Cells/core≈ 10^6 for varying Memory Bandwidth per core

with $P = 64$ cores and a problem of size 401^3 is shown in Figure 4.26. As the *processes-per-node* (*ppn*) decrease, the application performance increases. The reason for this is the reduced contention for the LLC and the main memory modules. Theoretically, there should come a point where the benefits of increasing memory bandwidth per core will be balanced by the increasing global and local synchronization time. This experiment strengthens the observation that stencil codes are memory bandwidth intensive. Interestingly, there are many topologies that outperform the traditional communication minimization topology of $4 \times 4 \times 4$ for varying number of processes per node. For all such topologies the values of $D_z \leq 4$ - an observation that lends support to our model. As the number of nodes increase and the processes-per-node decrease (while maintaining the total process count to 64), the performance gap between topologies decreases. This again reaffirms the fact that the efficiency of stencil-computations can be enhanced by increasing the available shared L3 cache-per-core and decreasing the contention for this shared memory. In a shared cluster of nodes in a multi-user environment, this under-utilization of nodes is not advisable.

4.6.3.6 19-pt Stencil

The 7-pt stencil considered in the tests discussed up to this point consists of six neighbour mesh points, corresponding to the six faces of a cube. Instead of using a 7-pt stencil, a 19-pt stencil [134] which includes the neighbour points on the twelve edges of a cube in addition to the points on the six faces can also be used to update the solution. The Jacobi iteration corresponding to a 19-pt stencil is illustrated in Figure 4.27. The sum of the weights of the

```
new[i][j][k] = (1/4.0)*
                (
                (1/6.0)*
                (
                old[i][j-1][k-1] + old[i][j-1][k+1] +
                old[i][j+1][k-1] + old[i][j+1][k+1] +
                old[i+1][j-1][k] + old[i+1][j+1][k] +
                old[i+1][j][k-1] + old[i+1][j][k+1] +
                old[i-1][j-1][k] + old[i-1][j+1][k] +
                old[i-1][j][k-1] + old[i-1][j][k+1]
                )
                 +
                (1/3.0)*
                (
                   old[i][j-1][k] + old[i][j+1][k] +
                   old[i][j][k-1] + old[i][j][k+1] +
                   old[i+1][j][k] + old[i-1][j][k]
                )
                ) ;
```

Figure 4.27: 19-pt stencil used in unweighted Jacobi, `new` and `old` are 3-D data arrays

points on the edges and the points on the faces equates to one. That is, if w_f, w_e are the weights of the face and edge neighbours respectively and n_f, n_e denote the number of mesh points on faces and edges in the 19-pt stencil, then

$$w_f n_f + w_e n_e = \frac{1}{24} \times 12 + \frac{1}{12} \times 6 = 1. \tag{4.3}$$

The 19-pt stencil adds a layer of complexity to the communication pattern since the corner points are also required from neighbouring processes. Thus, in addition to neighbour processes which share the faces with the process under consideration, twelve more neighbours which share the edges must also communicate with the current process. The term sharing faces or edges means that the next-to-boundary data of one process appears as the ghost data of another process. There are two methods [135] in which these edges and corner points can be exchanged with neighbouring processes. The first method is to send them directly to neighbouring processes. This requires explicitly calculating the MPI ranks of the neighbouring processes which share edges and the corner points in the 19-pt stencil. The second method is to use wider halos/ghost zones and sending them in two steps to the neighbouring processes. We use the latter method in our implementation by first sending the halos to neighbouring processes in the Y-direction, followed by a send in the Z-direction and then finally in the X-direction. It can be noted that communication in a specific direction must complete before communication can begin in the next direction. With three directions, namely, X, Y and Z, a total of six different permutations are possible and any order out of these can be used for communicating the edges and corner points.

Figure 4.28 shows the execution times of various topologies on a single node of ARC2 (i.e. 16 cores) using a 19-pt stencil. Interestingly, when the problem size is $65 \times 65 \times 65$, i.e. ≈ 16384 mesh points per core, the communication volume minimizing topology outperforms the remaining topologies. With 16384 single precision floating point values, each Jacobi array has a size of approximately 64 KB. Since the Working Set Size (WSS) contains two such arrays, the total data which is accessed is approximately 128 KB - a size small enough to fit into even the unified L2 level cache of 256 KB. Thus, at this size we do not expect a significant number of cache-misses and the number of communicated elements becomes a very significant factor. As the size of the domain is increased to $129 \times 129 \times 129$, the cache-minimizing topologies outperform the communication minimizing topology. The same pattern can be seen when the domain size is further increased to $257 \times 257 \times 257$ and $513 \times 513 \times 513$. With $P = 64$ and a problem of size $401 \times 401 \times 401$, the best performing topology is again $8 \times 8 \times 1$ and not $4 \times 4 \times 4$. The former outperforms the latter by 33.17% (see Figure 4.29).

4.7 Generality - Revisiting Assumptions

Any model must provide an insight into a process. Our model makes this attempt by viewing parallel domain partitioning for stencil-based codes in a different light as compared to the orthodox approach of minimizing the communication volume. The purpose of this section is to broadly discuss its generality, keeping in mind the assumptions used in its derivation. The assumptions that cannot be relaxed represent limitations of our model whereas other assumptions can be relaxed, making the applicability of our approach broader than Table 4.1 might initially suggest. The need for this discussion stems from Table 4.1 which divides the assumptions into several logical classes.

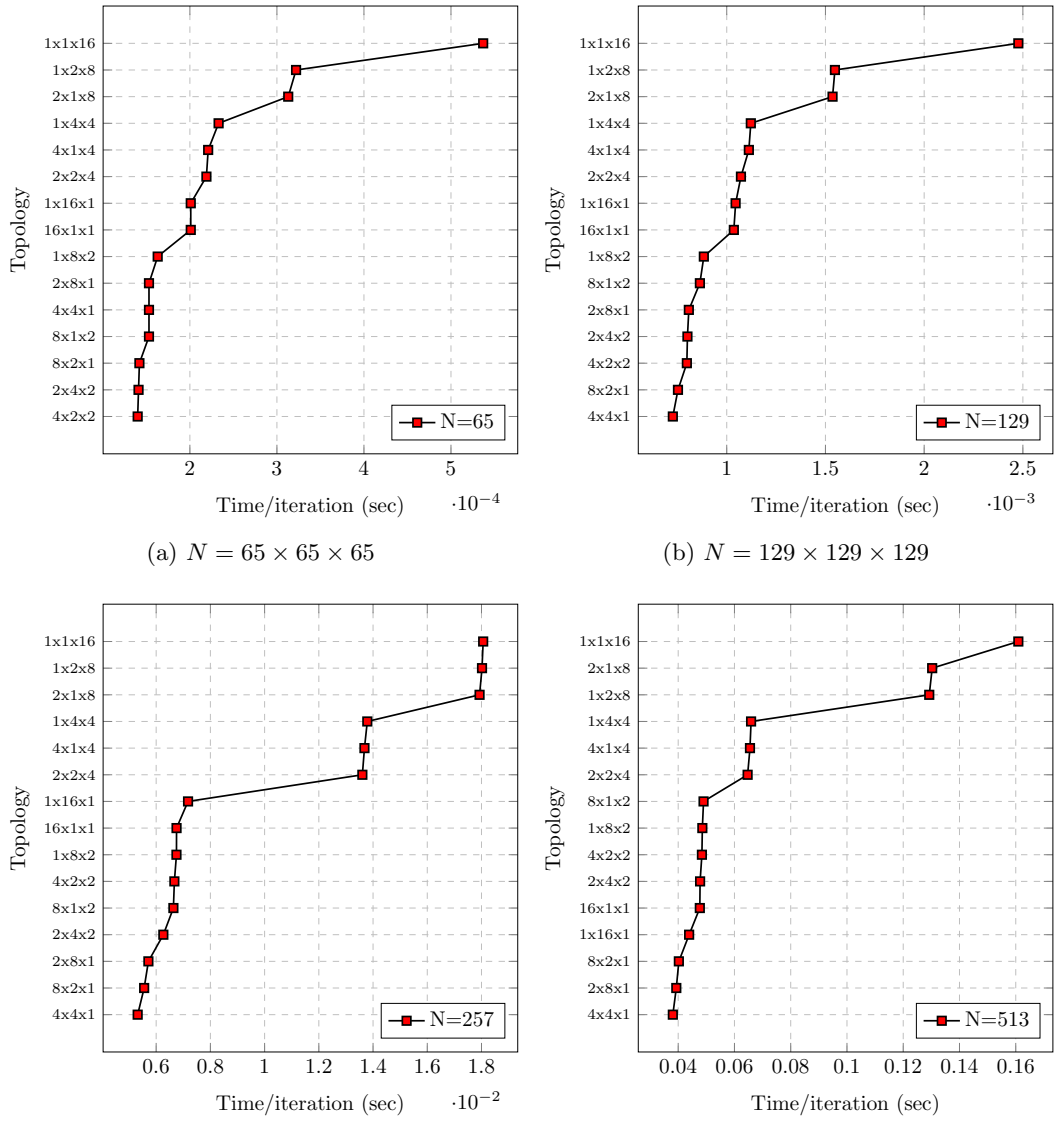
4.7.1 PDE class

As discussed in Chapter 2, PDEs can be classified as Elliptic, Parabolic or Hyperbolic. Although the work in this thesis concerns only Elliptic PDEs, our model can be extended to other types of PDEs as well. This section discusses its extension to *Parabolic* and *non-linear* PDEs.

4.7.1.1 Parabolic PDEs

Our model is not specific to second order Elliptic PDEs and can be extended to second order Parabolic PDEs when they are solved implicitly at each time step. As an example of the latter, we consider the Parabolic equation of heat conduction shown in Equation (4.4)

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}, \tag{4.4}$$



(a) $N = 65 \times 65 \times 65$

(b) $N = 129 \times 129 \times 129$

(c) $N = 257 \times 257 \times 257$

(d) $N = 513 \times 513 \times 513$

Figure 4.28: Time per iteration (seconds) of topologies using a 19-pt stencil when $P = 16$ with varying data sizes on a single node of ARC2, Intel compiler 17.0.1, Optimization level: -O2, OpenMPI 1.6.5

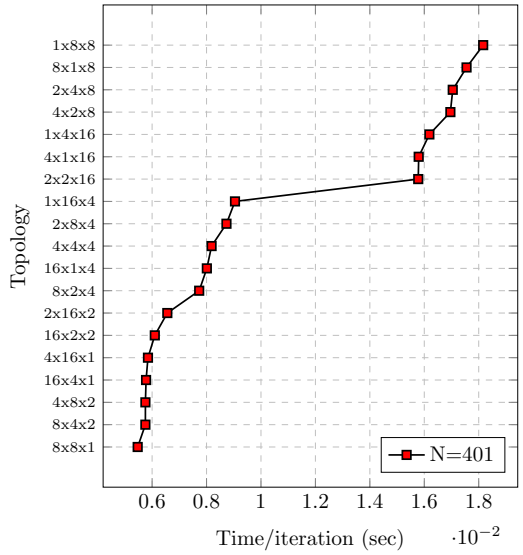


Figure 4.29: Time per iteration of various topologies using a 19-pt stencil with $P = 64$ and $N = 401 \times 401 \times 401$, Intel compiler 17.0.1, OpenMPI 1.6.5

where u gives the temperature in a thermally insulated rod at distance x from the origin after t seconds and κ is a constant. It is assumed that the boundary conditions and initial conditions are both known. Using FDM we can use a *backward difference* formula for approximating the LHS and a *central difference* formula for the RHS (for the dimensionless form of Equation (4.4), i.e. $\kappa = 1$ [22]). This is shown in Equation (4.5).

$$\frac{u_i^{t+1} - u_i^t}{\Delta t} = \frac{u_{i+1}^{t+1} - 2u_i^{t+1} + u_{i-1}^{t+1}}{h^2} \tag{4.5}$$

In a single spatial dimension, u_i^{t+1} is the unknown temperature at distance ih (i is the index of a point and $i = 0, 1, 2, ..., N$, h is mesh spacing) at time $t + 1$ which is determined explicitly in terms of the temperatures at time t. If the dimensionless form of Equation (4.4) is assumed to be satisfied at the midpoint of time t and $t + 1$ then we can approximate its RHS by the arithmetic mean of the FDM approximation at time t and $t + 1$ (*Crank-Nicolson* method) [22]. This gives rise to Equation (4.6):

$$\frac{u_i^{t+1} - u_i^t}{\Delta t} = \frac{1}{2}\left(\frac{u_{i+1}^t - 2u_i^t + u_{i-1}^t}{h^2} + \frac{u_{i+1}^{t+1} - 2u_i^{t+1} + u_{i-1}^{t+1}}{h^2}\right). \tag{4.6}$$

In both cases, the terms u_i^{t+1} create a system of N simultaneous equations for N unknowns. These can be solved using the same method as employed for a second order Elliptic PDE. Thus, although for the purpose of simplicity of discussion and limiting the scope of the current work we make use of Elliptic PDEs, the method is equally applicable to a standard implicit time discretization of a second order Parabolic PDE.

4.7.1.2 Non-linear PDEs

In Chapter 2 we differentiated between linear and non-linear PDEs. The discretization chosen for non-linear PDEs may result in a system of linear or non-linear algebraic equations [136]. A system of non-linear algebraic equations can be solved using methods such as *Picard* iteration or the well known *Newton's* method. To briefly describe Newton's method, consider a non-linear scalar equation $F(u) = 0$. Expanding $F(u)$ using *Taylor's series* about an approximation v, we obtain:

$$F(v+s) = F(v) + sF'(v) + \frac{s^2}{2}F''(v). \tag{4.7}$$

Assuming $u = v + s$ is a solution and neglecting the higher order terms in Equation (4.7), we obtain $0 = F(v) + sF'(v)$ and thus, $s = -\frac{F(v)}{F'(v)}$. Thus, v can now be updated as $v \leftarrow v - \frac{F(v)}{F'(v)}$, where the $\leftarrow$ denotes an assignment. This method can be extended to a system of non-linear equations denoted by $F(u) = 0$, where the parenthesis around the vector of unknowns u denotes that the operator F is non-linear. Thus, in the vector form:

$$F(u) \equiv \begin{bmatrix} f_1(u_1, u_2, \ldots, u_n) \\ f_2(u_1, u_2, \ldots, u_n) \\ \cdots\cdots\cdots \\ f_n(u_1, u_2, \ldots, u_n) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{4.8}$$

Expanding around v yields $F(v+s) = F(v) + J(v)s$, where $J(v)$ is the *Jacobian* matrix [63]. Assuming $v + s$ is a solution then $F(v+s) = 0 = F(v) + J(v)s$ and hence $s = -[J(v)]^{-1}F(v)$. This can be written as a linear system $J(v)s = F(v)$ for known v and unknown vector s. Applying an iterative method to this system will again involve a stencil computation that can be considered with our methodology. Thus, v can be updated as $v \leftarrow v + s$.

4.7.2 Boundaries

Our derivation and test problem assumed Dirichlet boundary conditions. These conditions specify a given value of the unknown variable u at the boundary. Instead of directly specifying the value of the unknown variable, its normal derivative in the direction of the outward normal can be specified. Such boundary conditions are called *Neumann* boundary conditions [25]. In Chapter 6 we experiment with and discuss the applicability of the model to a *mixed Dirichlet-Neumann* boundary problem. For the Neumann boundary, the mesh vertices themselves are treated as unknown values. A fictitious boundary is then introduced which is updated according to a finite difference approximation of the derivative of u at the physical boundary. As an effect, in addition to the update of planes, the Neumann boundary must also be updated. This can introduce additional cache-misses. It should be noted that although a Neumann boundary is updated, it is never communicated to any other process as it is a boundary. Since the data at the Neumann boundary is not communicated, the packing/unpacking cache-misses cost is zero.

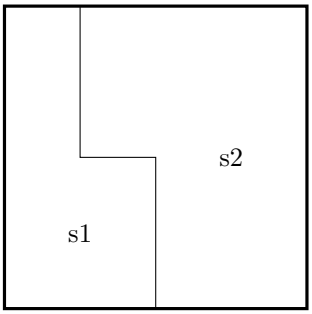


Figure 4.30: Example of an Irregular cut on a square domain that divides the domain into two sub-domains s1 and s2 which do not have identical shape

Thus, one can argue that the reduced cost of cache-misses eliminated due to unpacking/packing of data is offset by the cost of updating a Neumann boundary. Thus, although the test problem in this chapter assumes Dirichlet boundary conditions, the model in general is applicable to other boundary conditions such as Neumann or *Robin*, the latter being a more general boundary condition.

4.7.3 Structured Meshes and Decomposition

The limitation of requiring a structured mesh cannot be relaxed as there is no spatial locality present in unstructured meshes (as far as storage of data elements are concerned in the memory). The vertices of a regular structured mesh have a natural ordering. Our model is based on the data access pattern that is inherent for contiguous data. Since data is contiguous, the cache-line is able to fetch useful data from the memory as it is in the vicinity of the data elements being updated/packed/unpacked. Even with structured meshes, we take into account only cuts parallel to the Cartesian axes and not irregular cuts. In case of irregular cuts the Cartesian topology cannot determine the sub-domain shape. Thus, the Cartesian topology in a way is decoupled from the shape of the sub-domain when partitioning the domain using irregular cuts. Figure 4.30 shows an example of a square domain partitioned using an irregular cut. It can be seen that the cut is both parallel to the X and Y Cartesian axes. Thus, we do not allow cuts or process decompositions which are parallel to more than one axis or are not parallel to any axis. A structured square domain can be partitioned irregularly in several ways and assigned to MPI processes. It is difficult to determine the neighbours of each process and they need to be stored separately on each process. It is again difficult to define MPI data-types for passing such irregular boundary data. Thus, the work in this thesis remains restricted to structured 3-D domains and with straight cuts parallel to a single Cartesian axis. An alternate way of expressing this is that we require the sub-domain shapes to be identical when applying process decompositions parallel to the Cartesian axes.

Another assumption that we make in Table 4.1 is that the domain be cubic. This assump-

tion can be relaxed to a cuboid shaped domain without any modifications in the model. Since a cuboid can be specified using a problem of size $N_x \times N_y \times N_z$ where $N_x \neq N_y \neq N_z$ in general, our cache-minimizing condition $P_x = P_y$ implies that $\frac{N_x-1}{D_x} = \frac{N_y-1}{D_y}$ and further $P_z = N_z - 1$. For reasons of simplicity and without loss of generality, we choose cubic domains to validate our hypothesis.

To elaborate and justify the simplification to global domains that have all three dimensions the same, we note that when $N_x = N_y = N_z$, the `MPI_DIMS_CREATE()` topology creates sub-domain shapes which are cubic or as close to cubic as possible. Due to such sub-domain shapes/sizes, the total communication volume exchanged between processes is minimized. This produces the best partitioning case for the standard topology and serves as an optimal reference point as our aim is to show that minimizing communication volume is not the sole criterion for optimal domain partitioning. Thus, considering domains such that $N_x = N_y = N_z$ guarantees that we compare cache-minimizing topologies against the communication volume minimizing partition created by the MDC topology. This is not true when $N_x \neq N_y \neq N_z$ as the default MDC may or may not produce sub-domain shapes that minimize the total communication volume. The reason is that the `MPI_DIMS_CREATE()` function does not take into account the various mesh dimensions. Thus, such non-cubic global domains may or may not produce the best case (i.e. communication volume minimizing) for comparison. In such a case, to choose an optimal reference topology, we would need to exhaustively generate the topology space and select the topology that produces the closest possible P_x, P_y and P_z such that the total communication volume exchanges is minimal. Thus, choosing $N_x = N_y = N_z$ does not affect the generality, simplifies the discussion, saves exhaustive generation of topology space and allows the existing implementations of the function `MPI_DIMS_CREATE()` to return optimal volume minimizing domain partitions that serve as an optimal reference point for performance comparison purposes.

4.7.4 Discretization

We used the Finite Difference Method (FDM) to discretize the domain but the logic can be extended to the Finite Element Method (FEM) on structured grids. Like the FDM, FEM is also a technique to obtain the approximate solution of a Boundary Value Problem (BVP) but offers more flexibility as it is applicable to both unstructured and structured grids. The domain is first divided into a set of *finite elements*. Examples of finite elements in 2-D are a triangle and a quadrilateral. In 3-D they can be extended to tetrahedral and hexahedral elements. A *node* is a point on the finite element at which the value of the dependent variable is determined. The nodes that lie at the boundary of the element are called exterior nodes and can be used to connect the finite element to other elements. The nodes in the interior of the finite element cannot be used to connect it to other elements. The value of the dependent variable at the non-nodal points is approximated by *interpolation* of the nodal values. Assuming that a rectangular

```
new[i][j][k]=(1-alpha) * old[i][j][k] +
              alpha *
              (old[i-1][j][k]+old[i+1][j][k]+
               old[i][j-1][k]+old[i][j+1][k]+
               old[i][j][k-1]+old[i][j][k+1]);
```

Figure 4.31: Weighted Jacobi (ω-Jacobi) iteration kernel, `alpha`=constant, `new` and `old` are 3-D data arrays

finite element has only its four exterior nodes at the vertices and no internal nodes, the value at non-nodal points can be approximated as in Equation (4.9):

$$u(x,y) = N_1(x,y)u_1 + N_2(x,y)u_2 + N_3(x,y)u_3 + N_4(x,y)u_4. \tag{4.9}$$

In Equation (4.9), u_1, u_2, u_3 and u_4 represent the value of the dependent variable (field variable) at the nodal points (four vertices on a rectangular finite element) and N_1, N_2, N_3 and N_4 represent the *Interpolation/Shape/Blending* functions [28]. The shape functions are generally polynomials of independent variables and are predetermined. They must also satisfy certain conditions at the nodal points. In 2-D, using bilinear elements on a structured grid would result in a 9-pt stencil. This can be extended to a 3-D scheme where using a trilinear element results in a 27-pt stencil. We use a 27-pt stencil in Chapter 6 and show that the results are consistent with our predictions. Thus, we believe that our results can be extended to trilinear Finite Element discretizations on structured grids.

4.7.5 Iterative Methods

We used the unweighted Jacobi method for our discussion and derivation of the model for minimizing cache-misses. Nevertheless, our model is not restricted only to the unweighted method and can be applied to weighted Jacobi (ω-Jacobi), Gauss-Seidel (GS), Red-Black Gauss Seidel (RBGS) method, or the Successive Over-relaxation (SOR) [22]. The extension to the weighted Jacobi method is straightforward as the only extra term present in the weighted Jacobi iteration when compared to the unweighted Jacobi iteration is the weighted mesh point itself (i.e. the point at which the solution is being updated).

Figure 4.31 shows the ω-Jacobi update and contains an extra term `(1-alpha)*old[i][j][k]` as compared to Figure 4.10. Since the mesh vertex represented by `old[i][j][k]` is contained in the same cache-line as `old[i][j][k-1]` and `old[i][j][k+1]`, the number of data streams or the cache lines needed to contain the mesh vertices remain the same as in unweighted Jacobi updates. In Chapter 6, we use the ω-Jacobi updates for all our experiments and show that the results are in line with our inferences. The same can be applied to the Gauss-Seidel method which uses the latest values of the solution for updating the solution at a mesh point. Figure

```
new[i][j][k]=
              alpha *
              (new[i-1][j][k]+new[i+1][j][k]+
               new[i][j-1][k]+new[i][j+1][k]+
               new[i][j][k-1]+new[i][j][k+1]);
```

Figure 4.32: Gauss-Seidel iteration kernel, `alpha`=constant, `new` is a 3-D data array

4.32 shows the Gauss-Seidel iteration. The Gauss-Seidel algorithm differs from the unweighted Jacobi method in the sense that it uses a single array that contains the most recent values of the approximate solution. Thus, updating the solution at a mesh point (i, j, k) requires three updated solution values for points which are lexicographically before the point (i, j, k) and three old values of the points which fall lexicographically after the current mesh point. The big advantage of the Gauss-Seidel method is that it reduces the Working Set Size (WSS) and hence the memory traffic. Using the same argument, since the points are exactly the same as in unweighted Jacobi, the cache-miss equations remain the same.

The problem with the Gauss-Seidel method shown in Figure 4.32 is that the algorithm cannot be parallelized efficiently in the sense that simultaneous update of mesh points cannot be carried out by individual processes. This is because of the dependency on the updated values of the solution in the current iteration. Thus, the RBGS method is used, which divides the sub-domain into red and black points. The red points are dependent on the black points for their update and vice versa. Thus, each sub-domain is swept twice, once for the update of the red points and once for the update of black points. The communication/computation steps consist of first sending the red (or black) cells, updating the black cells (or red cells), and then sending the updated values of the black cells (or red cells) to update the red (or black) cells. Though the packing and unpacking cache-misses in this case increase, the model still yields the same equation structure as in Equation (4.1). Thus, the model can be extended to various iterative methods. The SOR method is a variation of the GS method and the model can be extended to it as well.

4.7.6 Stencil

We considered a 7-pt stencil for the derivation of our model but the same methodology can be applied to 19-pt or a 27-pt stencil in 3-D as well. In addition to the points considered by the 7-pt stencil, the 19-pt stencil also considers the neighbours corresponding to the middle points on the edges of the planes above and below the plane containing the point (i, j, k). Further, it also incorporates the diagonal mesh points in the plane containing the point (i, j, k). It should be noted that the addition of these points does not create a need for additional cache lines as these points are already contained in the cache lines being used for the 7-pt stencil. A similar

treatment can be applied to the 27-pt stencil which adds the corner points to the 19-pt stencil. We use the 27-pt stencil in the Restriction and Interpolation operator used in the Multigrid method [25] in Chapter 6 and further extend this discussion. The results in Chapter 6 again show that the 27-pt stencil exhibits the same behaviour as predicted by our model. These three stencils, namely, the 7-pt, 19-pt and the 27-pt stencil are the most commonly used stencils in 3-D structured stencil-based applications and our results may be applied to each of these.

4.7.7 Data Layout

Row-major (see Figure 4.8a) and Column-major (see Figure 4.8b) are the two types of data layout which are used for multidimensional arrays by common programming languages. We used the Row-major order to show the order of access in the Dependent Planes (DP) and the Independent Compute (IC) to model the cache-misses. The same can be applied to the Column-major order as well, with appropriate changes in the cache-miss equations. Thus, whereas in the Row-major order we had assumed the Z-direction to be contiguous, the X-direction is assumed to be contiguous in the Column-major order. This yields an equation similar to Equation (4.1) except that the Column-major equation is symmetric with respect to P_y and P_z but not with respect to P_x. Interchanging P_x and P_z will yield the total cache-misses equation for the Column-major form as shown in Equation (4.10):

$$S = 8P_yP_z + \frac{1}{2}P_xP_z + \frac{1}{2}P_xP_y = \alpha P_yP_z + \beta P_x(P_y + P_z). \tag{4.10}$$

We use the Column-major form when we use an Adaptive Mesh Refinement library called BoxLib [19], written in Fortran90 and C++ in Chapter 5 to show and verify that the inferences from the model still hold. We use only the Fortran90 version of the library and since Fortran90 supports a Column-major ordering, our results and discussions regarding Adaptive Mesh Refinement are Column-major oriented.

4.7.8 Data Type

The data type which we used throughout the current chapter is a *Single Precision* (SP) `float` data type. To extend the derivation of the model to a *Double Precision* (DP) data type, it is only the coefficients of the terms that need to change in Tables 4.3, 4.4, 4.2 and 4.6 that show the parameters for the X-plane, Y-plane, Z-plane and the total cache-misses, respectively. The `sizeof(float)` is 4 bytes whereas the `sizeof(double)` yields 8 bytes. Thus, 16 SP `float` values or 8 `double` values can be contained in a single cache-line. The only change to the parameters for Dependent Planes and the Independent Compute is that the denominator changes to 8 instead of 16 when using a `double` data type to model cache-miss equations. We use the `double` data type throughout Chapter 6 to show that the inferences from our model remain the same.

4.7.9 Sub-domains and MPI processes

In the derivation of the model we assumed that the sub-domain shape is completely determined by the domain size $N_x \times N_y \times N_z$ and the *Cartesian Topology* $D_x \times D_y \times D_z$. This implies that there is a single sub-domain per MPI process and that all sub-domains are homogeneous. This also implies a perfect load-balance as all the sub-domains have the same shape. These conditions help us to focus on the relative quality of the domain partitions without getting into the complexities of load imbalance. We do not attempt to generalize/extend the model to load-unbalanced scenarios although we do evaluate the quality of sub-domain shapes using the BoxLib library in Chapter 5 while letting the in-built algorithms in BoxLib balance the load. Having multiple sub-domains per MPI process complicates the communication pattern and does not make sense unless there is a restriction on the size of the sub-domain. We do not use any *threads* at the sub-domain levels as that gives rise to partitions at two levels: Domain Partitions dictated by MPI and a sub-domain level partition using some library of threads. Thus, for the purposes of simplicity and to limit the scope of the thesis, we do not attempt to extend the model for multiple sub-domains per MPI process, load-unbalanced scenarios or for a Hybrid program utilizing threads. A challenging natural extension to the work presented here will be to extend the model when using some from of Hybrid programming, e.g. MPI and OpenMP.

4.7.10 Overlapping Communication with Computation

One of our assumptions was that the computation is overlapped with communication. This is a standard practice in MPI and the non-blocking calls help to achieve it. It is because of overlapping that we separately update the Dependent Planes and the Independent Compute. When we remove the overlap, we do not separately update the DP and IC but update the whole sub-domain as a single update. Thus, the cache-misses which occur as part of the update of Dependent Planes become zero but the cache-misses for the update of the sub-domain increase to $\frac{5}{16}P_xP_yP_z$. The cache-misses for the packing and unpacking of planes remain the same and hence Table 4.6 can be reworked to yield Table 4.12.

Using Table 4.12, the total cache-misses attributed to the Dependent Planes can be written as Equation (4.11) below:

$$S = 2P_xP_y + \frac{1}{8}P_xP_z + \frac{1}{8}P_yP_z = \alpha P_xP_x + \beta P_z(P_x + P_y). \tag{4.11}$$

Thus, although the magnitude of coefficients decrease, the form of the expression remains equivalent to Equation (4.1). It is thus expected that one observes the same behaviour as the case of an overlap. We experiment with such a case using BoxLib in Chapter 5. By default BoxLib does not overlap communication with computation. Our results with single level uniform grids in BoxLib (see Chapter 5) show consistency with the results in this chapter,

Table 4.12: Non-overlapped cache-misses: Cache read/write misses for the X, Y and Z planes when computation is not overlapped with communication

Plane	**Pack read-misses**	**Unpack write-misses**	**Update read-misses**	**Update write-misses**	**Total**
Z-plane	P_xP_y	P_xP_y	0	0	$2P_xP_y$
X-plane	$\frac{P_yP_z}{16}$	$\frac{P_yP_z}{16}$	0	0	$\frac{P_yP_z}{8}$
Y-plane	$\frac{P_xP_z}{16}$	$\frac{P_xP_z}{16}$	0	0	$\frac{P_xP_z}{8}$

when communication is overlapped with computation.

4.8 Summary

The solution of a Partial Differential Equations (PDEs) can be numerically approximated after discretizing the domain with a scheme such as the Finite Difference Method (FDM), Finite Element Method (FEM) or the Finite Volume Method (FVM). After discretization, the numerical solution is obtained either by using an iterative method or a direct method. Parallel computing is frequently used to reduce the time to solution of the discretized PDE but introduces additional overheads in the form of local and global synchronization of processes. The first step in parallel computing is Domain Partitioning or domain decomposition - a fundamental step that divides the problem into sub-problems and maps the sub-problems to individual processes. The way domain partitioning is done can have a significant impact on the performance of the application. If the load is balanced, the orthodox approach to domain partitioning aims to minimize only the communication volume.

In this chapter, we challenge this orthodox approach of domain partitioning by creating a high level mathematical model that approximates cache-misses at the sub-domain level and identifies optimal domain partitions. To create this model we use the finite difference discretization of a linear, second order, constant coefficient Elliptic PDE and use the unweighted Jacobi iterative method to update the solution at a mesh point. The application of the unweighted Jacobi algorithm is similar to defining a Stencil - a fixed geometrical pattern that uses a weighted average of the solution at neighbouring mesh points to update the solution at a mesh point. It should be noted that a stencil accesses both contiguous and non-contiguous memory locations while updating the solution. Our model is *cache-aware* in the sense that it takes into account the cache-line size but the high level results are independent of the cache-line length, i.e. *cache-oblivious*. The simultaneous presence of cache-awareness and cache-obliviousness motivates us

to coin the term "*quasi-cache-aware*" to describe the model which at its core is solely based on the data-access pattern of stencil-codes.

The current chapter considers only scenarios where communication is overlapped with computation and hence the sub-domain is logically divided into two parts: the Independent Compute (IC) kernel that does not require data from other processes for updating the solution and the Dependent Planes (DP) that require data from other processes to buffer data into the ghost layers before the solution can be updated. More specifically, the model estimates the cache-misses incurred in the packing/unpacking/update of the DP and the update of the IC kernel. The cache-miss minimization condition from our model implies that the unit-stride dimension having contiguous data should be kept uncut and the sub-domain dimensions in the other two directions should be made equal for optimality. It is important to note that our model being high level does not take into account the problem size and architectural details to decide the domain partition and hence keeping the unit-stride dimension uncut does not always yield the optimal solution. As a contrast, the orthodox approach to domain partitioning i.e. minimizing the communication volume implies equalizing all sub-domain dimensions. Another limitation of the model is that it is applicable only on structured grids with cuts parallel to the Cartesian Axes. We also discuss the extension of the model to other PDE classes, Neumann boundaries, FEM discretization on structured meshes, Red-Black Gauss Seidel method, a Column major layout, data types and non-overlapping of communication with computation amongst others.

We substantiate our claims by comparing the performance of topologies obtained using our model and the partitions obtained using the default `MPI_DIMS_CREATE()` function of MPI that minimizes the communication volume. Our experiments demonstrate the efficacy of our model by solving the Laplace equation on a structured 3-D cubic domain using a finite difference discretization scheme employing a 7-pt stencil and the unweighted Jacobi iterative method. We also show that for the same sized X/Y/Z DP, the Z-plane which is orthogonal to the unit-stride dimension is the costliest plane to communicate in terms of cache-misses and time. The Weak Scaling results show that optimizing cache-misses are a much more significant factor than optimizing communication when the Working Set Size (WSS) does not fit into the cache-hierarchy. The results for Strong Scaling indicate that as the WSS reduces in size with increasing cores and starts fitting in the cache-hierarchy, cache-misses still play an important role in determining the optimal topology. The prime inference that emerges from our model and the experiments in this chapter is that the optimal partitions are "close to 2-D" rather than being cubic or near-to-cubic for stencil-based codes in 3-D.

Chapter 5

Adaptive Mesh Refinement

Discretized forms of Partial Differential Equations (PDEs) on uniform, structured 3-D meshes/grids, as mentioned previously, do not converge sufficiently fast enough for high resolution or large problem domains. Nonetheless, they form an integral part of multilevel stencil codes such as *Adaptive Mesh Refinement* (AMR) on structured grids, which is the subject of the current chapter. Adaptive Mesh Refinement locally refines a grid in an area of interest and thus creates a hierarchy of grid levels. We apply the cache-misses minimizing model that we developed in the previous chapter for solving an Elliptic PDE to a 3-D *block-structured* AMR code developed in a framework called *BoxLib* [4, 19]. This framework supports massively parallel multiphysics problems using a Fortran90 and C++ code-base. Initially we attempt to replicate our results for uniform single level meshes in BoxLib and then subsequently move onto the more complex multilevel AMR grid hierarchy where the load is typically not well balanced between cores.

5.1 Introduction

When PDEs are discretized and solved using iterative methods on a mesh, the accuracy of the solution is determined by the resolution of the mesh (i.e. mesh spacing). Specifically, to increase the accuracy of the approximated solution, the mesh spacing must be reduced. This leads to an increase in the number of degrees of freedom which naturally translates to an increase in the compute time. Further, it may be the case that, for a particular mesh, the error is not evenly distributed across the entire domain. Often, a few particular regions have a larger error than others. AMR [87–89] is a technique where the compute resources are directed towards obtaining an increased precision of the solution in particular regions of interest where the error is high. The regions of interest are dependent on the application and can, for example, be a spatial-region where the solution transitions rapidly. Thus, instead of approximating the solution on a globally refined grid, the solution can be obtained with less overall compute work by refining in local critical regions.

BoxLib [92] is a software library which may be used for developing parallel block structured AMR applications in two or three dimensions. BoxLib has been written with a combination of C++ and Fortran90. In addition, a pure Fortran90 version also exists. For simplicity and clarity, we only refer to the pure Fortran90 version of BoxLib in this thesis. However, alternative parallel AMR libraries do exist, *p4est* [137] and PARAMESH [1], for example. BoxLib abstracts away the complexity of communication/synchronization among processes, permitting the application developer to concentrate on complex multiscale multiphysics. Behind the scenes, BoxLib manages the parallel communication routines and the creation/destruction of grids. The basic abstraction BoxLib offers is the *box* - contiguous data representing the mesh points on a discretized domain. The choice of the box-size and shape impacts the application performance. Further, communication is not overlapped with computation in BoxLib. It offers support for solving *Elliptic, Parabolic* and *Hyperbolic* equations with *cell centered, face-centered* or *vertex centered* data.

In this chapter, we explore the impact of varying the box-sizes and shapes on application performance - ranging from a single level grid to a complex hierarchy of grids in AMR. We simulate and evaluate an *MPI Cartesian topology* for single grids and test our hypothesis, developed in the previous chapter, that minimization of communication is not the only governing factor necessary for optimal decomposition of stencil-based codes. That is, cache-misses for communication and compute must be taken into account for obtaining optimal domain partitions. We develop single grid and AMR codes in BoxLib using *Fortran90* and demonstrate that, contrary to the universally accepted strategy of minimizing communication volume to obtain optimal performance, non-cubic boxes can outperform the former for almost all processor counts and domain sizes on single grids. We further show that with a complex multilevel hierarchy in BoxLib, the non-cubic blocks can outperform the cubic block performance at only a certain core count and domain sizes. In the context of AMR codes, we discuss reasons for the partial effectiveness of our hypothesis. The load imbalance criterion, non-overlap of communication with computation, automatic distribution of boxes and metadata overhead emerge as the leading causes for the difficulties faced in verifying the hypothesis and establishing a consistent conclusion in case of AMR. Further, the effort spent in adapting the code to support non-cubic blocks for AMR in BoxLib is relatively high for the low performance gain obtained in specific cases.

The chapter begins by introducing the main concepts, key terms and describes the aim of the current work. After listing the contributions of the current work and motivation, we delve deeper into the description of AMR, the BoxLib library, our implementation of a MPI Cartesian topology, the AMR implementation specific to BoxLib and discuss results for both uniform and AMR test problems, before concluding the chapter with a high level summary.

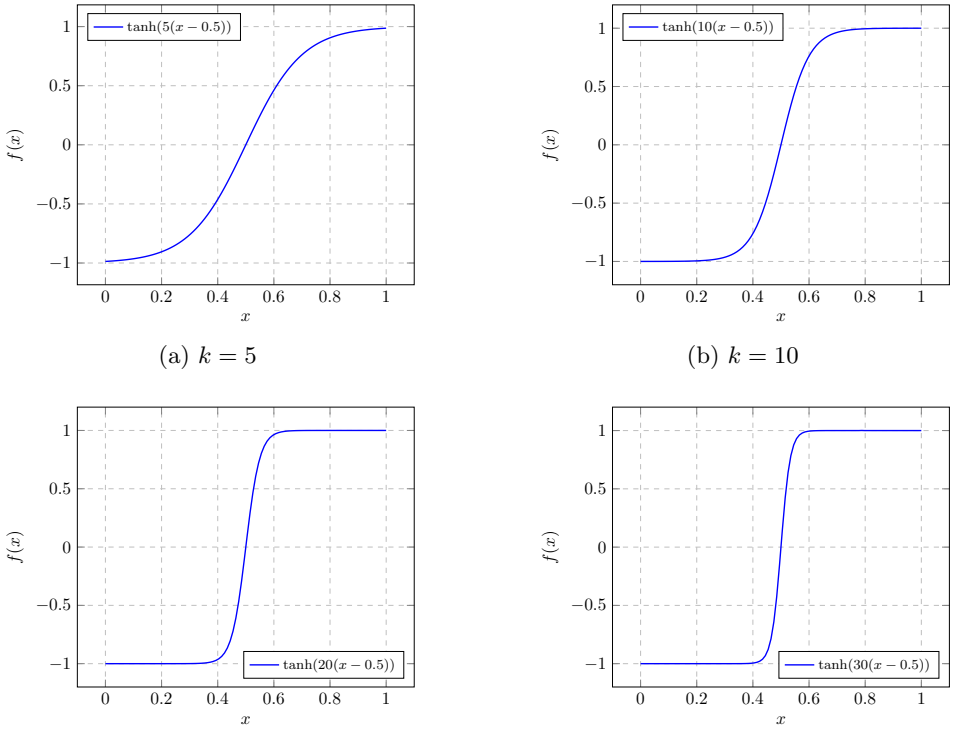
5.2 Motivation and Contribution

Adaptive Mesh Refinement is an integral technique of computational science and is also one of the key applications targeted for *Exascale* [20]. Researchers continuously search and explore optimization avenues for AMR frameworks, such as BoxLib, to enhance application scalability with the increasing complexity of problems and emergence of heterogeneous architectures. BoxLib has been shown to be effective on tens of thousands of cores however, though BoxLib is the basis of many mature applications, the literature [19, 99] assessing the performance of BoxLib codes is extremely scarce. This creates the motivation to explore selected performance aspects in BoxLib and test the expandability/applicability of our model from simple single grid applications to algorithmically and computationally complex AMR-based applications using BoxLib. With the plethora of AMR frameworks contributed by the scientific community, reinventing the wheel for such complex frameworks is inadvisable and hence the choice of using BoxLib for evaluating our ideas. The following are the contributions of this chapter:

- Implementation of a new layout simulating the MPI Cartesian Process Topology in BoxLib and its performance evaluation.
- Evaluate the hypothesis formulated in the previous chapter and [138], specifically for single grid codes in BoxLib in the absence of non-overlap of communication and computation.
- Investigate the performance impact of utilizing non-cubic boxes in Adaptive Mesh Refinement techniques and demonstrating that a communication minimization scheme does not always yield the optimal execution time.
- Implement, document and explain the changes to the library routines for seamlessly supporting use of non-cubic boxes.
- Explore and highlight the sources of inefficiency in BoxLib to formulate possible recommendations for its improvement. Though BoxLib stands deprecated now, its successor *AMReX* [98] is based on the same design principles as BoxLib. Our suggestions thus, hold for the latter framework as well.
- The current work can serve as a supplementary document to provide further insight into the working of BoxLib codes in addition to the BoxLib User guide [92].

5.3 AMR

Partial Differential Equations are numerically approximated by first discretizing over a domain and then solving them using direct or iterative methods on computer systems. The discretized



(a) $k = 5$ (b) $k = 10$

(c) $k = 20$ (d) $k = 30$

Figure 5.1: Plots for $y = \tanh(k(x - 0.5))$ on a domain [0,1] with $k = 5$, 10, 20 and 30

domain is first represented using a single grid/mesh. For obtaining an increased accuracy in particular regions, instead of refining the entire grid again, the region of interest is further refined according to some criteria. This criterion could be based upon an estimate of the accuracy of the approximated solution at specific points, the geometry of the domain or any other which makes the solution "interesting" in that particular region. Thus, in Adaptive Mesh Refinement [87–89], resources such as the compute power and memory are directed towards obtaining an increased precision of the solution in particular regions.

As an example, consider Figure 5.1 which plots the graph $y = \tanh(k(x - 0.5))$ for various values of k on a domain $x \in [0, 1]$. It can be seen that as the value of k increases, the transition of the solution between the extreme values of -1 and +1 occurs more rapidly. Thus, the region of interest in this case could be defined as the value of y for some $x = a < \frac{1}{2}$ and $x = b > \frac{1}{2}$. To obtain a higher precision to the approximated solution outside this interval (a, b) would be an inefficient use of compute resources and will also lead to an increase in the time to solution. Thus, we can selectively choose to refine the grid only in this region.

We can, for example, choose to refine the geometric region between $a = 0.2 < x < b = 0.8$

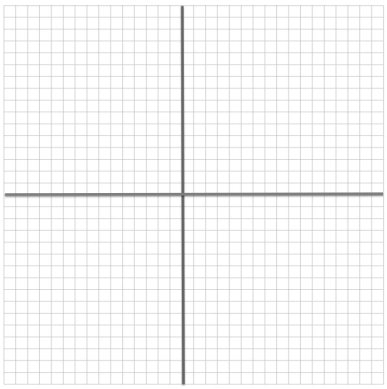
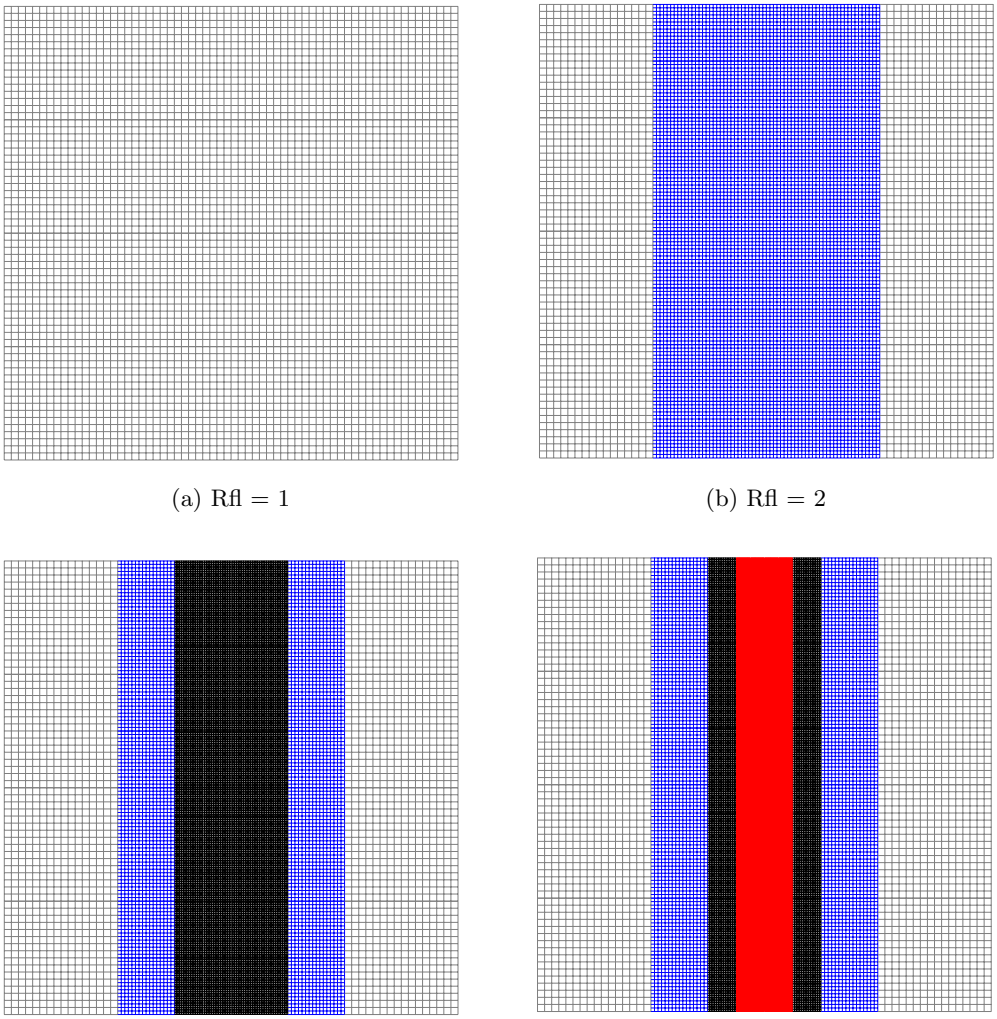


Figure 5.2: Domain [0,1] × [0,1] divided into 4 blocks having 16 × 16 cells each, grid spacing $h = \frac{1}{32}$

in Figure 5.1a and $a = 0.3 < x < b = 0.7$ in Figure 5.1b. The common characteristic between these is that the solution transitions more sharply as compared to the remaining domain in these regions and hence the need to obtain an increased precision. Instead of refining in a region where the gradient is high, we can choose to refine in a region where the curvature is high. For example, in Figure 5.1c, we can choose to refine in regions $0.38 < x < 0.42$ and $0.57 < x < 0.63$. Refining a subset of a grid also results in fewer degrees of freedom as compared to when refining the entire grid. Figure 5.2 shows a domain [0,1] × [0,1] represented as a mesh having 4 blocks with 16 × 16 cells each. This can typically be represented as a *Quadtree* in 2-D where each of the 4 blocks is represented as a child of the root and the root itself represents the entire domain. Since we consider only block-structured AMR, the refinement of even a single cell inside a block would result in the whole block being refined. Thus, it is possible that the whole mesh is refined even with block-structured AMR. This situation is shown in Figure 5.3a, which illustrates level 1 of refinement (the unrefined mesh being at level 0). When the refinement is applied again to the level 1 mesh, it produces the mesh shown in Figure 5.3b. It can be noted that now the entire mesh is not refined but only the region of highest error is refined. Subsequent refinements in the same geometrical area produce meshes shown in Figure 5.3c and Figure 5.3d. The cases in Figure 5.3b and 5.3c also form our test problems for AMR when we simulate the Elliptic PDE $\nabla^2 u = f$, and $u = \tanh(k(x - 0.5))$ represents its analytical solution.

In this thesis we use the convention that grid level zero ($l = 0$) represents the coarsest mesh i.e., the base mesh. The next refined level is represented by level one ($l = 1$) and so on. We only consider properly nested regions i.e. refined grids at level $l+1$ are completely nested inside grids at refinement level l. This condition must hold true except for at the domain boundaries. BoxLib also requires this condition to be true. An example of this is also shown in Figure 5.4, where the adaptively refined meshes result from refining a base mesh having 8 × 8 cells in each



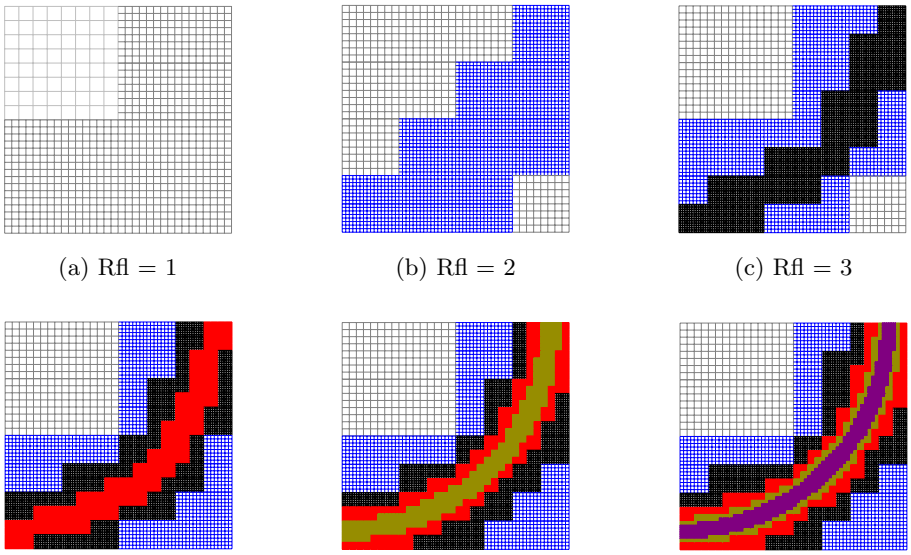
(a) Rfl = 1

(b) Rfl = 2

(c) Rfl = 3

(d) Rfl = 4

Figure 5.3: Refinement levels (Rfl) for obtaining increased precision for the PDE $\nabla^2 u = f$ having solution $u = \tanh(k(x - 0.5))$ by refining in the region $0.45 < x < 0.55$



(a) Rfl = 1 (b) Rfl = 2 (c) Rfl = 3

(d) Rfl = 4 (e) Rfl = 5 (f) Rfl = 6

Figure 5.4: Refinement levels (Rfl) for a mesh when the region $0.8 < x^2 + y^2 < 0.9$ is refined using blocks of size 8×8

block around the region $0.8 < x^2 + y^2 < 0.9$ (each block is refined if any of it overlaps this region). This can be visualized as the space region between two concentric circles in a quadrant of a unit square. As can be seen from Figure 5.4, grids $\mathcal{G}_{l+1}$ at level $l+1$ are contained within grids $\mathcal{G}_l$ at level l, where $0 \leq l \leq 5$. The aforementioned condition can be mathematically expressed as $\mathcal{G}_{l+1} \subset \mathcal{G}_l$. The combination of the coarse mesh representing the domain and the multiple levels of refinement may be referred to as a composite grid but the grids generally exist separately and the grid at level $l+1$ can be visualized as lying on top of the grid at level l. BoxLib, that we describe next, implements multiple levels of refinement in this way though it does not follow a Quadtree/Octree approach i.e. there is no direct parent-child relationship between grids at various levels.

5.4 Introduction to BoxLib

The most basic constituent element/abstraction in BoxLib is the *Fab* (FArray Box) which represents a set of contiguous data on a Box. A Box is a data structure for representing a rectangular domain on an index space and does not contain any data. Internally, the data associated with a Box is allocated using the `new` operator of C++ and can be mapped to, say, a 3-D shape by using the specification in the Fab object. Thus, a *grid* (a rectangular region in an index space) at any level is equivalent to a single Fab object [99]. The collection of all the `Fab` objects (defined as a `struct`) at a particular level is referred to as the *MultiFab*. The grids at any level are non-intersecting but the Fab objects may be defined on a Box larger than the grid if *ghost*

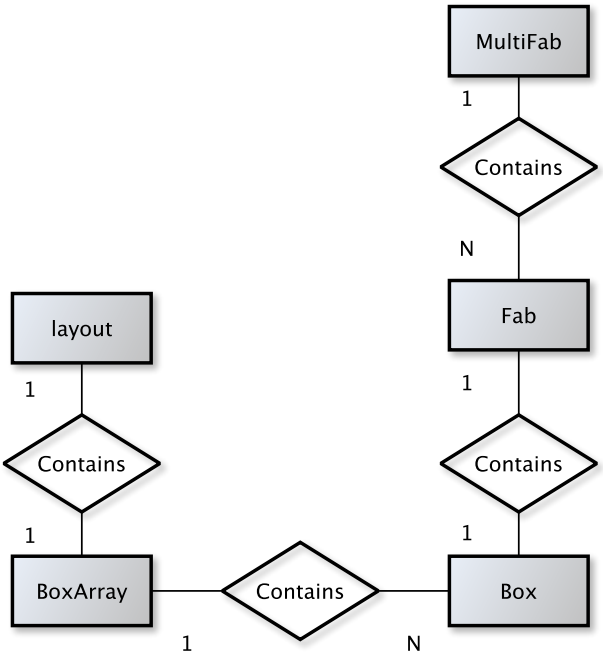
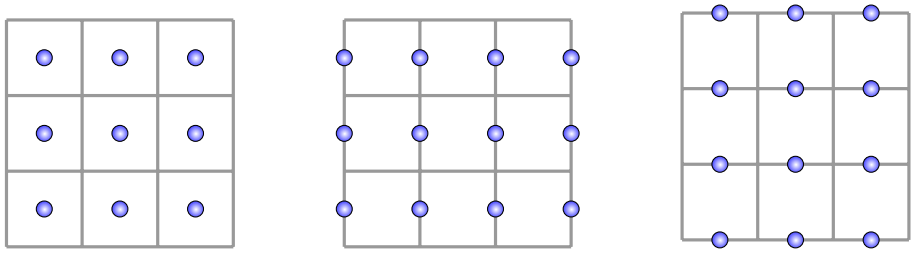


Figure 5.5: Relationship between a Box, Fab, BoxArray, layout and MultiFab. The labels 1 and N are the cardinality of the relationship named "Contains".

points are included. It is the Fab objects that are distributed among cores and are acted upon independently by the cores. In AMR, when the number of levels is greater than or equal to three, BoxLib requires and ensures proper nesting i.e. level $l+1$ grids must be fully contained in level l grids (except at the physical boundaries) [92]. This is because BoxLib has been written in such a way that it requires a balance ratio of 2:1 i.e. the difference in the level of any two adjacent cells cannot be more than one. The position of a grid or *cell* is with respect to a global mesh index which covers the entire domain at that level. Thus, if a 32×32 grid covers the entire domain at level 0, the index space range is $(0,0) - (31,31)$. If this grid is refined using a refinement ratio of 2 i.e. 64×64 cells cover the entire domain at level 1, the index space range is $(0,0) - (63,63)$ at level 1. A *BoxArray* is an array of boxes, implying that it is an ordered collection and not just a set. A *layout* is an enhanced BoxArray, which among many other fields, contains the information as to which box is assigned to which core or MPI rank (using the 1-D `prc(:)` array defined by BoxLib). A MultiFab contains all the Fabs at a particular region of refinement. The relationship between a Box, Fab, BoxArray, layout and MultiFab is depicted in the *Entity Relationship* diagram in Figure 5.5 where the relationship *cardinality* of 1:1 or 1:N is shown as the labels on the connecting edges.

BoxLib supports cell-centered and *nodal* data in single/multiple directions (see Figure 5.6).



(a) Cell centered (b) Nodal X-dir (c) Nodal Y-dir

Figure 5.6: Cell centered and nodal data in BoxLib

Assuming that the X-axis runs from left to right and the Y-axis runs from bottom to top, Figure 5.6b shows data that is nodal in the X-direction whereas Figure 5.6c depicts data that is nodal in the Y-direction. Both these configurations can be obtained from cell-centered data shown in Figure 5.6a by shifting the central circle representing data in the -ve X or Y direction, respectively. Data in 2-D which is nodal both in the X and Y direction creates vertex centered data. A similar analogy can be derived for 3-D data. We only use cell-centered data in our experiments, however.

Initially a Box is created to represent the entire domain. This Box can be split up into multiple small boxes to be given to various cores according to a data distribution algorithm. The boxes are generally all squares in 2-D and cubes in 3-D. Two data distribution schemes, namely the *Knapsack*, to equalize load distribution, and *Morton Space Filling Curve* (SFC), to optimize communication, are provided as part of the software. A dynamic decision, depending on the number and volume of grids is taken on whether to select the Knapsack or Morton Space Filling Curve data distribution. Each process contains enough metadata to locate the index space region of each box on every level so that it knows which processor core or MPI rank contains which box. The disadvantage is that as the number of boxes grow, this metadata also grows in size. Hybrid MPI i.e. a combination of MPI and (typically) OpenMP [50] is used to obtain larger grids which naturally reduces the total metadata. BoxLib is the basis of several mature applications such as MAESTRO [96] (low Mach number code), CASTRO [97] (compressible Astrophysics) and LMC [139] (Combustion code) which scale well but are limited by the high communication-intensive linear solves. The library can be downloaded for development at [4].

5.5 Box Distribution

The default algorithm for distributing boxes in BoxLib is the Knapsack algorithm. This comes into effect by setting `def_mapping = LA_KNAPSACK` in the source file `layout.f90`. The same

can also be achieved by invoking the `layout_set_mapping()` subroutine in the user created `main.f90` with `call layout_set_mapping(LA_KNAPSACK)` as the first statement after initializing BoxLib. When the number of boxes is equal to the number of cores, the boxes are given sequentially to ranks in the increasing order. If the number of boxes is less than the number of cores, the boxes are again given to ranks in increasing order. The remaining cores in the latter scenario do not perform any compute work but it may be pointed out that this leads to an inefficient use of compute resources.

5.5.1 Fab Numbering and Process Numbering

As mentioned in Section 5.4, it is the Fab data structure which represents boxes and is distributed among processes. Assume that a 2-D domain is composed of Fabs as shown in Figure 5.7a, then the numbers given to the Fabs are shown inside circles. This scheme of numbering is analogous to the *column-major* order for data layout in Fortran. If these six Fabs are to be given to each process assuming a Cartesian MPI topology then the ranks of the processes to which these boxes are given are shown in Figure 5.7b. The ranks given to processes follows the *row-major* data layout such as in the C language.

(a) Fab numbering in 2-D (b) MPI rank numbering in 2-D

Figure 5.7: Fabs and MPI Cartesian Topology Rank numbering in 2-D

The same concept applies in 3-D where Fabs (or boxes) are numbered from bottom to top, then left to right and then again in the same order but starting from the next Z-plane (front to back). It can be visualized as a series of 2-D plates arranged next to each other. The process numbering in a 3-D MPI Cartesian Topology first spans the Z-dimension, then follows a left to right direction and finally the downward direction. BoxLib internally maintains a one dimensional integer array called the `prc` array which is a mapping from the box numbers to the MPI process ranks. As an example, if there are 16 boxes, the `prc` array will have a length of 16 and if `prc(7) = 10` then the 7^{th} box or Fab is given to process having the MPI rank 10. Further, each process maintains a copy of this array, and there exists a separate `prc` array for each level of AMR. In short, we can write `prc(box_number) = MPI_Rank`.

5.5.2 Implementing an MPI Cartesian Topology

The original definition of the subroutine `layout_set_mapping()` defined in `layout.f90` contains only an integer constant representing the box distribution scheme. To implement an MPI

Cartesian Topology, we extend the parameter list in this subroutine to contain the X, Y, and Z integer process dimensions, namely `xdim`, `ydim` and `zdim`, respectively, in 3-D and thus, its new signature is shown in Listing 5.1.

```
1 subroutine layout_set_mapping(mapping,xdim,ydim,zdim)
```

Listing 5.1: Changed signature of the layout subroutine in layout.f90

```
1 call layout_set_mapping(LA_DD,xdim,ydim,zdim)
```

Listing 5.2: Invoking layout subroutine with our integer constant LA_DD representing MPI Cartesian Domain Decomposition

Immediately after initializing BoxLib in the application, this subroutine can be called as shown in Listing 5.2, where `LA_DD` is the named constant representing our MPI Cartesian Topology *D*omain *D*ecomposition. The arguments `xdim, ydim` and `zdim` are passed to variables named `D_x, D_y`, and `D_z` (visible to all subroutines) in the file `layout.f90`. Further, we allocate a *rank array* named `rank_array` that defines which Box (or Fab) is given to which core. We use this rank array to fill the BoxLib defined 1-D array `prc` which performs the mapping of boxes to cores.

The subroutine for our domain decomposition is shown below (defined in the file `layout.f90`) in Listing 5.3. This subroutine only handles situations in which the number of boxes (or Fabs) is equal to the number of processes. First, an integer array `rank_array(D_x,D_y,D_z)` is allocated which contains the MPI ranks of processes corresponding to boxes. The ranks are filled in the same order as the coordinates used in MPI process decomposition. Next, the pre-defined `prc(1:nboxes)` 1-D array is filled with the MPI rank for each box. Since boxes are numbered in Fortran order, the order of loops is important.

```
1  subroutine layout_dd(prc)
2   integer, intent(out), dimension(:) :: prc
3   integer :: i,j,k,ctr
4
5   allocate(rank_array(D_x,D_y,D_z))
6    ctr=0
7   do i=1,D_x
8           do j=1,D_y
9                 do k=1,D_z
10                      rank_array(i,j,k)=ctr
11                      ctr=ctr + 1
12                end do
13          end do
14   end do
15
16  ! Fill prc(:) - start bottom left-> up->next column -> next 2-D slab in Z-dimension
```

```
17 ! ctr is index into prc(:)
18
19 ctr=1
20 do k=1,D_z
21       do j=1,D_y
22             do i=D_x,1,-1
23                   prc(ctr)=rank_array(i,j,k)
24                   ctr=ctr + 1
25             end do
26       end do
27 end do
28 deallocate(rank_array)
29 end subroutine layout_dd
```

Listing 5.3: 3-D MPI Cartesian Topology

As an example, if the Cartesian Topology is $2 \times 3 \times 4$ for 24 processes and 24 boxes, then box 1 is allocated to rank 12, box 2 is given to rank 0, box 3 is given to rank 16 and so on.

5.5.3 Multiple boxes on a single core

It is possible to have multiple boxes per-core i.e. each sub-domain per core consists of multiple boxes. Assume a 2-D domain with `n_cells = 16` (a 16×16 domain), 4 processes decomposed as $D_x \times D_y = 2 \times 2$, and a box size of 4×4. Thus, there are $\frac{16}{4} \times \frac{16}{4}$ boxes in all (boxes in the X, Y direction are denoted by $N_x = 4$ and $N_y = 4$, respectively). The number of boxes for each process is given by $S_x \times S_y = \frac{N_x}{D_x} \times \frac{N_y}{D_y}$ i.e. $\frac{4}{2} \times \frac{4}{2} = 2 \times 2 = 4$. This is shown in Figure 5.8. Then according to Fab or Box numbering in BoxLib, boxes 3, 4, 7, 8 are assigned to rank 0, boxes 11, 12, 15, 16 are assigned to rank 1, boxes 1, 2, 5, 6 are assigned to rank 2 and boxes 9, 10, 13, 14 are assigned to rank 3. This is in accordance with the MPI process numbering in 2-D (or 3-D when appropriate). Listing 5.4 shows how an MPI Cartesian topology can be implemented when multiple boxes per sub-domain are allowed. The code is similar to Listing 5.3 except for now the inner loops can account for multiple boxes being allocated to the same rank. We

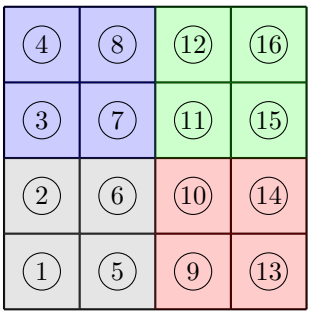


Figure 5.8: 16 Fabs (or boxes) spread on 4 processes arranged as 2×2. Each color shows a single MPI process and numbers inside circles show the Fab number

denote this scheme of decomposition by a constant `LA_DD_MB` (*LA*yout *D*omain *D*ecomposition *M*ultiple *B*oxes) in the file `layout.f90`. In Listing 5.4, `N_x, N_y, N_z` denote the total boxes and `S_x, S_y, S_z` denote the boxes per core in the X, Y and Z direction, respectively. First, the 3-D array `Bx3d` is filled with MPI ranks with each MPI rank being repeated multiple times corresponding to the multiple boxes which are to be given to a particular MPI rank. This is accomplished by the first loop-nest consisting of six `do` loops. The next step is to fill up the `prc` array by traversing in the Fab numbering order and use ranks in the `Bx3d` array to assign an MPI rank to each box.

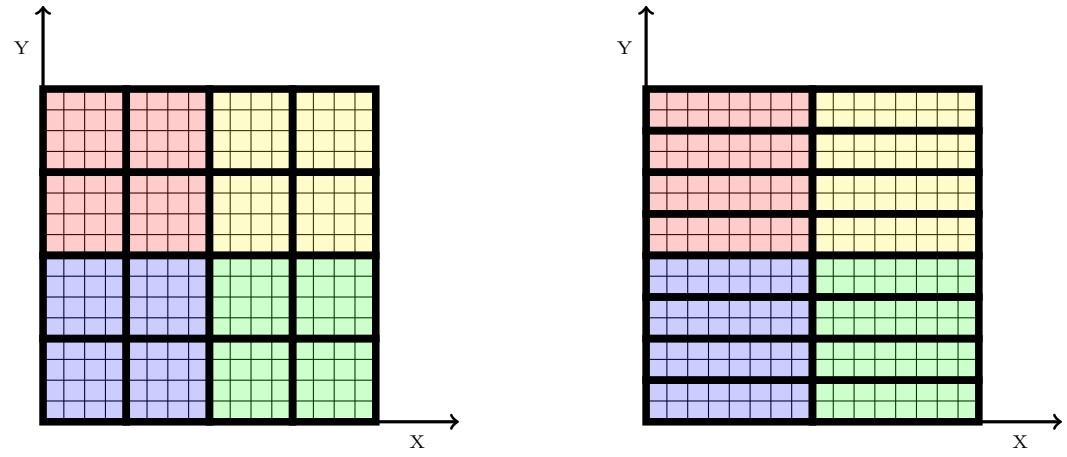
```
1  subroutine layout_dd_mb(prc)
2
3          integer, intent(out), dimension(:) :: prc
4          integer :: i,j,k,ctr,S_x,S_y,S_z
5          integer :: ii,jj,kk
6
7          S_x = N_x/D_x
8          S_y = N_y/D_y
9          S_z = N_z/D_z
10         allocate(Bx3d(N_x,N_y,N_z))
11
12         ctr = 0    ! Denotes MPI rank right now, multiple boxes can have same MPI rank now
13
14         do i = 1, N_x, S_x
15                 do j = 1, N_y, S_y
16                         do k = 1, N_z, S_z
17                                 do ii = i, i + S_x - 1
18                                         do jj = j, j + S_y - 1
19                                                 do kk = k, k + S_z - 1
20                                                         Bx3d(ii,jj,kk) = ctr
21                                                 end do
22                                         end do
23                                 end do
24                                 ctr = ctr + 1
25                         end do
26                 end do
27         end do
28
29         ctr = 1                                    ! Now this denotes the box or Fab number
30
31         do k = 1, N_z
32                 do j = 1, N_y
33                         do i = N_x, 1, -1
34                                 prc(ctr) = Bx3d(i,j,k)
35                                 ctr = ctr + 1
36                         end do
37                 end do
38         end do
39
```

```
40          deallocate(Bx3d)
41
42  end subroutine layout_dd_mb
43
```

Listing 5.4: MPI Cartesian topology when multiple boxes are assigned to a core

5.5.4 Varying shape of box within sub-domain

When there is a single box per core, then the sub-domain is the same as that box. Here the shape of the box (or sub-domain) is completely defined by the domain decomposition. When there are multiple boxes per core, the domain decomposition only determines the sub-domain shape (which in turn consists of multiple boxes). In the example in the previous subsection, the sub-domains had boxes of size 4×4. But we could alternatively have boxes of size 2×8 or 8×2. In BoxLib it is not possible to first divide the domain into a sub-domain and then divide the sub-domain into boxes. We have to start by specifying the box-size initially. Thus, the process can be thought of as specifying the box-size first, then specifying the domain decomposition to create sub-domains of a specific box size, i.e. we need to follow a bottom-up approach as opposed to a top-down approach. Figure 5.9a shows a 16×16 domain divided among 4 cores arranged as 2×2 and each sub-domain having 4 boxes each of size 4×4. Figure 5.9b shows a 16×16 domain divided among 4 cores arranged as 2×2 and each sub-domain having 4 boxes each but with a size of 2×8.



(a) Box size = 4×4 (b) Box size = 2×8

Figure 5.9: Varying box sizes with Domain = 16×16, 4 processes (arranged as 2×2), and 4 boxes per sub-domain

5.6 AMR in BoxLib

BoxLib creates a MultiFab for each level as part of the AMR hierarchy. The boxes at each level are distributed independently of the other levels while using the Knapsack/Morton order algorithm. Further, the grid portions which are refined are not destroyed i.e. if a particular part of level l grid is refined resulting in a grid at level $l+1$, then this portion of grid at level l is not destroyed. It provides routines to fill the ghost cells at the same level, to carry out *interpolation* and *restriction* between levels, and several other functionalities. The interpolation routine transfers data from the coarse grid to the fine grid whereas the restriction subroutine transfers it in the opposite direction. BoxLib ensures proper nesting when the number of levels is greater than or equal to three because two levels are always properly nested. When the number of levels is 3, there exists a possibility that a level 3 cell might be an immediate neighbour of a cell at level 1 and this violates the proper nesting condition. Thus, such a condition necessitates further refinement of the grid which is at level 1. This (necessary) condition is sometimes also called the *2:1 balance*. We use the terminology *active box* to represent a box which has not been refined. An *inactive* box is a box at level l which has been refined to create a grid at level $l+1$. The composite grid then can then be visualized to be made of the union of all the active boxes at all levels of refinement.

5.6.1 Note on various control parameters

There are several parameters in BoxLib that control the box-size and the refinement criteria. The `max_grid_size`, which represents the box-size, must be ≥ 1, i.e. the minimum box-size must be 2×2 in 2-D or $2 \times 2 \times 2$ in 3-D. Further, it is not necessary that if we fix the box-size, BoxLib will maintain the box-size while refining, as it is controlled by the factors that we describe below. There are four factors which affect the refinement procedure. We describe their role in a 2-D setting but the explanation can be extended to a 3-D domain.

1. `amr_buf_width`: Cells for refinement are tagged according to the tagging criteria in `tag_boxes.f90` but additional cells can also be tagged according to this factor. This factor sets the radius of the cells which are tagged in all directions i.e. in addition to the cells marked for refinement, all directions N, E, W, S, NE, NW, SE, and SW will have `amr_buf_width` additional cells, where N stands for North, S for South, E for East and W for West.

2. `cluster_minwidth`: Any newly created grid should at least have these number of cells in each direction i.e. `cluster_minwidth` $\times$ `cluster_minwidth` cells in 2-D and a corresponding number in 3-D in each direction. This may or may not hold true in the case of non-cubic blocks but always holds true for cubic blocks. In BoxLib, this factor is defined as a scalar and poses problems when dealing with non-cubic blocks. Ideally, this should be defined as a vector to implement the possibility of having different numbers of cells in

each dimension/direction.

3. `cluster_blocking_factor`: The number of cells in each direction of a newly created grid must be divisible by this factor.

4. `cluster_min_eff`: This is a number in the range $[0, 1]$ and denotes the minimum fraction of tagged cells needed in a block at which the entire block is refined. When the value of this factor is one, only the cells marked for refinement according to the user-defined criterion are refined. When `cluster_min_eff = 0`, even tagging one cell should/will tag the entire block. As an example, if `cluster_min_eff=0.25`, then only one user-defined tagged cell is needed to refine the entire block if the block size is 2×2 in 2-D (since 25% of 4 cells is a single cell). In 3-D, for a $2 \times 2 \times 2$ block and `cluster_min_eff=0.25`, at least two user-defined tagged cells would be needed to refine the entire block.

5.7 Test Problems

We now describe the 3-D test problems that we use for evaluating and testing the extension of our model developed in the context of uniform single grids. Since there is no overlap of communication with computation in BoxLib, the cache-miss model due to separation of data into an *Independent Compute* zone and the *Dependent Planes* zone is not directly applicable for codes written in the BoxLib library. Thus, the major contributors of cache-misses is the contiguous compute zone and the packing/unpacking of planes of data. In the following discussion we therefore, do not use the terms Independent Compute or Dependent Planes.

To evaluate the efficacy of our model on single grids in BoxLib, we implement a cell-centered, *Finite Difference* scheme to solve the *Laplace* equation $\nabla^2 u = 0$ on a unit cube with *Dirichlet* boundaries. The *unweighted Jacobi* method is used to update the solution at mesh points.

In the case of adaptively refined meshes, we solve an Elliptic PDE,

$$-\nabla^2 u = 2k \tanh(k(x - 0.5))(k - k \tanh(k(x - 0.5)) \tanh(k(x - 0.5))),$$

having the solution $u = \tanh(k(x - 0.5))$, using a cell-centered, Finite Difference scheme on a unit cube with Dirichlet boundaries. The unweighted Jacobi iterative method is used to update the solution. The parameter k is chosen as 10 for the test problem.

Since the Dirichlet boundaries in a cell-centered scheme do not coincide with the actual boundaries, they are updated by equating the average of the ghost cell (u_g) representing the boundary and the next-to-boundary (u_0) internal cell values to the actual boundary condition (u_a) at the domain boundary after each iteration. Thus, $\frac{u_g + u_0}{2} = u_a$, implying $u_g = 2 \times u_a - u_0$. The refinement criterion for the first level is that the y-coordinate distance should lie between

0.35 and 0.65 i.e. $0.35 < y < 0.65$. Whenever any cell is tagged for refinement in a block, the entire block is refined (pure block-structured AMR). The refinement criterion for the second level changes the values of the y-coordinate to $0.455 < y < 0.545$.

5.8 AMR Implementation

The program can be divided into two logical parts: Set-up and Solve. Though the Set-up phase appears to be more complex than the Solve phase, the latter is responsible for a very high fraction of the total execution time and thus in experiments we concentrate on the execution time of only the Solve phase. The discussion which follows considers only a 3-D implementation.

5.8.1 Set-up

After initializing BoxLib with `boxlib_initialize()`, we need to specify the domain size (`n_cell`), the maximum number of AMR levels allowed (`max_levs`) and the dimension of the problem (`dim`). We impose that an entire box is refined even if there is a single tagged cell in it (i.e. true block structured AMR behaviour). Further, there is no *safe* layer i.e. if a block is refined, no cells outside the block are tagged for refinement. Thus, `amr_buf_width` is set to 0. For a cubic block, the `max_grid_size` variable can be set such that the box has equal cells in all directions and `n_cell` is generally perfectly divisible by `max_grid_size`. If the box-shape is non-cubic, then the sizes are specified in a 1-D array having three elements namely, `chunk_dims(1:3)`. Thus, for a cubic block, `chunk_dims(1:3)=max_grid_size`. The `cluster_min_eff` indicates the minimum fraction of cells which must be refined so that the entire block is refined. Thus, since the total cells in a box are `chunk_dims(1) * chunk_dims(2) * chunk_dims(3)`, the `cluster_min_eff` is set to

$$\frac{1.0}{chunk\_dims(1) * chunk\_dims(2) * chunk\_dims(3)}.$$

Setting the value of `cluster_min_eff` in such a way should have the same effect as tagging all the cells in a block in the file `tag_boxes.f90`. We stress that it is important to experiment with both these factors and verify the result after plotting and visualizing the domain with a visualization package such as *VisIt* [140]. The BoxLib user manual [92] defines the `cluster_minwidth` as the minimum number of cells in each direction of the newly formed grid. This is defined as a scalar variable in BoxLib. Thus, the implementation and the User Guide assumes that the user generally uses cubic-blocks to minimize communication and hence the `cluster_minwidth` remains the same in all directions. This is not the case with non-cubic blocks and hence ideally the `cluster_minwidth` should be a `dim`-dimensional vector depending on the dimensionality (`dim`) of the problem. Our correspondence with the developers of BoxLib made it clear that they might make the `cluster_minwidth` a vector in the future implementations but it was not a priority as most users used cubic-block sizes. Thus, for

Table 5.1: Set-up Variables: Declared variables during Set-up phase

Variable	Meaning
lo(3), hi(3)	Low, High end of box
is_periodic(3)	Periodicity in each direction
prob_lo(3), prob_hi(3)	Physical domain of problem
phys_bc(dim,2)	Physical boundary types
dx(max_levs)	Mesh spacing at each level
phi(max_levs), oldphi(max_levs)	Jacobi update MultiFabs
rhs(max_levs)	RHS array
la_array(max_levs)	Layout array at each level
error(max_levs)	Error MultiFab at each level

our experiments when $chunk\_dims(1) \neq chunk\_dims(2) \neq chunk\_dims(3)$ and the value of `chunk_dims(2)` lies between `chunk_dims(1)` and `chunk_dims(3)`, we set the `cluster_minwidth` to `2 * chunk_dims(2)`. The result was always verified by observing the resulting box-shapes with VisIt and the chosen non-cubic block shape was consistently observed at all levels of refinement. The `cluster_blocking_factor` gives a value such that all newly formed grid dimensions in each direction are divisible by `cluster_blocking_factor`. This was set equal to the `cluster_minwidth` after experimentation. To summarize, it requires experimentation and visualization to determine the appropriate values of the factors `cluster_min_width` and `cluster_blocking_factor` to ensure that the non-cubic block size is maintained at all levels of refinement.

The boundaries in our experiments are all Dirichlet boundaries and their values can be set in the file `bc.f90`. Further, the Dirichlet boundary is represented by the integer `15` by BoxLib. 3-D arrays (or MultiFabs) are allocated to represent various conditions such as the low (minimum indices of coordinates) and high end (maximum index of coordinates) of a box, its physical dimensions etc., as shown in Table 5.1. The physical domain of our test problem (`prob_lo(3), prob_hi(3)`) ranges from 0 to 1 in each direction (unit cube). The mesh spacing for the coarsest grid is calculated as $\frac{prob\_hi(1)-prob\_lo(1)}{n\_cell}$. Since we use a unit cube domain, the direction we use to calculate the mesh spacing does not matter.

Built-in subroutines are called to set the `cluster_minwidth, cluster_blocking_factor` and the `cluster_min_eff`. The refinement ratio between levels is set to two by using the `amr_ref_ratio_init(max_levs,dim,2)` subroutine call. Since AMR has multiple levels, instead of building a single BoxArray object, a multilevel BoxArray object is built using the subroutine call `ml_boxarray_build_n(mba,max_levs,dim)`, where `mba` is of type `ml_boxarray` i.e. a Multilevel BoxArray. The refinement ratio must be passed to the `mba` variable using `ml_boxarray_set_ref_ratio(mba)` and this uses the ratio set by the `amr_ref_ratio_init` call above. The default ratio used by BoxLib is two, i.e. the resolution/mesh-spacing of the grid at

level 2 is half that of the mesh at level 1. In general, the mesh spacing at level $l+1$ is half that of at level l. The subroutine `bc_tower_init()` allocates the array defined in `define_bc_tower.f90` but does not initialize it. In general, the `bc_tower` datatype defines the boundary conditions at each level. Thus, we must pass the `max_levs, dim` and `phys_bc` to the initializing subroutine.

The Multilevel BoxArray object `mba` has a field named `pd` which must point to the Box at a particular level. Thus, we set `mba%pd(1)=bx`, where the variable `bx` represents the box spanning the initial given domain. This is followed by setting the `mba%pd(level)` to appropriate boxes that are obtained by refining the box at level 1. Another field in the `mba` object gives the size of the boxes at a particular level and is given by `mba%bas(level)`. Since initially the whole domain is represented as a single box at level 1, we execute `boxarray_build_bx(mba%bas(1),bx)` to internally set `mba%bas(1)%bxs(1)=bx`. The call `boxarray_maxsize(mba%bas(1),chunk_dims)` breaks the box at level one into boxes having sizes `chunk_dims(1:3)`.

The first level layout can now be built using the `layout_build()` subroutine by passing the multilevel box array pointer, the problem domain and the periodicity of the problem. The boundary conditions tower can be built next using the call to `bc_tower_level_build()` and passing into it the layout array which was built in the previous step. Next all the MultiFabs are allocated memory and the initial solution/guess is initialized to zero. MultiFabs can be passed into user-defined low level subroutines which are accessed as either 2-D or 3-D arrays, element by element in the column major order.

When statically refining the grid, as in our test problem, the refinement criteria is tested to see if the base level grids/ coarsest grids need to be refined. If this is the case then they are refined and the new grids are built by invoking the `make_new_grids()` subroutine. This is carried out along with allocating new MultiFabs for the newly created levels and addition of the boundary conditions to that level using the `bc_tower_level_build()` subroutine. The MultiFabs are rebuilt again if the proper nesting conditions are violated and this nesting is forced using the `enforce_proper_nesting()` subroutine. Since the number of levels now may be less than the `max_levs` specified earlier, the restricted layout is built using `ml_layout_restricted_build()`. The Set-up phase is not trivial, especially when the proper nesting condition is violated which results in re-building of data structures. This leads us to the next phase where the solution is approximated at each iteration i.e. the Solve phase.

5.8.2 Solve

The convergence criterion of our test problem is based on the norm of the actual error i.e. l_2 norm of the difference between the actual and the approximate solution for the mesh points constituting all the active boxes. Initially, a separate MultiFab called the error MultiFab, i.e. `error(i)` for level i, is initialized. It is done such that the active boxes on any level are initial-

ized with the value of the solution but the inactive boxes at any level are initialized to a value of zero. An inactive box at any level is initialized as zero because this box has further been refined and will not be updated at this level. It can be noted that changing the convergence criterion to l_2 norm of the *residual* should not change the performance results in any way. At the beginning of the solve phase, the solution MultiFab (`phi(i)` and `oldphi(i)`) are initialized to zero and the error at each process is calculated i.e. the square of the difference between the actual and the approximate solution (`error(i)-phi(i)`)^2. The local sums are then added using `parallel_reduce` at the root processes, followed by taking the square root of this sum. This gives us our initial l_2 norm of the error (`r0_global`). After every update of the composite grid, the norm of the error is calculated using the same procedure which constitutes the error norm at the k^{th} update (i.e. `rk_global`). The update of the grid is carried out while the ratio of the norm of the error at the k^{th} step to the initial norm remains more than a specified user tolerance i.e. $\frac{rk\_global}{r0\_global} > TOL$.

5.8.2.1 Solution update

Each level is composed of a MultiFab, which in turn is composed of several Fabs (or boxes). First, the coarsest grid is updated by updating one Fab at a time. If the Fab has been refined further, it is an inactive Fab and is not updated. For updating, as mentioned, we use the unweighted Jacobi iterative method with a 7-pt stencil. Thus, a new value of the solution MultiFab (`phi`) is calculated using the old solution MultiFab (`oldphi`) and the RHS array (`rhs`). Since the scheme is cell-centered, we update the Dirichlet boundaries as explained in the beginning of this section i.e. the average of the fictitious ghost layer point representing the boundary and the next-to-boundary point is equated with the value of the point at the actual boundary. This procedure is carried out using the subroutine `multifab_physbc()`.

5.8.2.2 Interpolation

The fine grid cells at the next level need the values of the coarse grid cells at the *coarse-fine* grid boundaries and thus, the updated values at the coarse grid points are interpolated using the `multifab_fill_ghost()` subroutine - a built-in subroutine that abstracts away the details of implementation from the user. This subroutine does not affect the physical boundaries at any level but only the internal coarse-fine interface cells. This procedure is carried out for all levels except for the finest grid. The interpolation represents a flow of data between two adjacent grid levels i.e. from level l to level $l + 1$.

5.8.2.3 Restriction

Since the solution is more precise at the finer levels, the values are then restricted back to the coarse grid cells for all levels using the routine `ml_cc_restriction()`. This is done after the

new norm of the error has been calculated. The restricted values being transferred from level $l+1$ to level l will map to the inactive box at level l but will be used by cells at level l which have a common boundary with such cells in the inactive box. This procedure carries on till the combined error norm ratio of the entire composite grid becomes less than the specified tolerance as explained above. It is important to note that the individual convergence at each level is not tested but only the convergence of the composite grid as a whole is tested.

5.8.2.4 Plotting the solution

The BoxLib library has a built-in routine `write_plotfile()` for plotting any MultiFab. Thus, we can plot the solution MultiFab by passing the multilevel layout array `mla`, the solution MultiFab `phi`, the array containing the mesh spacing for all levels `dx` and the physical dimensions of the problem (`prob_lo` and `prob_hi`) to this routine. The output data which is generated can be read using the visualization software VisIt [140].

5.8.3 Changes to the library

Some subroutines in the BoxLib library do not work with non-cubic boxes and hence require a change in the signature and bodies. We list below the changes we made to the library routines for them to work seamlessly with non-cubic blocks. To give precedence to these subroutines over the default routines, we make changes to them and copy the appropriate source files to the current project working directory to make sure our versions of the routines are invoked. We first give the file name to which the subroutine belongs and then describe the necessary change. In addition to these changes, we discuss some precautions, compilation options and the method to profiling BoxLib based applications using Scalasca [121] (see Chapter 3) in Appendix B.

- `make_new_grids.f90`: One of the parameters passed to the subroutine `make_new_grids()` subroutine is the scalar `max_grid_size` that is used to specify a cubic block shape. This must be changed to a vector i.e. `max_grid_size(:)` (in Fortran90). When the `boxarray_maxsize()` subroutine is called inside the body of this routine, the vector is divided by the refinement ratio to give the correct box-size at the finer resolution. Similarly vectors `max_grid_size_2(:)` and `max_grid_size_3(:)` must be passed to the subroutine `enforce_proper_nesting()` instead of scalars. The original subroutine declares a local integer variable by the name of `max_grid_size_3` which must be changed to an allocatable vector (dynamic array) using `allocate(max_grid_size_3(mba%dim))`, where `mba` denotes the variable of type Multilevel BoxArray and `dim` represents the dimension field in this structure.

- `multifab_physbc.f90`: This file contains the boundary conditions for periodic, aperiodic or exterior boundaries etc. The way most codes are written in BoxLib does not require the updates of the physical boundaries using `multifab_physbc()`. Since we use a cell-centered

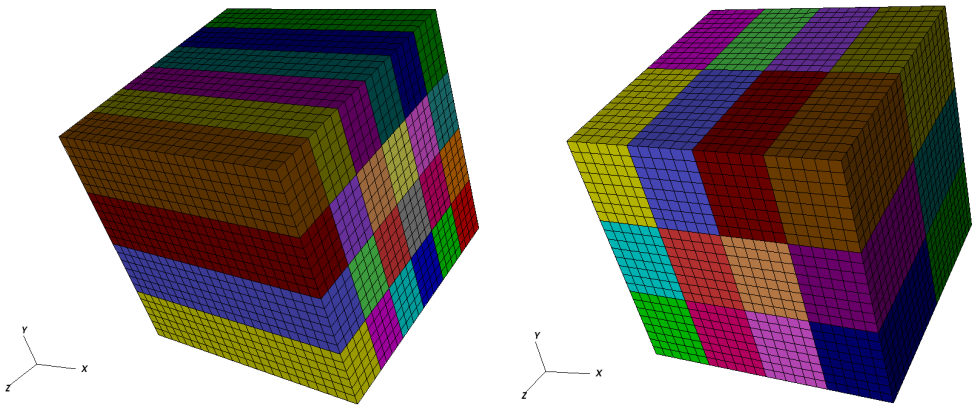
scheme, we must update the Dirichlet boundary conditions and hence we incorporate the condition $\frac{u_0+u_g}{2} = u_a$ for all the boundaries of a 3-D unit cube. Here u_0, u_g and u_a are the near-to-boundary data, ghost data lying outside the actual boundary and the actual boundary condition, respectively, as explained in the introduction of Section 5.7. This subroutine is called from several other subroutines and hence it becomes necessary to change that subroutine call to point to our version of this subroutine. Further, since we use the mesh resolution in setting the boundary conditions, an array containing equal elements in 2-D or 3-D must be passed to this subroutine. This is necessary in the discretized version of the problem because the loop indices by themselves do not translate to the actual distance on the physical domain. Since our implementation is cell-centered, we add a value of half to the loop index and then multiply it with the mesh resolution to get the actual distance on the physical domain.

– `fillpatch.f90`: This file contains several subroutines which call the `multifab_physbc()` subroutine. Since we modify the latter routine for non-cubic blocks and carry out Dirichlet boundary updates for cell-centered implementation, we need to change the invocations of this subroutine in this particular file. The first change comes in the signature of the subroutine `fillpatch()` which must now also contain the mesh spacing for the fine (`dx_fine`) and coarse (`dx_coarse`) grid and also the lower (`prob_lo_in`) coordinates of the problem domain. The fine and coarse mesh spacings are passed as a scalar and the lower coordinates of the problem domain is passed as a vector having three components (as we implement 3-D problems). The scalars must be copied to vectors having three components (which are equal), as the call to our version of `multifab_physbc()` requires vectors. Depending on the call to the fine or coarse grid, the appropriate vectors are passed into the `multifab_physbc()` subroutine.

– `multifab_fill_ghost_cells.f90`: The routine `multifab_fill_ghost_cells()` contained in this file calls the subroutine `fillpatch()` in `fillpatch.f90`. Thus, we need to change the signature of `multifab_fill_ghost_cells()` to address this change. We again pass the fine grid mesh spacing, coarse grid mesh spacing and the lower coordinates of the problem domain to `multifab_fill_ghost_cells()`, which are then passed to the `fillpatch()` subroutine and also utilized in calls to `multifab_physbc()` subroutine. As mentioned above we rename the subroutines by adding an appropriate prefix, rename the files and copy them to the working directory.

5.9 Experimental Results

We now discuss the experimental results for the test problems that we described for single uniform grids and AMR. It should be noted that for comparison of various topologies the application was executed on the same set of cores to eliminate process placement issues. Along



(a) Topology $1 \times 4 \times 6$, cells $= 24 \times 6 \times 4$ (b) Topology $4 \times 3 \times 2$, cells $= 6 \times 8 \times 12$

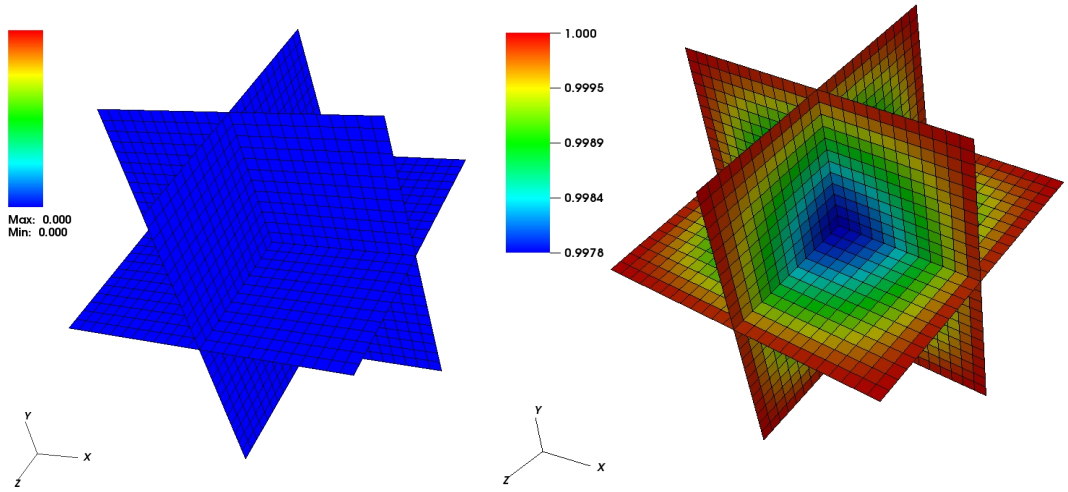
Figure 5.10: Sub-domain shapes/sizes resulting from two of several MPI Cartesian Topologies on a 24^3 domain possible using Listing 5.3

with comparing the execution timings, we extract and compare the cache-misses for both single grids and AMR. Since for AMR there are many sub-routines that are called by BoxLib, we only present the aggregate sum of cache-misses in the top-level subroutine. The test platform for all the tests is the ARC3 facility described previously (see Chapter 3).

5.9.1 Single grid timings

For the single grid problem, we set the Dirichlet boundary conditions to one and update the ghost cells representing the boundaries after every iteration. Since we use a single sub-domain per MPI rank, the domain decomposition completely determines the shape of the sub-domain. As an example, if a 24^3 domain is decomposed as a $1 \times 4 \times 6$ topology, it results in 24 cells in the X-direction, 6 cells in the Y-direction and 4 cells in the Z-direction, respectively. This is achieved by using the layout scheme shown in Listing 5.3. As an example Figure 5.10a shows the box-shapes resulting from a $D_x \times D_y \times D_z = 1 \times 4 \times 6$ domain decomposition on a 24^3 domain and 24 cores (single node). This results in $P_x \times P_y \times P_z = 24 \times 6 \times 4$ cells per sub-domain. The `mpi_dims_create()` topology of $4 \times 3 \times 2$ for 24 cores produces a sub-domain of shape $6 \times 8 \times 12$ on a 24^3 domain as shown in Figure 5.10b. The evolution of the solution for a 3-D domain is shown in Figure 5.11 for iteration counts 0 (initial guess in Figure 5.11a) and 800 (Figure 5.11b) for a uniform mesh having 24^3 cells. The numerical solution advances from an initial guess of zero towards the exact solution i.e. approaches unity everywhere on the domain (as the Dirichlet boundary conditions are set to one).

Table 5.2 compares the execution times per iteration of the topology returned by the default `mpi_dims_create()` (henceforth referred to as MDC) subroutine of MPI and the best topology for 24 to 1536 MPI processes. It is also appropriate to compare the best timings with the

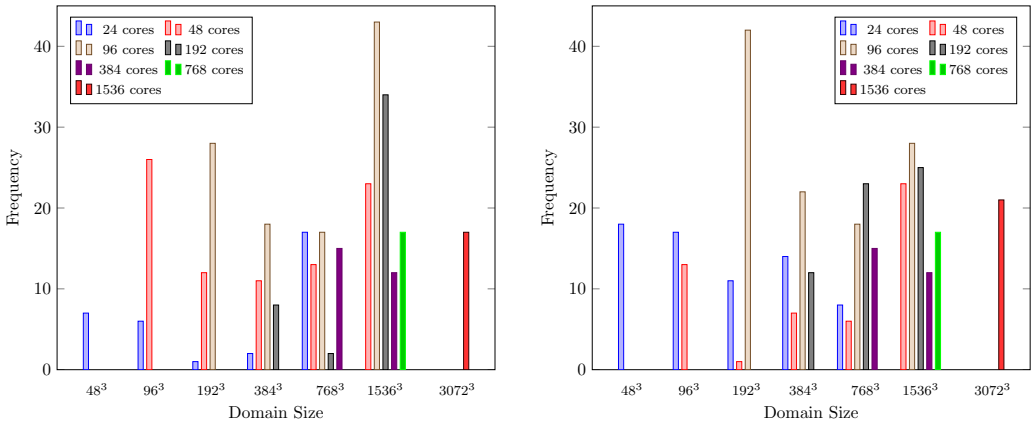


(a) Iterations=0 (b) Iterations=800

Figure 5.11: 2-D slices of a 3-D domain having 24^3 cells at $x = 0.5$, $y = 0.5$ and $z = 0.5$ showing evolution of the numerical solution for $\nabla^2 u = 0$ with Dirichlet boundaries set to 1 at iteration count 0 and 800

reverse of `mpi_dims_create()` (referred to as Rev. MDC) as the code was written in Fortran where the first dimension is the contiguous dimension. For 24 cores (single node), it can be seen that the Rev. MDC outperforms the MDC for all the domain sizes except for 768^3. Further, in no case is MDC the best topology. The number of topologies performing better than the MDC or Rev. MDC is significant for most of the domain sizes and core counts. In BoxLib, by default, communication is not overlapped with computation, yet the communication minimizing topology is outperformed by several topologies. For example, for 96 cores and 3.62 billion degrees of freedom, there are 28 topologies which outperform the MDC topology, the corresponding figure being 23 topologies for 48 cores. Although the best topology ($D_x \times D_y \times D_z$) is $6 \times 16 \times 1$ for 96 cores, the value of $D_y = 16$ is much higher than the D_y for MDC (which is 4).

Let D_{bx}, D_{by}, D_{bz} denote the MPI Cartesian topology process dimensions of the best topologies and D_{sx}, D_{sy}, D_{sz} that of the `mpi_dims_create()` topology. It can be seen from Table 5.2 that $D_{bx}D_{by} \geq D_{sx}D_{sy}$ holds with only two exceptions (Cores=24, Domain=384^3 and Cores=48, Domain=384^3). This implies that the three planes of the compute kernel to be brought into the cache for updating a single plane of data for the best topologies are smaller in size than the ones which are brought into the cache with the communication minimizing topology (MDC). For all the best performing topologies, $D_{by} \geq D_{bz}$ - a criterion that is in agreement with our discussion on optimal sub-domain dimensions in Chapter 4 and [141,142]. We also expand on these relations in Chapter 6. Further, for non-cubic sub-domains $D_{sx}D_{sy} > D_{rx}D_{ry}$, where D_{rx} and D_{ry} denote the Cartesian topology dimensions of the reverse of MDC (or



(a) default `mpi_dims_create()` (b) Rev. `mpi_dims_create()`

Figure 5.12: Number of topologies outperforming the default `mpi_dims_create()` and Rev. `mpi_dims_create()` topology at various domain sizes and number of cores

Rev. MDC). For example, if MDC $= 4 \times 3 \times 2$ then Rev. MDC $= 2 \times 3 \times 4$ and thus $D_{sx}D_{sy} = 4 \times 3 > D_{rx}D_{ry} = 2 \times 3$.

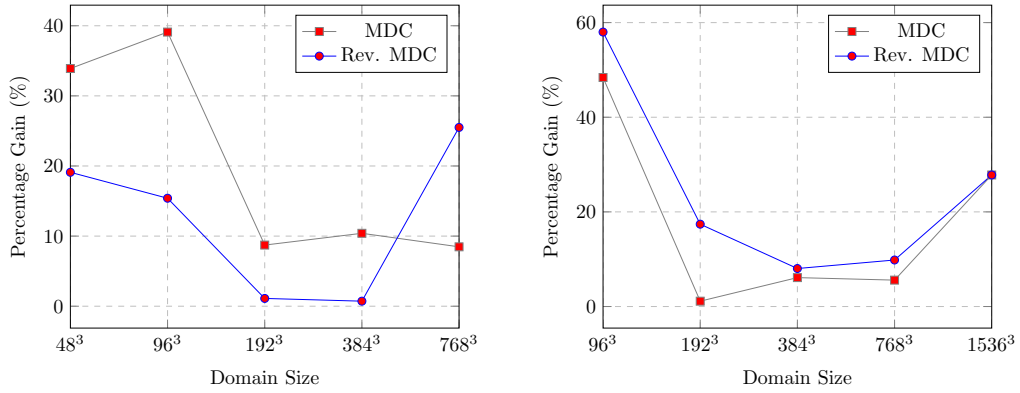
At all processor cores and domain sizes, we were able to find topologies which performed better than the `mpi_dims_create()` and the Rev. MDC topology. Figures 5.12a and 5.12b show the number of topologies which outperformed the MDC and Rev. MDC topology at various domain sizes and cores. Interestingly, even at a domain size of 3072^3 or 28 billion degrees of freedom, there existed 21 topologies which outperformed the MDC and 43 topologies which performed better than the Rev. MDC. The percentage gains of the best topologies over the MDC and Rev. MDC are shown in Figures 5.13a, 5.13b, 5.13c and 5.13d for 24, 48, 96 and 192 cores, respectively. The percentage gain of the best topology over MDC ranged from approximately $1 - 70\%$ and $1 - 66\%$ for Rev. MDC at these core counts, respectively. The percentage gain of the best topology over the MDC for 384 cores at a domain of size 768^3 was 19.8% and 9.67% at a domain size of 1536^3. For 768 cores the gain was 11.30% while being 11.11% for a core count of 1536. This showed that the gains need not decrease with an increasing domain size or core count.

5.9.2 Single grid cache-misses

We use *Scalasca* to extract the Performance API (*PAPI*) metrics, `PAPI_L1_DCM` and `PAPI_L2_DCM` i.e. the L1 data cache-misses and total L2 cache misses for various topologies on a single node (24 cores) which is shown in Listing 5.5 and forms part of the submission shell script to the *Son of Grid Engine* (SGE). The cache-misses are independent of the number of cores and only depend on the sub-domain size per core. It can be noted that while communicating, there are

Table 5.2: Uniform Grid: `mpi_dims_create()` (`MDC`) topology execution times per iteration as compared to best topology times and reverse MDC. #MDC and #Rev. MDC gives the number of topologies performing better than MDC and Rev. MDC, respectively. No Loop blocking/Tiling was used, Intel compiler 17.0.1, OpenMPI 2.0.2

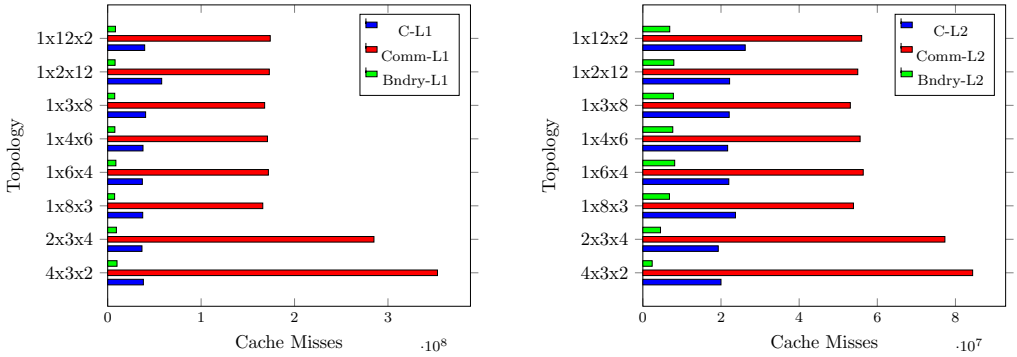
Domain	**Best**	**MDC** (sec)	**Best** (sec)	**Rev. MDC** (sec)	#MDC	#Rev. MDC
		Cores=24	MDC=4x3x2	Rev. MDC=2x3x4		
48^3	1x12x2	3.98E-5	2.63E-5	3.25E-5	18	7
96^3	1x12x2	1.50E-4	9.14E-5	1.08E-4	17	6
192^3	2x6x2	1.95E-3	1.78E-3	1.80E-3	11	1
384^3	1x6x4	1.54E-2	1.38E-2	1.39E-2	14	2
768^3	3x8x1	1.18E-1	1.08E-1	1.45E-1	8	17
		Cores=48	MDC=4x4x3	Rev. MDC=3x4x4		
96^3	1x24x2	1.70E-4	8.77E-5	2.09E-4	13	26
192^3	2x12x2	7.07E-4	6.99E-4	8.46E-4	1	12
384^3	1x8x6	7.69E-3	7.22E-3	7.85E-3	7	11
768^3	2x12x2	5.73E-2	5.41E-2	6.00E-2	6	13
1536^3	3x16x1	6.25E-1	4.51E-1	6.25E-1	23	23
		Cores=96	MDC=6x4x4	Rev. MDC=4x4x6		
192^3	2x24x2	9.10E-4	2.80E-4	8.10E-4	42	28
384^3	4x6x4	4.98E-3	4.05E-3	4.86E-3	22	18
768^3	2x12x4	3.20E-2	2.78E-2	3.19E-2	18	17
1536^3	6x16x1	3.06E-1	2.23E-1	3.25E-1	28	43
		Cores=192	MDC=8x6x4	Rev. MDC=4x6x8		
384^3	4x12x4	2.96E-3	2.42E-3	2.68E-3	12	8
768^3	4x12x4	1.77E-2	1.49E-2	1.59E-2	23	2
1536^3	4x16x3	1.34E-1	1.14E-1	1.47E-1	25	34
		Cores=384	MDC=8x8x6	Rev. MDC=6x8x8		
768^3	4x24x4	1.01E-2	8.1E-3	1.01E-2	15	15
1536^3	4x24x4	6.20E-2	5.60E-2	6.31E-2	12	12
		Cores=768	MDC=12x8x8	Rev. MDC=8x8x12		
1536^3	4x48x4	3.45E-2	3.06E-2	3.51E-2	17	17
		Cores=1536	MDC=16x12x8	Rev. MDC=8x12x16		
3072^3	8x32x6	1.35E-1	1.20E-1	1.61E-1	21	43



(a) 24 cores (b) 48 cores

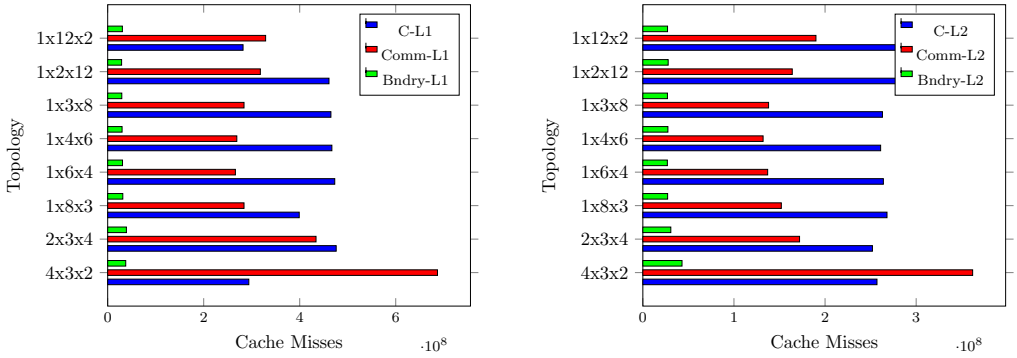
(c) 96 cores (d) 192 cores

Figure 5.13: Percentage gain of the best topology over MDC and Rev. MDC for varying domain sizes and cores



(a) L1 cache-misses (b) L2 cache-misses

Figure 5.14: L1d and L2d cache-misses for domain=48^3 for the Compute kernel (C), Communication (Comm) and Boundary update (Bndry)



(a) L1 cache-misses (b) L2 cache-misses

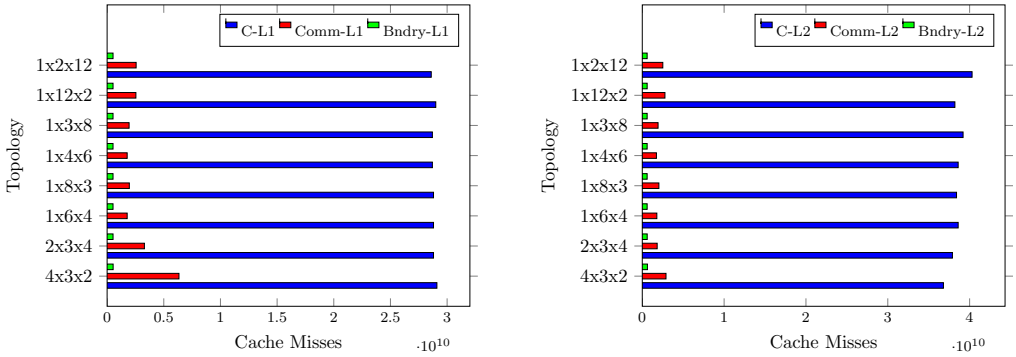
Figure 5.15: L1d and L2d cache-misses for domain=96^3 for the Compute kernel (C), Communication (Comm) and Boundary update (Bndry)

no cache misses while the data is actually being communicated but only when the data is being packed/unpacked or being copied to the MPI buffers from the application buffers. Though the L2 cache is a *Unified* cache, there are separate counters available for the data and instruction cache-misses. Such separate options are not available for the L3 cache which is Unified, *Inclusive* and shared among multiple cores of the socket. All these options can be checked using `papi_avail` at the command line.

```
1 export SCOREP_METRIC_PAPI=PAPI_L1_DCM,PAPI_L2_DCM
```

Listing 5.5: PAPI metrics for Cache Misses using Scalasca

Figures 5.14a, 5.14b show the L1 and L2 cache-misses for a domain of size 48^3. At such a domain size, the number of sub-domain cells per core is $P_x \times P_y \times P_z = \frac{48^3}{24} = 4608$, excluding



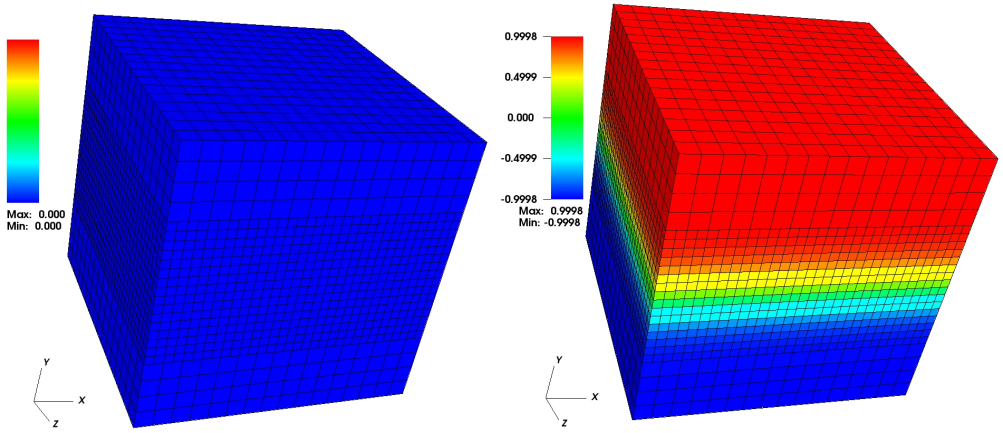
(a) L1 cache-misses

(b) L2 cache-misses

Figure 5.16: L1d and L2d cache-misses for domain=384^3 for the Compute kernel (C), Communication (Comm) and Boundary update (Bndry)

the ghost cells. This equates to a single array of size $\frac{4608\times8}{1024}$ = 36 KB: a size which slightly exceeds the L1d cache of 32 KB but is less than the combined size of L1d and L2 cache. Even with 2 arrays (as is the case with unweighted Jacobi for solving Laplace Equation), the size of 72 KB (without ghost cells) is small enough to fit in the L1d and L2 cache. But even at this "in-cache" data-size, the cache-misses for communication in the $4 \times 3 \times 2$ topology are higher than that of a topology such as $1 \times 4 \times 6$ or $1 \times 6 \times 4$. This is because the X-plane, i.e. the plane which is perpendicular to the unit-stride dimension (as the language of implementation is Fortran), is 4 times larger in $4 \times 3 \times 2$ as compared to $1 \times 4 \times 6$. Thus, it is the packing/unpacking cache-misses which contribute to a significant fraction of the total execution time. Further, at this domain size the communication L1 cache-misses are approximately 3 times that of the compute misses for topologies other than the MDC ($4 \times 3 \times 2$), the factor being 7 for the latter. For the L2 cache-misses, this ratio is in the range of 3 to 4.1. As the compute domain begins to increase in size, the compute cache-misses start exceeding the communication cache-misses as shown in Figure 5.15a and Figure 5.15b. Interestingly, the communication cache-misses for the MDC for a domain of 96^3 still exceed the compute cache-misses due to the large X-planes, whose packing/unpacking contribute maximally to the total execution time. The ratio of communicate to compute cache-misses, i.e. $\frac{Comm}{C}$, for the topology $4 \times 3 \times 2$ in a domain of size 96^3 is approximately 2.75 for L1 and 1.42 for L2, whereas it is less than one for the other topologies (except the case of L1 for $1 \times 12 \times 2$).

As the size of the domain increases to 384^3, the compute cache-misses become significantly greater than the communication cache-misses, as shown in Figure 5.16a and 5.16b. Both these figures show that the compute cache-misses for most topologies are almost equal, as is expected, because of the continuous compute sub-domain per core (i.e. the computation is not divided into the Independent Compute and Dependent Planes). Although the communication cache-misses decrease in magnitude as compared to the compute misses, the relative difference between



(a) Initial solution (b) Converged solution

Figure 5.17: Initial guess of zero to the final solution for 2 levels of a 16^3 domain for our AMR test problem

communication cache-misses almost remains the same, i.e. they are higher for the MDC as compared to other topologies which are shown in Figures 5.16a and 5.16b. This shows that our abstract model for determining the optimal topology based on cache-misses still holds due to the packing/unpacking cost of cache-misses in communication despite no overlap of computation with communication.

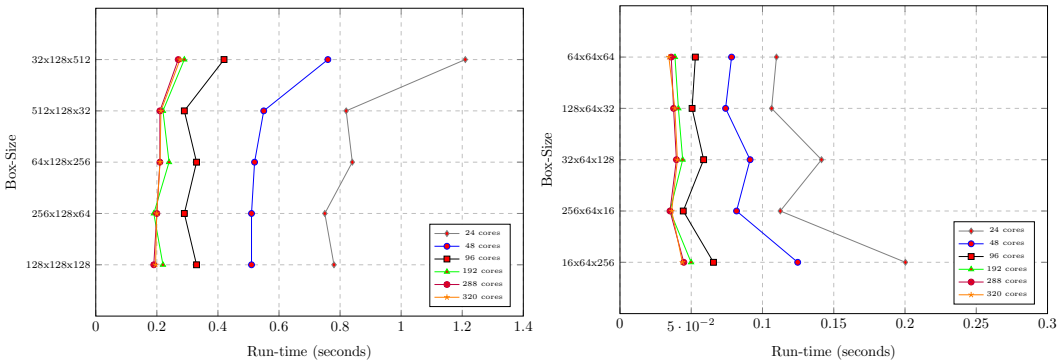
5.9.3 AMR timings

We first evaluate the behaviour of non-cubic blocks on domains of sizes 512^3 and 256^3 for one level of Adaptive Mesh Refinement i.e. the coarsest grid is refined only once. For each of these cases, the total number of boxes at level 1 is 64, out of which 32 are refined (active at level 2) and 32 are not refined (active at level 1). At level 2, there are a total of 256 boxes (as each of the 32 inactive blocks have been divided into 8 boxes and $32 \times 8 = 256$). Thus, a total of 288 active boxes are updated for the solution. Considering a three level problem (i.e. two levels of AMR), 128 boxes out of a total of 256 boxes at level 2 are refined again to give $128 \times 8 = 1024$ active boxes at level three. Thus, in a three level problem, we have a total of $32 + 128 + 1024 = 1184$ active boxes in all which must be updated at each iteration of the solver. While varying the box shape, the volume of each box is kept constant. Figure 5.17 shows the initial guess of zero (Figure 5.17a) and the final solution (Figure 5.17b) for our AMR test problem having two levels, with the base grid/coarsest level having $16 \times 16 \times 16$ cells.

Figure 5.18a shows the performance of various box-shapes for a domain of size 512^3 and a two level problem. It can be seen that a non-cubic box shape of $256 \times 128 \times 64$ outperforms or matches the performance of the cubic block of $128 \times 128 \times 128$ from 24 to 192 cores. Since in

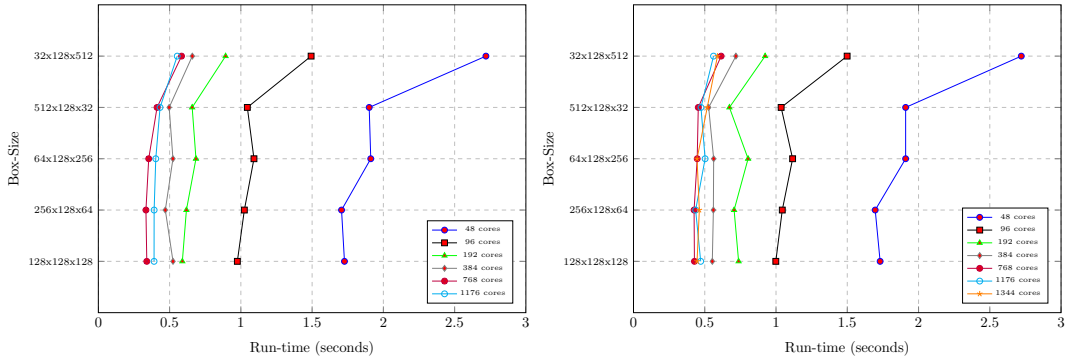
Table 5.3: AMR: Gain percentage for the best performing topology over MDC for various core counts, MDC=Solve time/iteration in seconds, Best=Best solve time/iteration

Domain=256^3, **2-levs, OpenMPI 2.0.2**						
Cores	24	48	96	192	288	320
MDC (sec)	1.10E-01	7.83E-02	5.29E-02	3.86E-02	3.60E-02	3.43E-02
Best (sec)	1.06E-01	7.41E-02	4.44E-02	3.51E-02	3.51E-02	3.43E-02
Gain (%)	3.10	5.36	16.07	9.07	2.50	0.00
Domain=512^3, **2-levs, OpenMPI 2.0.2**						
MDC (sec)	7.80E-01	5.10E-01	3.30E-01	2.20E-01	1.90E-01	2.00E-01
Best (sec)	7.50E-01	5.10E-01	2.90E-01	1.90E-01	1.90E-01	2.00E-01
Gain (%)	3.85	0.00	12.12	13.63	0.00	0.00
Domain=512^3, **3-levs, OpenMPI 2.0.2**						
Cores	48	96	192	384	768	1176
MDC (sec)	1.73E+00	9.99E-01	7.38E-01	5.53E-01	4.28E-01	4.72E-01
Best (sec)	1.70E+00	9.99E-01	6.73E-01	5.28E-01	4.25E-01	4.36E-01
Gain (%)	1.99	0.00	8.75	4.61	0.58	7.53
Domain=512^3, **3-levs, Intel MPI 17.1.132**						
MDC (sec)	1.73E+00	9.76E-01	5.90E-01	5.24E-01	3.40E-01	3.91E-01
Best (sec)	1.71E+00	9.76E-01	5.90E-01	4.70E-01	3.34E-01	3.91E-01
Gain (%)	1.19	0.00	0.00	10.31	1.76	0.00



(a) Domain 512^3 (b) Domain 256^3

Figure 5.18: Strong Scaling (time/iteration) two AMR levels problem with boxes of varying shapes but equal volume using Intel compilers 17.0.1 and OpenMPI 2.0.2, Optimization flags: `-O3 -xHost -ip -align array64byte`



(a) Domain 512^3 with Intel MPI

(b) Domain 512^3 with OpenMPI

Figure 5.19: Strong Scaling (time/iteration) three AMR levels problem with coarsest grid being 512^3 and boxes of varying shapes but equal volume using Intel compilers 17.0.1, Intel MPI 2017.1.132, OpenMPI 2.0.2 and Optimization flags: `-O3 -xHost -ip -align array64byte`

Fortran the first dimension is the contiguous data dimension, a box of shape $256 \times 128 \times 64$ has twice the data points in the contiguous dimension as a box of shape $128 \times 128 \times 128$. Another topology that ouperforms the cubic block is that with a box-shape of $512 \times 128 \times 32$ at 96 and 192 cores - a box-shape which again has a large contiguous data dimension. The number of communication elements grows with an increasing size of a particular dimension in a non-cubic block and thus, there is always a trade-off between the gain in packing/unpacking and the loss in communicating large volumes of data. The complexity of this trade-off explains why we do not see a consistent performance ranking across all process counts. Figure 5.18b also shows the same conclusion for a 256^3 domain in the sense that the cubic box-shape is not the optimal choice at all core counts. Conversely with a process count of 320 when each box resides on a different core, the cubic topology outperforms the best performing non-cubic topology by 1.85% at a domain size of 512^3 and by 5.50% at a domain size of 256^3.

Figures 5.19a and 5.19b show the performance of various block shapes for the AMR test problem when the number of levels is three. It can be seen from Figure 5.19a that the cubic block size is again not optimal for 48, 384 and 768 cores and the performance difference between it and the optimal non-cubic block ranges between $1.18 - 10.30\%$. For Figure 5.19b with OpenMPI, the range of performance difference is $1.9 - 8.74\%$. The maximum and minimum ratio of the execution time per solve iteration when using OpenMPI to when IntelMPI is used is 1.27 and 0.99, respectively, for a domain of size 512^3 (see Figure 5.19a and Figure 5.19b). The ratio increases as the number of processes increase from 48 to 1176. It is not correct to say that one MPI implementation is faster than the other as the allocation of nodes changes between using the two MPI implementations. With 1344 cores, i.e. when every box is placed on a separate core, the results are inconclusive in the sense that the box-shape of $64 \times 128 \times 256$ performs better than the box-shape $128 \times 128 \times 128$ in some runs and worse than the latter in others

Table 5.4: Macroscopic view: Total L1, L2 and L3 cache-misses in the AMR application with 2 levels, domain=512^3 with box-sizes $128 \times 128 \times 128$ and $256 \times 128 \times 64$

Box-size	Total L1	Total L2	Total L3
128x128x128	1.77E+11	1.36E+11	7.08E+10
256x128x64	1.68E+11	1.34E+11	6.73E+10

(the average execution time was the same for both).

5.9.4 AMR cache-misses

We profiled two cases of the single level AMR with block-sizes $128 \times 128 \times 128$ and $256 \times 128 \times 64$ covering the domain of 512^3 to compare the cache-misses. The cache-comparison in AMR is significantly more complex than for the single uniform grid case due to a substantial increase in complexity of the library's functions and the algorithm itself. Our major focus is on the compute cache-misses in the solve phase and the cache-misses while performing packing/unpacking for all types of communication: exchange of next-to-boundary-layers at the same level ; restriction from the finer grid to the immediate coarser grid and interpolation from the coarser to the finer grid.

5.9.4.1 Macroscopic view

Table 5.4 shows the total cache-misses for the complete AMR application with two different box-sizes and a domain of 512^3 with two levels. From Table 5.3 it can be seen that the non-cubic blocks outperform the cubic-blocks by 3.85% using a single node of ARC3 (i.e. 24 cores, OpenMPI 2.0.2, 2-levs problem) for a domain of size 512^3. The difference between the L1 and L2 cache-misses is 5.08% and 1.47% respectively, with the cache-misses being higher for the cubic-blocks. The percentage difference in the L3 category is 4.94% with the non-cubic blocks again performing better than the cubic blocks.

5.9.4.2 Microscopic view

Table 5.5 shows the major sources of cache-misses for a domain of size 512^3 and the two block sizes $128 \times 128 \times 128$ and $256 \times 128 \times 64$. The user-defined subroutine `update_phi_3d()`, which is the main computational kernel, accounts for the majority of cache-misses and the percentage difference between the two block sizes for L1, L2 and L3 misses is 6.77, 4.18 and 1.35%, respectively. As a relative percentage of the total cache-misses, this subroutine makes up $26.41 - 29.9\%$, $37.11 - 39.16\%$ and $41.08 - 43.85\%$ of the total L1, L2 and L3 cache-misses, respectively. The difference in the cache-misses could possibly be because of the size of the three planes which are needed to update a single plane. Thus, a single plane of $128 \times 128 \times 128$ has $128 \times 128 = 16384$ elements as opposed to a single plane of $256 \times 128 \times 64$ which has

Table 5.5: Cache-Misses Subroutines: Major sources of cache-misses for a 2 level AMR with domain=512^3, block-sizes=$128 \times 128 \times 128$ and $256 \times 128 \times 64$

Subroutine	128x128x128			256x128x64			**Description**
	L1	L2	L3	L1	L2	L3	
`init_phi_3d`	3.15E8	3.18E8	1.70E7	3.15E8	3.19E8	1.86E7	Initialize solution
`init_rhs_3d`	7.78E7	7.80E7	2.33E7	7.79E7	7.79E7	2.44E7	Initialize RHS
`update_phi_3d`	4.68E10	5.04E10	2.91E10	5.02E10	5.26E10	2.95E10	Update Solution
`fill_boundary`	7.20E9	3.04E9	2.49E9	4.33E9	2.29E9	1.52E9	Boundary exchange
`fill_ghost_cells`	4.97E10	8.38E9	5.90E9	4.61E10	6.85E9	4.08E9	Interpolation
`cc_restriction`	2.54E10	3.31E10	1.49E10	2.37E10	3.15E10	1.48E10	Restriction

$256 \times 128 = 32768$ elements. However, we expect the latter to have fewer cache-misses while performing the packing/unpacking in communication because of the smaller size of the X-plane, i.e. the plane which is orthogonal to the unit-stride dimension. This plane (X-plane) has a size of 128×128 for the block of size $128 \times 128 \times 128$ and a size of 128×64 for the block of size $256 \times 128 \times 64$.

The `fill_boundary()` subroutine, which exchanges boundaries between sub-domains at the same level, shows this behaviour. Here the non-cubic blocks perform better than the cubic-blocks and the percentage difference between these are 39.86% for L1, 24.67% for L2 and 38.95% for L3 cache-misses. But it should be noted that as a percentage of the total cache-misses these form only $4.06 - 2.58\%$ for L1, $2.24 - 1.71\%$ for L2 and $3.51 - 2.25\%$ for L3 for the blocks of size $128 \times 128 \times 128$ and $256 \times 128 \times 64$, respectively. The interpolation subroutine forms a significant fraction of the total cache-misses, being $28 - 27.48\%$ for L1, $8.38 - 5.09\%$ for L2 and $8.33 - 6.05\%$ for L3 for the cubic and non-cubic blocks, respectively. For the BoxLib implemented restriction operator, the corresponding values are $14.3 - 14.1\%$ for L1, $24.35 - 23.45\%$ for L2 and $21.1 - 21.98\%$ for L3 for the cubic and non-cubic blocks, respectively. It can be seen that the cache-misses in the communication routines, i.e. exchanging ghost data, interpolation and restriction for the non-cubic block, are consistently less than the cubic blocks although the latter communicates a smaller data volume. This supports our model which conveys that using a non-cubic block with a smaller X-plane (Z-plane for the C language) reduces the packing/unpacking/plane-update cache-misses.

5.10 Difficulties in validating the hypothesis

In the previous chapter, we formulated a strategy for minimizing the cache-misses of a sub-domain and showed the superiority of such partitions by experimenting on single grids. Our comparison showed that our cache-minimizing topologies performed better than the communication minimizing topology for almost all combinations of grid sizes and process counts. Overlap of communication with computation formed a significant part of our analytical derivation for cache-minimizing topologies. The reason is that when communication is overlapped with com-

putation, both while packing/unpacking and communicating data, the next-to-halo layers are accessed separately after the halo data arrives. This has the advantage of MPI advancing its communication progress engine while the serial computing thread updates the Independent Compute kernel but at the same time suffers from a disadvantage that the next-to-halo layers now cannot be updated along with the Independent computational kernel, resulting in extra cache-misses. There are several reasons why we emphasize overlapping communication with computation and we list these below:

1. *Non-blocking communication in MPI*: The reason why these non-blocking routines exist is that we are expected to overlap communication with computation. MPI 3.1 also has versions for non-blocking collective operations.

2. *Increasing distance*: As the nodes grow fatter and the number of nodes in a cluster continues to increase, the distance between cores is increasing. Thus, it would/has become imperative to overlap communication with computation in future/current architectures.

The hypothesis that we formulated in the previous chapter holds only partially when evaluating Adaptive Mesh Refinement in BoxLib. For single grids, the communication minimizing topology (MDC) never outperforms the cache-minimizing topologies for any data size and core counts in our experiments. Since the codes are in Fortran, we also took into account the reverse communication minimizing topology (Rev. MDC) but there existed topologies which outperformed both MDC and Rev. MDC for all the cases. This demonstrates that the communication minimizing topology is not the optimal choice for single grids as shown previously. For AMR codes, there existed cases where the MDC was outperformed by specific non-cubic sub-domains, thus establishing that the MDC is not always the optimal choice at all data sizes or processor counts. However, the superiority of the cache-minimizing topologies was not at all clear cut in these cases. The following plausible reasons explain why our hypothesis partially fails when considering AMR using BoxLib, and also the difficulties in analyzing BoxLib codes.

1. *Communication and Computation*: In BoxLib, communication of the halo zones is not overlapped with computation and, further, the packing and unpacking of data from the boxes does not use MPI *derived* data types. Thus, there is no overlapping while packing/unpacking or communicating data. The sub-domain is updated when the data arrives from neighbouring processes and it is treated as a contiguous sub-domain without any need for updating the planes separately. This completely eliminates the cache-misses that we calculated separately for the Dependent Planes in our abstract high level mathematical model for minimizing cache-misses.

2. *Internal data structures*: It is difficult to estimate the size of the metadata and the consequent effect on the application performance that BoxLib maintains for both single grids and AMR. Clearly, the metadata for the latter is more complicated and much larger in size.

3. *No Control over distribution of boxes*: The user does not have any control over distribution of boxes in AMR with BoxLib. This is completely controlled by BoxLib using the Knapsack or Morton ordering with a dynamic switching scheme implemented to choose the appropriate algorithm. Since boxes are distributed per-level, BoxLib does not distinguish between inactive or active boxes. An inactive box is a box where the solution is not updated: thus there is a large probability that the active boxes may not be load-balanced.

4. *Load-balancing over shape*: The load-balancing, i.e. the number of boxes per core, changes when the shape of the box is changed even though the volume remains constant. Thus, the load-balancing algorithm used by BoxLib takes into account the sub-domain points in each direction.

5.11 Summary

Adaptive Mesh Refinement (AMR) is a computational technique where local regions on a mesh are refined to obtain an increased accuracy in those regions. It helps to direct the compute resources towards regions of interest (or higher error) rather than devoting them to a globally refined mesh. Though theoretically this is an ideal strategy, software packages such as BoxLib which are used for building complex multiscale multiphysics structured AMR applications, incur additional overheads in the form of maintaining metadata and synchronization in a parallel settings.

The parallelization in BoxLib is abstracted away from the user and thus the user is free to focus on the problem, but at the cost of losing some of the control of the execution. Using BoxLib, we have tested the applicability and extension of our previously formulated hypothesis that there exist cache-miss minimizing topologies which outperform the communication minimization topology in solving PDEs using point iterative methods such as Jacobi iteration on structured 3-D uniform grids. We further extended this evaluation to AMR codes with up to 3 levels (2 refined, 1 unrefined). All the codes for the uniform grid and AMR were developed using Fortran90 in BoxLib and tested with no overlap of communication and computation. In this process, we implemented an MPI Cartesian topology of MPI processes that can be used in BoxLib for single and multiple boxes per core for uniform meshes. Further, we compared the execution timings of uniform as well as AMR codes, while profiling the cache-misses for both cases to experimentally investigate the validity of our aforementioned hypothesis.

Chapter 6

Multigrid

In the previous chapter we evaluated the use of non-cubic sub-domains on single grids and with *Adaptive Mesh Refinement* (AMR) using a library called *BoxLib*. We demonstrated that our hypothesis, that there exist cache-miss minimizing domain partitions (or Boxes) that outperform cubic sub-domains, holds true for uniform meshes even with blocking communication (as in BoxLib). When using a structured, nested, AMR hierarchy, the hypothesis is only partially true due to a multitude of issues. These issues are strongly linked to the BoxLib implementation and include the load imbalance and the non-overlap of communication with computation. In this chapter, we continue to evaluate our hypothesis, but now using a multiple grid, hierarchical convergence acceleration technique called *Geometric Multigrid*. Our results will validate our hypothesis for this important class of iterative method, but will also uncover additional subtle factors in determining optimal sub-domain dimensions. The key focus in this chapter again remains on investigating, quantifying, measuring and improving the parallel efficiency by predicting high performing domain partitions.

6.1 Introduction

After a domain has been discretized to numerically approximate a linear PDE, iterative methods such as Jacobi, weighted Jacobi (ω-Jacobi), Gauss-Seidel (GS), Red-Black Gauss-Seidel (RBGS), Conjugate Gradient (CG) and others can be used to compute the solution of this discrete system [33, 37, 52]. Due to the slow rate of convergence of these iterative methods, and the time taken to solve large systems on uniform structured grids, multilevel algorithms have been created that accelerate the rate of convergence to the solution. The Multigrid [25,63] method is an optimal hierarchical method which can be used for solving *sparse* systems of linear equations that arise from a local discretization of Elliptic PDEs in $\mathcal{O}(N)$ time, where N is the number of unknowns or degrees of freedom (dof) in the system. The hierarchy in Multigrid consists of several linear systems corresponding to discretizations on several levels of grids of

decreasing resolution, where the finest level grid represents the actual problem to be simulated. It accelerates the convergence of the solution by quickly and systematically eliminating low frequency error components on the series of coarse grids. To further decrease the solve time of Multigrid methods, they are parallelized on distributed, shared memory or hybrid architectures to allow simulation of extremely large scale problems [86, 143, 144], where the number of unknown variables can be of the order of billions or trillions. It is the parallelization of Multigrid that is challenging and requires a careful design and implementation to achieve near perfect *Weak Scaling* and thus preserve its theoretical optimality.

When Multigrid is parallelized over distributed-shared memory architectures, traditionally, the domain partitioning creates cubic partitions of the mesh to minimize overall communication. We extend and apply our high level analytical model in the scenario of multiple grid levels of Multigrid to investigate its effectiveness on this optimal algorithm. To this effect, we first extend the model to Geometric Multigrid (GMG) and again show that "close to 2-D" partitions for GMG can give higher performance than the partitions returned by the default `MPI_Dims_create()` function which minimizes the communication volume by default. Further, our model seeks to put this in the context of all the factors that might influence the choice of sub-domain shape and size. Thus, we qualitatively and quantitatively consider factors such as cache-misses, prefetching, cache-eviction policy, Vectorization etc., and explore their effect on determining optimal sub-domain dimensions. Though these factors have been separately well explored in the literature, the focus of our work is on establishing a connection between them and *domain partitioning*. We present the results of our investigations and discuss their limitations to open further research avenues. It may be noted that we use the term Multigrid to refer to GMG and not *Algebraic Multigrid* (AMG), the latter being beyond the scope of the current work.

The chapter begins by giving a general introduction to GMG and our aim in the current work. Following it is a detailed description of GMG and an explanation of the terms associated with it. We then describe the extension of our model to Multigrid along with underlying assumptions. An attempt to identify, explain and connect various serial parameters to decide optimal sub-domain dimensions evolves as the next logical step. This step can be considered as a part of the single uniform grid and is equally applicable there but its conception lies in the multiple grids scenario. Next we describe the mixed-boundary value test problem followed by our experimental results. Our results lead us into conclusions and a discussion of our work.

6.2 Motivation and Contribution

Multigrid adds a significant layer of complexity over single level uniform grid solvers due to a multiple level grid hierarchy, a decreasing computation to communication ratio at coarser grid

levels and the appearance of inter-grid transfer operators which are based upon higher order stencils themselves. In the pure sense AMR is a grid resolution technique whereas Multigrid is a convergence acceleration method. Both these computational techniques are extremely important in Scientific Computing and belong to the set of candidates for Exascale computing. Though fundamentally different, their structured versions have a common feature in the form of using uniform structured grids of varying resolution. This common factor, the optimality of Multigrid for Elliptic problems, and its vast applications in real world problems become our motivation for extending/applying our high level mathematical model to Parallel GMG. The following are the contributions of the current chapter:

- Extension of our *quasi-cache-aware* model for minimizing cache-misses to Parallel GMG.
- Demonstration that the fine grid execution time dominates the total solve time and hence even when a topology is sub-optimal at coarser levels, this cannot offset the effect of the optimal topology at the finest level.
- Realization that the *Smoothing, Restriction* and *Interpolation* operators have equivalent characteristic expressions for cache-misses, even when the Smoothing operator is a 7-pt stencil and the latter operators are represented by a 27-pt stencil.
- Identification and connection of other Serial Control Parameters (SCP) such as *Vectorization, Prefetching, Least Recently Used* (LRU) policy, and *Cache Line Utilization* (CLU) to optimal sub-domain dimensions.
- Experimentally verify that our model is independent of the hardware (using test platforms ARC2 and ARC3 at the University of Leeds) and software by using a combination of compilers (Intel and GNU) and MPI implementations (OpenMPI and Mvapich2) to obtain the same relative behaviour of topologies, along with the observation that the execution timing curve is a characteristic of the compiler that is used.
- Developing a lightweight, dynamic cache space tiling/loop-blocking *heuristic*, dependent on the shared L3 cache and the number of arrays in the *Working Set Size* (WSS).
- Demonstrate the effect of domain decomposition by passing an equal number of X, Y and Z planes through a hierarchy of network elements and measuring the execution time.
- Measuring the positive/negative accuracy of our model to demonstrate that the accuracy need not decrease with an increasing number of cores.
- An overall demonstration through theory and experimentation that the problem of domain decomposition for GMG is much more complex than just minimizing the communication volume.

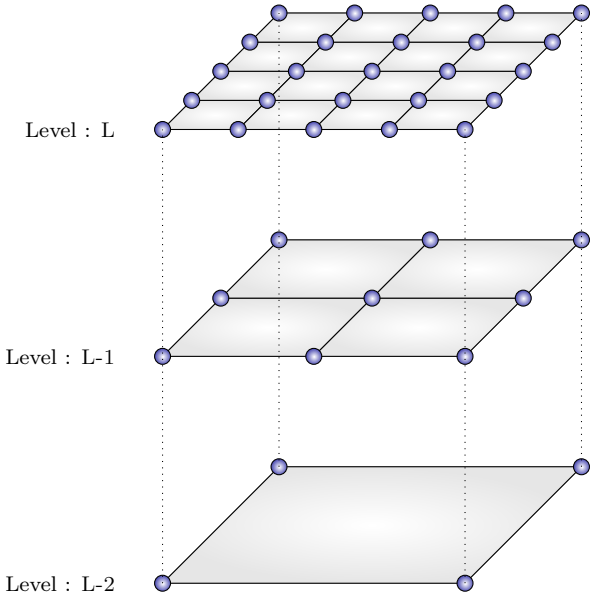


Figure 6.1: Decreasing mesh resolution with decreasing level in 2-D Geometric Multigrid

6.3 Multigrid

Local Iterative schemes [25, 37, 63, 72] such as weighted Jacobi, Gauss-Seidel, Red-Black Gauss-Seidel, can remove high frequency error components quickly (known as smoothing) but decrease the low-frequency error spectrum very slowly. Thus, the overall convergence is slow. These low-frequency components can be represented as relatively high frequency components on coarser grids [25, 59, 63] and thus effectively smoothed on that grid. These smoothing properties of certain iterative methods, and the equivalent system of equations at various levels, form the basis of Multigrid [59].

Multigrid [25, 37, 63, 72] is a multilevel convergence acceleration concept that involves using coarser forms [63, 72, 73] of the given fine grid discretization to remove the low-frequency errors and more efficiently provide an estimate of the approximated solution. Figure 6.1 shows a grid hierarchy of decreasing grid resolution where the coarser grids can be used to remove the low frequency errors. Clearly, the number of unknowns on the coarse grid are fewer and this leads to reduced computation on those grids. Further, the convergence factor of a single grid smoother is approximately $1 - \mathcal{O}(h^2)$, where h is the grid spacing (assumed as uniform in all directions) and for each successive coarse grid, the grid resolution decreases [63]. As mentioned in Chapter 2, depending on the pattern of the traversal between grids, two common types of cycles are categorized as *V-cycles* and *W-cycles* [25, 63]. The following section introduces notations to explain the concept of Multigrid in detail, focusing on the V-cycle.

6.3.1 Notation used and Multigrid Steps

Let $A^h u^h = f^h$ denote a linear system of equations arising from a local discretization of a linear Elliptic PDE, where the superscript h denotes the grid spacing. Successive grid levels (finest to coarsest) are represented as: $\Omega^h \rightarrow \Omega^{2h} \rightarrow \Omega^{4h} ... \rightarrow \Omega^{2^i h}$. We use *standard* coarsening in our implementations which reduces the total degrees of freedom by approximately one-eighth on the immediate coarser grid in 3-D (one quarter in 2-D). After ν_1 *pre-smoothing* iterations on Ω^h, an approximation to u^h is obtained (denoted by v^h) and the *residual* is then calculated as $r^h = f^h - A^h v^h$. A restriction (I_h^{2h}) operator transfers this residual (r^h) to the next immediate coarser grid (Ω^{2h}). In the 2-grid method (detailed in the next section), the error e^{2h} is obtained after solving $A^{2h} e^{2h} = r^{2h}$ (error equation) exactly on the coarser grid. This error is then transferred back to the finer grid using the *interpolation/prolongation* operator (I_{2h}^h) to obtain a better approximation to the solution on the finer grid, followed by ν_2 *post-smoothing* iterations. For Multigrid, the error equation is not solved exactly, instead it is replaced by a recursive use of the 2-grid method to update the estimated error. Only at the coarsest level is an exact solve used. The recursive algorithm halts when the ratio of the current *norm* of the residual ($||r_k^h||$) on the finest level to its initial norm (($||r_0^h||$)) becomes less than a specified tolerance. Typically the pre-smoothing (ν_1) and post-smoothing (ν_2) iterations of the smoother vary between one and three for most practical problems [59].

6.3.2 2-grid Algorithm

The basis of Multigrid is the *2-grid correction scheme* which forms the heart of the Multigrid concept. The following sequence of steps explains this method in detail:

1. Choose a starting estimate for u^h.
2. Approximate u^h satisfying $A^h u^h = f^h$ using ν_1 iterations of an iterative (smoothing) scheme, starting with the latest estimate, to obtain $u^h_{approx} = v^h$.
3. Calculate residual $r^h = f^h - A^h v^h$ for Ω^h.
4. Transfer (Restriction) r^h to Ω^{2h}: let it be denoted by r^{2h}.
5. Solve $A^{2h} e^{2h} = r^{2h}$ exactly to obtain e^{2h}.
6. Transfer (Interpolation/Prolongation) e^{2h} to Ω^h: let it be denoted by e^h.
7. Obtain better approximate for u^h i.e. $u^h_{approx} = v^h + e^h$.
8. Improve u^h_{approx} using ν_2 further iterations of the smoother.
9. Repeat procedure from step 2 until convergence for the finest grid Ω^h.

The two vital steps of Restriction and Interpolation for transferring the residual and the error respectively, are explained in the next section. In general, we refer to them as *Transfer* operators or Inter-grid Transfer operators as they determine the flow of information between the fine and coarse grids.

6.4 Inter-grid Transfer Operators

The current section explains in detail how to exchange information between the fine and coarse grids. The discussion assumes standard coarsening i.e. the degrees of freedom in each direction decrease by a factor of two from the immediate fine grid. Both the inter-grid transfer operators can be treated as stencils and, in this particular case, they are treated as 27-pt stencils in 3-D.

6.4.1 Restriction

A Restriction operator transfers the residual from the fine grid to the immediate coarse grid. The operator acts on the fine grid (Ω^h) residual vectors (r^h) to produce a coarse grid vector (r^{2h}) i.e. $I_h^{2h} r^h = r^{2h}$. The simplest restriction operator is *injection*, which equates the value of the residual at a point on the finer grid to the corresponding point on the coarser grid. In 1-D, injection implies $r_{2j}^h = r_j^{2h}$ when standard coarsening is used and where j is varied over all the coarse grid points. Thus, injection is simply an identity function for the alternate fine grid points which correspond to the coarse grid points. The *full weighting restriction* operator considers the average of the neighbouring points and is expressed as $r_j^{2h} = \frac{1}{4}(r_{2j-1}^h + r_{2j+1}^h + 2r_{2j}^h)$ (in 1-D), where j varies over all the points of the coarse grid. Let α_i be the weight of the i^{th} neighbouring point, then $\sum_{i=1}^{n} \alpha_i = 1$, where the summation is over all the n immediate neighbouring points and the point itself. Generally, $n = 3, 9$ and 27 for 1-D, 2-D and 3-D in the standard cases, respectively. *Stencil notation* can also be used to specify the weights of the immediate neighbours. The advantage of this notation is that it maps directly to the geometrical arrangement of points in 1-D and 2-D. A *full restriction and half restriction* operator in 2-D is shown below [59]:

- *Full Restriction* in 2-D: The central point has a weight of $\frac{1}{4}$, the horizontal and vertical directions have a weight of $\frac{1}{8}$ and the corner points are scaled by a factor of $\frac{1}{16}$:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^{2h}.$$

- *Half-restriction* in 2-D: The corner points are not taken into consideration and the weight of the central point is $\frac{1}{2}$, whereas the points in the horizontal and vertical direction contribute $\frac{1}{8}$ their value:

$$\frac{1}{8}\begin{bmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{bmatrix}_{h}^{2h}.$$

Figure 6.2 illustrates a *27-point full restriction stencil* in 3-D where the weights of each of the points considered are shown (equal weights are color coded). It can be seen that the points at the faces have a weight of $\frac{1}{16}$. These are the same points which make the 7-pt stencil. The edges of the upper and lower planes have a weight of $\frac{1}{32}$ - the same as the corners of the middle plane, i.e. the plane containing the central point. The upper and lower plane corner points contribute $\frac{1}{64}$ of their value. A 19-pt stencil in 3-D can be constructed by not considering the corner points on the upper and lower planes (and adjusting the weights accordingly).

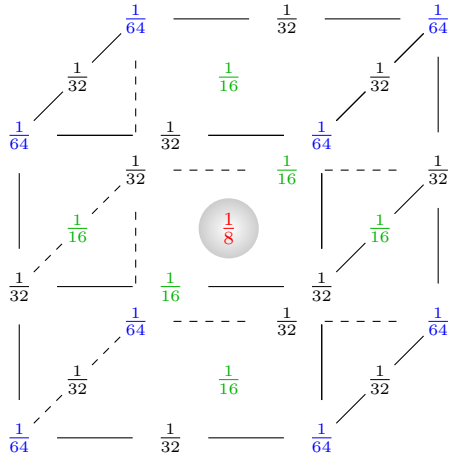


Figure 6.2: Full 27-point restriction weights in 3-D for the central point (red)

6.4.2 Interpolation or Prolongation

Table 6.1: Interpolation: operator in 2-D

Ω^h	Ω^{2h}
$v^h_{2i,2j}$	$v^{2h}_{i,j}$
$v^h_{2i,2j+1}$	$\frac{1}{2}(v^{2h}_{i,j} + v^{2h}_{i,j+1})$
$v^h_{2i+1,2j}$	$\frac{1}{2}(v^{2h}_{i,j} + v^{2h}_{i+1,j})$
$v^h_{2i+1,2j+1}$	$\frac{1}{4}(v^{2h}_{i,j} + v^{2h}_{i+1,j} + v^{2h}_{i,j+1} + v^{2h}_{i+1,j+1})$

The error e^{2h} approximated on Ω^{2h} after solving $A^{2h}e^{2h} = r^{2h}$ (error equation) must be transferred back to grid Ω^h. Clearly, since the number of mesh points on Ω^h and Ω^{2h} are different, e^{2h} cannot simply be mapped to Ω^h. An interpolation/prolongation operator (I^h_{2h}) is used for this purpose. I^h_{2h} acts on coarse grid error vectors (e^{2h}) to produce a fine grid vector (e^h) i.e. $I^h_{2h}e^{2h} = e^h$. In two dimensions, when considering a point $e^h_{i,j}$ on Ω^h, the operator I^h_{2h} takes into account different weights for different points on Ω^{2h}. A standard interpolation scheme for 2-D is shown in Table 6.1.

In stencil notation the *linear interpolation* operator in 2-D is shown below:

– *Linear* interpolation:

$$\frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}^h_{2h}.$$

A table similar to Table 6.1 can be drawn for the 3-D case of interpolation. The 3-D *Trilinear* Interpolation operator is shown in Table 6.2. When full Restriction and Linear Interpolation are used, it can be stated that $I^h_{2h} = c(I^{2h}_h)^T$, where $c \in R$. That is, the interpolation operator is the transpose of the restriction operator up-to a certain constant c, which depends on the spatial dimension. This is known as the *variational* property [63].

Table 6.2: Trilinear Interpolation: operator in 3-D

Ω^h	Ω^{2h}
$v^h_{2i,2j,2k}$	$v^{2h}_{i,j,k}$
$v^h_{2i+1,2j,2k}$	$\frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k})$
$v^h_{2i,2j+1,2k}$	$\frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i,j+1,k})$
$v^h_{2i,2j,2k+1}$	$\frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i,j,k+1})$
$v^h_{2i+1,2j+1,2k}$	$\frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i,j+1,k} + v^{2h}_{i+1,j+1,k})$
$v^h_{2i+1,2j,2k+1}$	$\frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i+1,j,k+1})$
$v^h_{2i,2j+1,2k+1}$	$\frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i,j+1,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i,j+1,k+1})$
$v^h_{2i+1,2j+1,2k+1}$	$\frac{1}{8}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i,j+1,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i+1,j+1,k} + v^{2h}_{i+1,j,k+1} + v^{2h}_{i,j+1,k+1} + v^{2h}_{i+1,j+1,k+1})$

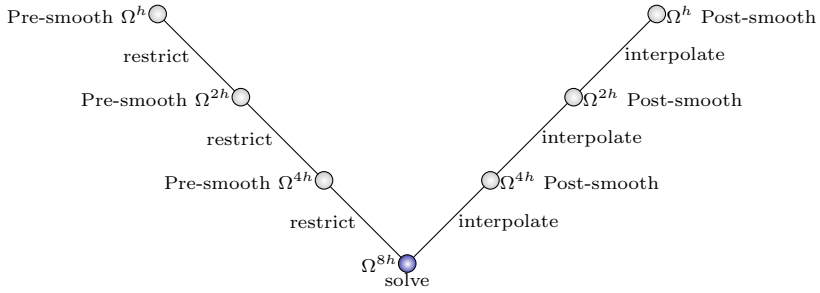


Figure 6.3: V-cycle in Multigrid

Require: Initial solution approximation: v^h, RHS: f^h on fine grid: Ω^h
1: **while** not converged or completed fixed iterations **do**
2: $i \leftarrow 1$
3: **while** $\Omega^{ih} \neq$ Coarsest grid **do**
4: Pre-smooth ν_1 times on $A^{ih}v^{ih} = f^{ih}$ to obtain new v^{ih}
5: $f^{2ih} \leftarrow I_{ih}^{2ih}(f^{ih} - A^{ih}v^{ih})$
6: $v^{2ih} \leftarrow 0$
7: $i \leftarrow i \times 2$
8: **end while**
9: Solve $A^{ih}v^{ih} = f^{ih}$ exactly to obtain new v^{ih}
10: $i \leftarrow \frac{i}{2}$
11: $v^{ih} \leftarrow I_{2ih}^{ih}v^{2ih} + v^{ih}$
12: **while** $\Omega^{ih} \neq$ Finest grid **do**
13: Post-smooth ν_2 times on $A^{ih}v^{ih} = f^{ih}$ to obtain new v^{ih}
14: $i \leftarrow \frac{i}{2}$
15: $v^{ih} \leftarrow I_{2ih}^{ih}v^{2ih} + v^{ih}$
16: **end while**
17: Post-smooth ν_2 times on $A^{ih}v^{ih} = f^{ih}$ to obtain new v^{ih}
18: **end while**

Figure 6.4: Multigrid Algorithm $v^h \leftarrow MG(v^h, f^h)$

6.4.3 Multigrid Algorithm

The Multigrid algorithm can be broken down into its constituent parts, namely: Smoothing, Residual calculation, Restriction, Interpolation and Error correction. A *V-cycle* consisting of all these steps is illustrated in Figure 6.3. As explained in Section 6.3.1 above, the V-cycle can be used to obtain the solution of the discretized PDE. We next describe the Multigrid algorithm formally as illustrated in Figure 6.4.

The Multigrid procedure begins by using the initial guess v^h and the right hand side term f^h at the finest grid as the input to the procedure $MG(v^h, f^h)$. Then according to the V-cycle (see Figure 6.3), ν_1 smoothing operations of an iterative method are carried out to obtain a new approximation of v^h using $A^h v^h = f^h$. In our implementations we use the weighted Jacobi

method as the smoother on a mixed boundary value problem. The Restriction operator I_h^{2h} is used next on the residual $f^h - A^h v^h$ to obtain the RHS for the next immediate coarser grid Ω^{2h}. This process is carried out till we reach the coarsest grid. The coarsest grid error equation is then solved exactly and the error is interpolated back to the immediate finer grid. After using the error on the coarser grid to obtain a better approximation on the immediate finer grid, ν_2 post-smoothing operations are carried out to again obtain a better approximation. The solution is again interpolated to the next immediate finer grid and the process continues till we reach the finest grid. ν_2 post-smoothing operations are carried out at the finest grid level after which the solution is tested for convergence or if a fixed number of cycles have been completed.

6.5 Terminology and Problem Description

This section introduces the notation and assumptions on which our model is based, and gives a brief low-level description of the problem under consideration. This is followed by a description of the test problem that we use for our experiments.

6.5.1 Notation Recap

A structured 3-D grid having dimensions $N_x N_y N_z$ can be divided among P parallel processes running on individual cores in several ways. In general, D_i represents the number of processes along direction i where $i = x, y, z$. Thus, P can be decomposed as any valid permutation of D_x, D_y and D_z such that $P = D_x D_y D_z$, and for simplicity, we assume that $N_i \% D_i = 0$ for $i = x, y, z$. In the following we consider cuts/partitions parallel to the Cartesian axes and the model assumes a 7-pt iteration stencil with a 1-element deep ghost zone. Each sub-domain with a single element deep ghost/halo zone has dimension $(P_x + 2)(P_y + 2)(P_z + 2)$. The 3-D sub-domain on each core can be viewed as 3 parts: the inner *Independent Computational* (IC) kernel which needs no data from neighbouring processes (zone 1), the next-to-boundary layer (*Dependent Planes*) which requires data from neighbouring processes for its update (zone 2) and the *buffer/ghost/halo* region (zone 3) [138]. Thus, in the worst case, a sub-domain will need to pass six planes to its nearest neighbour processes. Without loss of generality we assume that the unit stride dimension is in the Z-direction (depth) and the data is in row-major order as in the C language. It can be noted that four of the six nearest data neighbours in the 7-pt stencil are not contiguous in memory. We collectively refer to the two YZ planes as X-planes, the two XY planes as Z-planes, and the two XZ planes as Y-planes.

6.5.2 Brief Description of the Problem

This division of P as $D_x D_y D_z$ can have a large effect on the packing/unpacking times of data which is to be sent to/received from the neighbouring sub-domains, the update of Dependent planes, and the compute times of the Independent Compute kernel. There are generally

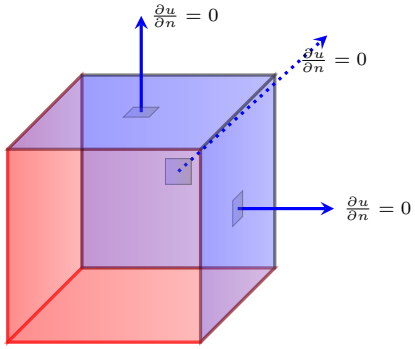
several permutations of D_x, D_y and D_z which satisfy $P = D_x D_y D_z$. We refer to a valid permutation as a Topology or a Process Topology (MPI Cartesian Process Topology [48]). For example, a total of 28 Process Topologies exist for $P = 64$, three of which (for example) are, $D_x \times D_y \times D_z = 4 \times 4 \times 4$, $D_x \times D_y \times D_z = 4 \times 16 \times 1$ and $D_x \times D_y \times D_z = 8 \times 4 \times 2$. These process topologies decide the sub-domain data dimensions of the hierarchy of grids. Typically and traditionally, the topology which minimizes the communication volume to be sent, created by the default `MPI_Dims_create()`, is chosen as the preferred topology for domain partitioning/mesh partitioning. We investigate the optimality of partitions returned by `MPI_Dims_create()` and whether only minimizing communication is sufficient to obtain optimal sub-domain dimensions for parallel GMG. Our work in Chapter 4 demonstrated the dependence of domain partitioning for single grids on cache-misses in computation and communication. Since parallel GMG is significantly more complex than a single grid and incorporates further stencil operators, the current chapter examines the efficacy of extending the model to parallel GMG.

6.5.3 Test Problem

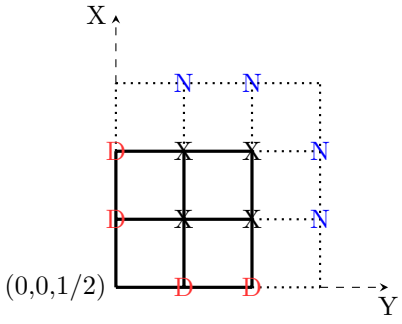
Parallel GMG was implemented for a 3-D *mixed Dirichlet-Neumann* boundary value problem on a unit cube to solve $-\nabla^2 u = -\frac{3\pi^2}{4} \sin\frac{\pi x}{2} \sin\frac{\pi y}{2} \sin\frac{\pi z}{2}$, which is a linear, second order, inhomogeneous PDE with constant coefficients, having a smooth solution $u(x, y, z) = \sin\frac{\pi x}{2} \sin\frac{\pi y}{2} \sin\frac{\pi z}{2}$. We use a vertex-centered finite-difference scheme in our implementation. Dirichlet boundary conditions ($u = 0$) are applied to the X=0, Y=0 and Z=0 faces of the cube whereas Neumann boundary conditions ($\frac{\partial u}{\partial n} = 0$) are applied at the X=1, Y=1 and Z=1 faces. A halo layer (or the Neumann boundary ghost layer) is added to the Neumann boundaries as the boundary values are also considered as unknowns [22]. These halo layers need to be updated at each iteration according to the neighbouring point inside the physical sub-domain using a central difference approximation. Figure 6.5a shows the Dirichlet-Neumann boundaries on the boundary of the domain $\partial\Omega$ specified on a unit cube, Ω. Figure 6.5b shows a 2-D plane of Figure 6.5a when the unit cube is cut at $Z = \frac{1}{2}$. For a $2 \times 2 \times 2$ problem size, the number of unknowns in the X,Y and Z direction are two each, making it a total of 8 unknowns. Four unknowns out of these eight illustrated as cross-points i.e. 'x' can be seen in Figure 6.5b. The Dirichlet boundaries on the left plane and bottom plane are denoted by 'D', while the fictitious Neumann ghost points are denoted by 'N'. Clearly, $\partial\Omega = \Gamma^D \cup \Gamma^N$ and $\Gamma^D \cap \Gamma^N = \phi$.

If we consider a mesh point $u_{b,j,k}$ on the upper Neumann boundary, we can denote the corresponding ghost boundary/fictitious Neumann ghost point (outside the domain) as $u_{b+1,j,k}$ and the downward vertical neighbour point as $u_{b-1,j,k}$. Since the derivative of the outward normal $\frac{\partial u}{\partial n} = 0$ at the Neumann boundary, we can approximate $u_{b+1,j,k}$ using a central difference scheme (second order approximation) by equating

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial x} = \frac{u_{b+1,j,k} - u_{b-1,j,k}}{2h} = 0$$



(a) Dirichlet (Left, Bottom, Front - in red) and Neumann (Top, Back, Right - in blue) boundaries with outward normals on a unit cube



(b) A plane at Z=1/2 depicting Dirichlet (D), Neumann (N) boundary, unknowns (X), given problem (solid lines), halo layer for Neumann boundary (dotted lines).

Figure 6.5: Dirichlet-Neumann mixed Boundary Value Problem

where h is the mesh spacing. Thus, for our problem $u_{b+1,j,k} = u_{b-1,j,k}$. Similar approximations can be carried out in the Y and Z directions as well. It is important to update the Neumann boundary conditions before updating the solution in the interior and also before calculating the residual.

A full 27-pt Restriction scheme was implemented to transfer data to the coarser grid. This needs more than one communication step to make the corner points available to a process. If we visualize eight processes arranged as an MPI Cartesian Topology of $2 \times 2 \times 2$, then the process on the lower left front corner would require data from all the other processes. In a general case where the process is surrounded by other processes and does not touch any domain boundary, it would require data from all 26 of its neighbours. Therefore, the communication pattern for a 27-pt stencil in the Restriction and Interpolation operator is different from that of the 7-pt stencil used in the smoother. The case of Trilinear Interpolation onto the finer grid is similar. Care should be taken to modify the stencil while carrying out restriction at the intersection of Neumann-Neumann boundaries [25]. An unmodified stencil results in an error in the smoother which leads to a severe deterioration of the Multigrid convergence rate. The reason is that the residual at the Neumann boundary ghost points is zero to begin with and since the 27-pt restriction takes into account the residual at these points, we do not obtain correct scaling at the coarse grid level. Thus, in our scheme instead of modifying the stencil at these points, we copy the residual of the point that is used to determine the value of the ghost Neumann boundary point to the corresponding point on the fictitious Neumann boundary. This allows us to use the same stencil at the Neumann-Neumann boundaries as now the fictitious Neumann boundary point contributes equally as the corresponding point inside the physical domain.

At the finest grid level, the l_2 norm of the residual can be calculated after each V-cycle and the execution stops when the ratio of the current norm to the initial norm becomes less than a specified tolerance i.e. $\frac{||r_k||}{||r_0||} < TOL_N$. However, for performance analysis purposes, it is sufficient to fix the number of V-cycles. The levels are numbered from the highest to the lowest - starting at the finest grid (level L) to level zero corresponding to the coarsest grid. The coarsest grid problem can be solved till convergence (or a fixed number of iterations can be performed depending on the experiment). The iterative scheme used is ω-Jacobi, with the option to change the weighting factor (ω) for both smoothing (fine and coarser grids) and solve (coarsest grid) operations. The general optimum values of ω for 1-D, 2-D, 3-D are $\frac{2}{3}$, $\frac{4}{5}$ and $\frac{6}{7}$, respectively (for pure Dirichlet boundaries) [25] but for our mixed Dirichlet-Neumann test case we found $\omega = 1$ to be optimal.

Although the number of unknowns per process is equal, the problem is slightly load-imbalanced because the processes containing the Neumann boundary have to perform more work than processes containing the Dirichlet boundaries. This is because in addition to the points to be updated, the former category of processes must also adjust the Neumann boundary before the values of the boundary points can be updated. However, such processes do not send/receive planes to/from other processes at the Neumann boundary. Thus, we expect that the increase in the computational work at such processes is evenly balanced out by the zero communication overhead and such processes do not govern the overall computational complexity. The number of smoothing operations in the downward phase of the V-cycle is ν_1 (pre-correction smoothing) and ν_2 on the up-cycle (post-correction smoothing). The complete V-cycle is then written as $V(\nu_1, \nu_2)$ where typically we use $\nu_1 = \nu_2 = 3$.

6.6 Cache-Misses Minimization Model

Our work in Chapter 4 (and [138]) exhaustively identified and quantified cache-misses as the single most important factor influencing domain partitioning of structured single level grids and thus, while extending the model in this section, our focus remains on the cache-misses in the update/packing/unpacking of the Dependent Planes and the update of the Independent Compute kernel. We further elaborate on the super-set of factors influencing cache-misses directly or indirectly to shed light on the complexity of attaining truly optimal sub-domain dimensions for high performing partitions in Parallel GMG. It is to be noted that our high level model is different from the *analytical* models used to model Multigrid cycle times and performance. Classical analytical models have attempted to model the execution timing and analyze the overall Weak Scaling using only the relaxation phase of *semi-coarsening* Multigrid with 1-D, 2-D and 3-D processor topologies/partitions [145, 146]. A baseline model with penalties in parallel settings has been formulated for modelling the cycle of Algebraic Multigrid in [146, 147]. An analytical/empirical comparison for the execution times of an iteration of *Newton-Multigrid* and *FAS* (Full Approximation Scheme) has been carried out in [77]. Performance prediction of

Multigrid codes on large numbers of cores by *benchmarking* the code on a very small number of processes presents another alternative [148]. Most of these models take into account only the algorithmic characteristics and not the hardware parameters. Our model is different from these in the sense that we take into account the cache-line characteristics but obtain a cache-oblivious result, thus leading us to a quasi-cache-aware model [141]. Further, our model does not predict the execution timings but predicts the topologies which outperform the standard `MPI_Dims_create()` topology.

The following section extends the cache-misses minimizing model that was created in Chapter 4 to GMG. Similar to the approach used in Chapter 4, we begin by considering a *Poisson* equation discretized using the Finite Difference Method that uses a 7-pt stencil for updating the solution at the mesh points. First we quantify the cache-misses for the Dependent Planes and then deal with the Independent Compute. This quantification is more general in the sense that as opposed to Chapter 4,

1. a *source* term f on the RHS is present,
2. a double precision data type `double` is used,
3. instead of the unweighted Jacobi, we use the ω-Jacobi iterative method,
4. both Dirichlet as well as Neumann Boundaries are present.

6.6.1 Extending the Model

We consider an *Elliptic, linear* PDE: $\nabla^2 U = F$. The discretized form is $Au = f$, with A being the discretization matrix and u representing the vector of unknowns. The key component of the smoothing phase of Multigrid consists of an iterative method such as the "out-of-place" weighted Jacobi ($\omega-$Jacobi) shown in Equation (6.1) below:

$$v_{i,j,k} = (1-\omega)u_{i,j,k} + \omega(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1} + h^2 f_{i,j,k}) \tag{6.1}$$

The *Red Black Gauss-Seidel (RBGS)* updates "in-place", however, the observations that we make will still hold in principle (though with appropriate quantitative differences). The advantage of RBGS is that the local working set consists of only two arrays which reduces the memory traffic and the cache conflict misses. The disadvantage of RBGS is that the red and black points are communicated separately and hence it requires twice the message exchanges as ω-Jacobi, resulting in twice the latency of messages as a penalty. The worst case for Neumann updates occurs at the top back right boundary process which has three Neumann boundaries. For this process, the cache misses for updating the three boundaries in the X, Y and Z-direction are $\frac{P_y P_z}{\mathcal{W}}$, $\frac{P_x P_z}{\mathcal{W}}$ and $P_x P_y$ (here $\mathcal{W} = 8$), respectively. It is to be noted that while updating the Neumann boundaries, both the read and write arrays are the same. However, the planes which

Table 6.3: Predicted Cache-Misses: Cache read/write/update misses for the dependent X, Y and Z-plane

Plane	**Read Misses**			**Write Misses**		**Total**
	Pack	Update	RHS	Unpack	Update	
Z-plane	P_xP_y	$5P_xP_y$	P_xP_y	P_xP_y	P_xP_y	$9P_xP_y$
X-plane	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{5P_yP_z}{\mathcal{W}}$	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{P_yP_z}{\mathcal{W}}$	$\frac{9P_yP_z}{\mathcal{W}}$
Y-plane	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{5P_xP_z}{\mathcal{W}}$	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{P_xP_z}{\mathcal{W}}$	$\frac{9P_xP_z}{\mathcal{W}}$

undergo Neumann updates are not communicated to any other process and nor do processes containing the Neumann boundary receive data from other processes (at this boundary). Thus, the packing/unpacking cost of such planes is zero. Since the sum of cache-misses for packing and unpacking planes is more than that of Neumann updates for the plane, we can safely consider processes which send and receive data from other processes to derive the upper bound for cache misses. Such a process does not touch any boundary and sends/receives all six planes to/from neighbouring processes.

Assuming that the cache-line length is $\mathcal{L}$ bytes and the width of a `double` element is $\mathcal{D}$, the number of elements fetched from the memory to the cache are $\mathcal{W} = \frac{\mathcal{L}}{\mathcal{D}}$. For example, for the systems used here $\mathcal{L} = 64$ bytes and $\mathcal{D} = 8$ bytes and thus $\mathcal{W} = \frac{\mathcal{L}}{\mathcal{D}} = 8$. Assuming a minimal number of cache-lines for accommodating the six different read streams (and one write stream) in Equation (6.1) and disregarding the loop invariant terms, namely, ω and h^2 (square of mesh spacing), the cache-misses for update/packing/unpacking of Dependent Planes (S_P) can be summarized in Table 6.3. Table 6.3 is similar to the total cache-misses table in Chapter 4 but differs in the fact that a `double` data type is used here. The procedure for calculating the cache-misses is exactly the same as in Section 4.4.3. The cache-misses for the Independent Compute are calculated in the same way as in Section 4.4.4. As mentioned previously in the case of uniform single grids, the `double` data type does not affect the derivation or the inferences from our model. The derivation and the result here further confirm this fact.

The total cache-misses of the Independent Computation (S_I) kernel can be calculated as:

$$\begin{aligned} S_I &= (P_x-2)(P_y-2)(P_z-2)(\frac{5}{\mathcal{W}}+\frac{1}{\mathcal{W}}+\frac{1}{\mathcal{W}}) \\ &= (P_x-2)(P_y-2)(P_z-2)(\frac{5}{8}+\frac{1}{8}+\frac{1}{8}), \end{aligned} \tag{6.2}$$

where the $\frac{5}{8}$, $\frac{1}{8}$ and $\frac{1}{8}$ terms give the read misses in the update, write misses in the update and right hand side term read misses, respectively. The total cache misses ($S = S_I + S_P$) for the

Independent Computation and Dependent Planes are:

$$S = \gamma(P_x - 2)(P_y - 2)(P_z - 2) + \alpha P_x P_y + \beta P_z(P_x + P_y), \tag{6.3}$$

where $\gamma = \frac{7}{8}$, $\alpha = 9$ and $\beta = \frac{9}{8}$ (see Table 6.3 and $\mathcal{W} = 8$) and are dependent on the computational kernel and the length of the cache-line.

We now consider a Multigrid V-cycle with $L + 1$ levels, where the level $k = 0$ denotes the coarsest grid and $k = L$ the finest grid. We assume the following:

1. The costliest operation is Smoothing.
2. The cost of applying a single grid transfer operator is proportional to a single Smoothing operation.
3. The cost of a solve on the coarsest level may be neglected compared to the fine grid smoothing cost.

Let the cache-misses at level k be denoted by S_k where $S_k = S$ at level $k = L$ as in Equation (6.3). The sum of cache-misses at all levels (S_T) is bounded above by S_∞, where

$$S_T = \sum_{k=0}^{L} S_k < S_\infty = \sum_{k=0}^{\infty} S_k. \tag{6.4}$$

Summing two separate infinite geometric series with common ratios $\frac{1}{8}$ and $\frac{1}{4}$ yields the expression for S_∞ as shown in Equation (6.5) below:

$$S_\infty = \frac{8\gamma}{7}(P_x - 2)(P_y - 2)(P_z - 2) + \frac{4}{3}(\alpha P_x P_y + \beta P_z(P_x + P_y)). \tag{6.5}$$

By considering $\frac{\partial S_\infty}{\partial P_x} = \frac{\partial S_\infty}{\partial P_y} = 0$ to minimize the total cache-misses with respect to sub-domain dimensions, we obtain $P_x = P_y$ for optimality (condition one) but this does not yield any information regarding P_z. Since we can generate all D_x, D_y, D_z subject to $P = D_x D_y D_z$ and because P_x, P_y and P_z are only dependent on N and D_x, D_y, D_z, we can exhaustively find that $D_z = 1$ minimizes S_∞ by substituting these values of P_x, P_y, P_z in the equation for S_∞. Thus, cache misses are minimized by maintaining a balance between the X/Y dimensions of the sub-domain and maintaining an unaltered unit stride dimension (theoretically). Further, the communication minimizing condition to minimize the surface area of planes implies $P_x = P_y = P_z$ (condition two). Taking the intersection of the cache-misses and communication volume minimization conditions yields a strong (common) condition: $P_x = P_y$. Further, when $S_I >> S_P$ in Equation (6.3), S is minimized with $P_x = P_y = P_z = \frac{N}{P^{\frac{1}{3}}}$. These two limits i.e. cache-miss dominated and communication volume dominated imply that $1 \leq D_{z_{optimal}} \leq P^{\frac{1}{3}}$ (assuming P is a perfect cube).

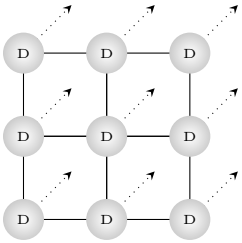


Figure 6.6: Front 2-D view of nine data-streams indicated by a 'D' in a 27-pt stencil in 3-D, dotted lines and arrows show direction in which data is contiguous

6.6.2 Data Streams and Inter-grid Operators

A contiguous set of mesh data points forms a data-stream. As an example, if we consider the 7-pt stencil then there are five separate data-streams in the read array u for ω-Jacobi (see Equation (6.1)). We can also interpret the data-stream as a set of points which belong to the same cache-line when they are fetched. Thus, when considering the mesh points $u_{i,j,k}$ and $u_{i,j,k\pm1}$, all three form part of the same data-stream (if index k is in the unit-stride direction). Figure 6.6 shows nine data streams in the read array for a 27-pt stencil in 3-D. Thus, although there are 27 mesh points to be considered, they can be grouped into nine groups, each having 3 contiguous data points. Whenever any data point in any data-stream is brought into the cache-memory, the other points associated with the same data-stream also form part of the cache-line fetched into the cache memory. There is a possibility that the total number of elements used in higher order stencils is larger than the length of a cache line. In such a case our definition of a data stream does not hold. Discussion of such higher order stencils is beyond the scope of the current work.

6.6.2.1 Restriction

After the residual is calculated i.e. $r^h = f^h - A^h v^h$, the residual is restricted using a 27-pt stencil. We can approximate the number of cache-misses for every point on the coarse grid. Since for each point in the coarse-grid, we need to consider 27 points on the fine grid, we do span the entire fine grid sub-domain as data-points cannot be selectively brought into the cache if they are part of the same cache-line. Assuming enough cache-lines for nine data-streams, there is approximately a single cache-miss for each data-stream after every 8 `double` elements. Thus, we can approximate the number of cache-misses to $\mathcal{R} = \frac{9(P_x+2)(P_y+2)(P_z+2)}{8} \approx \frac{9P_xP_yP_z}{8}$. This quantity has the same form as the expression for Independent Compute kernel cache-misses i.e. $\mathcal{R} = kS_I$ for some constant k. This validates our assumption above where we state that the cost of grid transfer operators is proportional to the smoothing operator. It is to be noted that the smoothing operator is applied $\nu_1 + \nu_2$ times whereas the Restriction operator is only applied once per level per V-cycle.

6.6.2.2 Interpolation

After solving the error equation on the coarsest grid, the error is interpolated to the immediate finer grid and this process continues through to the finest grid. The number of mesh points utilized by the Interpolation operator is different from that of the Restriction operator. Table 6.2 shows that depending on whether the mesh point coordinates are odd or even, a different number of coarse grid points are used while interpolating. In the worst case, Trilinear Interpolation can use up to 8 points of the coarse grid (see last row of Table 6.2). These eight points form eight vertices of a cube and hence are equivalent to 4 data streams. We can then create an upper bound on the number of cache-misses by assuming that each point on the fine grid uses 8 points on the coarse grid for interpolation. Since the number of `double` elements in a single cache line is eight, each of the 4 data streams fetches up to 8 double elements of their respective data stream. Each of the 4 data streams on the coarse grid causes a cache-miss after every 8 elements are interpolated on the fine grid. Thus, there are 4 cache-misses for every 8 elements of the fine grid and this gives rise to approximately $\mathcal{I} \approx \frac{4}{8} P_x P_y P_z = \frac{1}{2} P_x P_y P_z$ cache-misses. Clearly it can be seen that $\mathcal{I} = KS_I$ for some constant K. Thus, this shows that the cache-misses associated with Restriction i.e. $\mathcal{R}$ and Interpolation i.e. $\mathcal{I}$ are proportional to the Independent Compute cache-misses (S_I).

6.6.3 Pruning the Topology Search Space

Out of all the topologies which are possible, a small set can be examined keeping in mind the balance between the X and Y sub-domain i.e. P_x and P_y dimensions and the minimization of D_z. Thus, if P is a perfect square then $D_z = 1$ and $D_x = D_y = \sqrt{P}$ else we find $min(|D_x - D_y|)$ such that $D_x D_y = P$. To alleviate the effect of process placement we introduce a factor ρ that represents the deviation from the balanced pair of (D_x, D_y) i.e. assuming $P = 64$ and $\rho = 1$, we start with $D_x = 8$, $D_y = 8$ and $D_z = 1$ and then consider $(8 \times 2^1) \times (\frac{8}{2^1}) \times 1 = 16 \times 4 \times 1$ and $(\frac{8}{2^1}) \times (8 \times 2^1) \times 1 = 4 \times 16 \times 1$. For $\rho = 2$, we would also consider $(8 \times 2^2) \times (\frac{8}{2^2}) \times 1 = 32 \times 2 \times 1$ and $(\frac{8}{2^2}) \times (8 \times 2^2) \times 1 = 2 \times 32 \times 1$. In practice our experiments show that $\rho = 1$ is sufficient for obtaining optimal topologies.

6.6.4 Factors affecting sub-domain dimensions

To place the above model in its true context, we now discuss all of the factors influencing selection of sub-domain dimensions (assuming the data streams at no point are too large to fit into the cache). We discuss their impact in isolation and with respect to other factors. The discussion primarily brings out the need for a fine balance between multiple factors for optimizing the domain partitions and sheds light on their interplay. That is, that the problem of domain partitioning is much more subtle than just minimizing the communication.

Independent Compute (IC): This represents the sub-domain zone that does not need data

from other processes for updating the mesh points. To update the solution at all mesh points contained in a plane, three planes i.e. the plane under consideration and the two planes immediately above and below it are needed for a 7-pt stencil. The smaller the total size of these 3 planes, the more is the probability that they would fit into the Last Level Cache (LLC)/Cache-hierarchy. We define the quantity *Working Plane Set Size* (WPSS) as $3 \times (P_y + 2) \times (P_z + 2) \approx 3P_yP_z$ elements. The Independent Compute (IC) tries to minimize the WPSS by minimizing P_y but not P_z as the latter adversely affects the Vectorization and prefetch efficiency. Thus, it is preferable to decrease P_y rather than reducing P_z to decrease the overall WPSS. But when P_y is decreased (or D_y is increased as $P_y = \frac{N}{D_y}$) to some value $<< P_x$, it violates the cache-minimizing condition ($P_x = P_y$) which in turn leads to much higher communication and update times for the Y-plane that contains P_xP_z elements. Ideally, the MPI implementation should hide the entire communication cost behind the cost of executing the IC kernel. Practically, this is never the case as the computation and packing/unpacking of planes is carried out by the same thread or process (assuming no separate core for communication exists) that may result in switching between the two tasks: computation and packing/unpacking. This switching may also lead to an increased cache-contention and conflict misses as different data streams from computation and packing/unpacking are brought into the cache. In summary, decreasing P_y optimizes the execution time for the Independent Compute (IC) but increases the transmission times of the Y-plane. Further, when $P_y << P_x$, both the communication volume and cache-minimization conditions are violated.

Communication Volume (V): Minimizing communication implies $P_x = P_y = P_z$. For simplicity of discussion we assume the number of processes P is a perfect cube and thus for a domain of size N^3 this implies that the Cartesian process dimensions $D_x = D_y = D_z = P^{\frac{1}{3}}$ and $P_x = P_y = P_z = \frac{N}{P^{\frac{1}{3}}}$. Since the default `MPI_Dims_create()` returns $D_{sx} \geq D_{sy} \geq D_{sz}$, considering the equality condition, the worst case growth rate of the Z-plane size becomes $P^{\frac{1}{3}}$, leading to an increase in its communication and update time. This can be seen by assuming P to be a perfect square (in addition to being a perfect cube) and noticing that our cache-minimizing model yields $D_x = D_y = P^{\frac{1}{2}}$ with $D_z = 1$. Thus, the size s_1 of the Z-plane according to our model becomes $s_1 = P_xP_y = \frac{N}{P^{\frac{1}{2}}} \times \frac{N}{P^{\frac{1}{2}}} = \frac{N^2}{P}$ and the corresponding value considering the default MDC $s_2 = \frac{N}{P^{\frac{1}{3}}} \times \frac{N}{P^{\frac{1}{3}}} = \frac{N^2}{P^{\frac{2}{3}}}$. The Z-plane then grows as $\frac{s_2}{s_1} = P^{\frac{1}{3}}$. Thus, the maximum performance difference (theoretically) between the cache-minimizing topology and the standard MDC occurs when the number of processes is both a perfect cube and a perfect square. Our model shows that $1 \leq D_{zoptimal} \leq D_{sz}$ and hence minimizing only the communication volume is insufficient.

Prefetch: For any topology, updating the Independent Computation (IC) kernel involves multiple contiguous data streams and thus prefetch hides the latency. Since prefetch usually exploits spatial locality and assumes streaming fetches, maximizing P_z should increase the

utilization of the prefetched cache lines. The L1d cache has two hardware prefetchers in the Intel *Sandy Bridge* architecture present on ARC2. The first one, the *Data Cache Unit* (DCU) prefetcher, prefetches data in an ascending order from the address which has most recently been loaded. Thus, assuming an address A has been loaded (and each address can contain a `double` value) i.e. a cache-line is populated by `double` elements from A to A+7, the DCU prefetches the data from A+8 to A+15 in another cache line. The second prefetcher, the *Instruction Pointer (IP)-based stride prefetcher*, detects the stride in different load instructions and prefetches a cache line from the current address which is the sum of the current address and the stride. A stride of up to 2KB can be detected (or equivalently 256 `double` elements) [149]. The two prefetchers that bring data into the L3 cache are called the *Streamer* and the *Spatial Prefetcher*. The data may not always be brought into the L2 cache due to pending read/write misses. The Spatial Prefetcher fetches an additional 64 bytes into the unified L2 cache when a cache-line is brought into L2. The Streamer monitors cache misses from L1d, hardware prefetch requests from L1d and L1i Instruction cache requests, and can maintain up to 32 streams of ascending/descending data [149]. Thus, most prefetchers depend on contiguous data streams and hence it is important that P_z is maximized to minimize the presence of ghost data elements. This is inherently connected to *Cache Line Utilization* (CLU) which is explained later in this section. More specifically, the update of the Independent Compute and the packing/unpacking/update of the X/Y Dependent Planes can benefit from maximizing P_z. It is important to notice that the packing/unpacking/update of the Z-plane does not benefit from maximizing P_z. In summary, efficient prefetching demands maximizing the value of P_z. However, this condition violates the volume minimizing condition and increases the WPSS (Working Plane Set Size). With an increase in the WPSS, there is a danger that the three planes required for the update of a single plane may not fit into the LLC (Last Level Cache)/Cache-hierarchy.

Least Recently Used (LRU) Eviction: This cache eviction policy replaces the cache-lines which have not been used recently. The distance between the mesh point $u_{i,j,k}$ and $u_{i,j+1,k}$ (see Equation (6.1)) is $P_z + 2$ and typically for a large enough P_z, these mesh points will belong to a different cache-line. Thus, the larger the value of P_z, the greater the clock cycles elapsed between re-accessing/re-using the mesh point $u_{i,j+1,k}$ to update $v_{i,j+1,k}$. This translates to having a higher probability for the eviction of this cache-line before it is re-used when P_z increases. This factor is different from all the other factors in the sense that it requires minimization of P_z to achieve maximum efficiency.

Planes Cache Misses: To minimize the cache-misses in packing/unpacking/updating planes our model indicates that $P_x = P_y$ and $1 \leq D_{zoptimal} \leq D_{sz}$. This indicates a partition which is close to a 2-D partition. As discussed above, when P_z is large, the Least Recently Used (LRU) policy used to evict cache-lines negatively affects the re-use of a cache-line. Further, increasing P_z increases the product $3P_yP_z$ (WPSS), possibly causing the combined size of 3 planes

required to update a plane to become larger than the cache capacity. It is to be noted that the *Effective* WPSS (EWPSS) evaluates to $5P_yP_z$ as it involves 3 planes of the array u and one plane each from the arrays v and f (see Equation (6.1), Section 6.6.1). Further, when $P_z \geq 256$ `double` elements, the IP-based stride prefetcher of the L1d cache is rendered ineffective. Thus, when packing/unpacking/updating a Z-plane for a sub-domain that has $P_z \geq 256$, neither the DCU nor the IP-based stream prefetcher are effective - resulting in increased cache-misses.

Cache Line Utilization (CLU): We define this to mean the fraction of data elements used in a cache line which has been fetched. Thus, $0 \leq$ Cache Line Utilization (CLU) ≤ 1. As an example consider the packing of an X-plane which has contiguous data (except for the near-to-boundary data points next to ghost/halo/boundary region). Consider a cache-line which fetches data elements far-away from the ghost regions. All the elements in this cache line will be used and hence the CLU = 1. But for a cache-line which has been prefetched/loaded containing the two ghost points (one following the back sub-domain Dependent Layer and one before the front Dependent Layer), the CLU = $\frac{6}{8} = 0.75$. Thus, theoretically if $P_z \longrightarrow \infty$, almost all cache-lines will have a CLU = 1 while packing the X-plane. The same is the case with the Independent Compute and packing/unpacking the Y-planes. The worst CLU is seen with the Z-plane. Assuming $P_z > 8 = \mathcal{L}$, where $\mathcal{L}$ denotes the cache-line length, the minimum CLU = 0 (for a cache-line that is prefetched after the cache-line containing the data element on the Z-plane) and the maximum CLU = $\frac{1}{8} = 0.125$ for packing the Z-plane. Thus, whereas increasing P_z increases the CLU for the IC and X/Y planes, it decreases it for the Z-plane. Even when the data completely fits into the cache hierarchy, accessing elements from a different cache-line incurs a penalty as compared to accessing the data from the same cache-line.

Vectorization: is a combination of loop unrolling and packed SIMD instructions - 256 bit AVX instructions in case of Intel Sandy Bridge architecture. These work on streaming data and thus, maximizing P_z is a step in this direction. With Independent Compute (IC), the ghost data acts as bubbles in the data stream, i.e. the ghost points are fetched as part of a cache line but are not used in the IC. The smaller the value of P_z and the larger the value of P_y, the greater will be the number of such junctions where ghost data forms a part of the cache line fetched. Thus, Vectorization demands a maximal P_z which again is in direct contradiction with the LRU policy discussed above and also deviates from the condition required for minimizing the communication volume.

The essence of our discussion on the multiple factors influencing sub-domain dimensions is summarized in Figure 6.7. The cache-misses minimization condition, particularly maximizing P_z, as derived in our model in Section 6.6.1 is re-enforced by many factors, namely, Independent Compute, Vectorization, Prefetch, Cache Line Utilization, Plane Cache Misses but is opposed by the LRU Eviction policy and the Communication Volume minimization condition. The

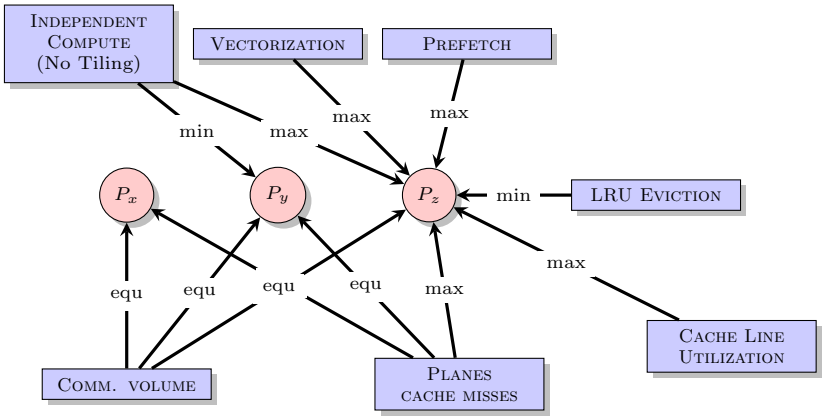


Figure 6.7: Factors affecting selection of sub-domain dimensions

Table 6.4: Trade-off: Theoretical Communication Volume Vs Predicted Z-plane Cache-Misses

Sub-domain Dims.			**Z-Plane**			**Communication**	
P_x	P_y	P_z	Size	Cache-misses	$n = 64$	Volume	$n = 64$
n	n	n	n^2	$9n^2$	36864	$6n^2$	24576
$\frac{n}{\sqrt{2}}$	$\frac{n}{\sqrt{2}}$	$2n$	$\frac{n^2}{2}$	$\frac{9n^2}{2}$	18432	$6.65n^2$	27266
$\frac{n}{\sqrt{4}}$	$\frac{n}{\sqrt{4}}$	$4n$	$\frac{n^2}{4}$	$\frac{9n^2}{4}$	9216	$8.5n^2$	34816
$\frac{n}{\sqrt{8}}$	$\frac{n}{\sqrt{8}}$	$8n$	$\frac{n^2}{8}$	$\frac{9n^2}{8}$	4608	$11.56n^2$	47364

Independent Compute opposes the equalization condition imposed on P_y by the Communication Volume and the Plane Cache Misses condition but minimizing P_y i.e. maximizing D_y may increase the communication costs as $|D_x - D_y|$ increases. The least constrained sub-domain dimension is the X-dimension i.e. P_x which needs to satisfy the condition $P_x = P_y$ as dictated by the Communication Volume and Plane Cache Misses conditions. We further emphasize that the major benefit in deviating from a cubic sub-domain shape can be attributed to the decreasing Z-plane packing/unpacking/updating cache-miss costs. At the same time, the increasing cost of the communication volume cannot be neglected when the unit-stride dimension (i.e. P_z) grows - even though the performance can increase due to the Vectorization, Prefetch and Cache Line Utilization (CLU). Table 6.4 shows the trade-off between increasing communication volume and decreasing cache-misses of the Z-plane. As the unit-stride dimension increases, the Z-plane cache-misses decrease at the expense of increasing communication volume. Thus, the decrease in cost due to the Z-plane cache-misses (most significant), improved Vectorization, Prefetch, and Cache Line Utilization must outweigh the cost of increased communication volume along with the extra cache-misses due to the LRU cache-eviction policy.

6.7 Dynamic Cache Tiling Heuristics

Cache tiling is one of the most important optimization techniques on modern microprocessor systems. The aim of cache tiling is to utilize the data maximally which already resides in the cache memory. Thus, if the computational sub-domain per process does not fit into the cache memory, small chunks of it are processed such that they fit into the memory. It is difficult to determine the tile shape and size even though cache-tiling is one of the most researched upon problems, with a vast amount of literature and research groups focussing on it [6,12,16,17]. For a 3-D sub-domain space, we focus on 2-D square tiles as outlined in [6] and leave the unit-stride dimension uncut. Our aim is to take away the burden of finding the optimal tile size from the application programmer and find a method for creating a light-weight, dynamic cache-tile size which can be calculated at run-time. We thus outline three heuristics designed to serve this purpose. It is to be noted that although we mention and use the L3 cache-per-core, they can be used for any cache i.e. L1, L2 or L3.

6.7.1 H_1: based on WSS

The first heuristic H_1 is based on the Working Set Size (WSS). We also evaluate this heuristic in the section describing the experimental results. To construct this we assume a square tile in the X and Y direction i.e. least unit-stride dimensions and leave the Z dimension (unit-stride dimension) uncut. As seen in Equation (6.1) the weighted Jacobi update uses three arrays. Thus, we assume that the L3 cache per core is equally divided between these three arrays. Since the tiling is done only for the read array, we equate the tile size to one-third of the L3 cache size. Stated more precisely, the total number of elements in the tile should be equal to the number of elements that can be held in one third of the L3 cache. Since the L3 cache is 2.5 MB per core for ARC2, the total elements of type `double` that can be contained in a third of it is $\frac{2.5\times1024\times1024}{3\times8}$. Thus, if the tile size is assumed to be k in the X and Y dimension but uncut in the unit-stride dimension, these number of elements can be equated to $k^2 \times (P_z + 2)$. This is further elaborated in the experimental section where we evaluate the performance of this heuristic against the optimal tile size, obtaining the latter being an extremely computational intensive and time consuming process.

6.7.2 H_2: based on number of working planes

The problem with H_1 is that we assume that we can divide the L3 cache equally between the three arrays used in Jacobi updates. This is not so as more elements of the read array are brought into the cache as compared to the write and RHS array (see Equation (6.1)). In this heuristic, we divide the L3 cache depending on the number of planes that are needed to update a single plane of the solution array. Thus, for a 7-pt stencil, 3 planes of the read array are needed, whereas only a single plane of each of the write and RHS array are needed. Thus, we can be more precise in the sense that we can allocate $\frac{3}{5}$ of the L3 cache to the read array for

which tiling is performed and allocate the remaining to the other two arrays. Thus, using H_2, we equate the tile size to $\frac{3\times2.5\times1024\times1024}{5\times8}$ elements.

6.7.3 H_3: based on Data Streams

A drawback of H_2 becomes visible when we consider and compare a 7-pt stencil and a 27-pt stencil. They both require three planes of the read array to update the write array. Thus using H_2 we do not get a different tile size for the 7-pt and 27-pt stencil although we expect that the fraction of the cache occupied by the read array will be larger in case of the 27-pt stencil. Thus, H_3 divides the L3 cache depending on the number of data-streams. For a 7-pt stencil, we have five different data streams in the read array and one data stream each for the write and RHS array. Thus, the fraction of cache occupied by the read array is $\frac{5}{7}$ of the total size. The number of data-streams for the 27-pt stencil is nine in the read array and (again) one each for both the write and RHS array. This logically allocates $\frac{9}{11}$ of the L3 cache to the read array and the tile size is equated to the number of elements contained in this fraction.

Another heuristic based on the number of elements used for each array can be constructed but since elements are fetched in groups, i.e. in terms of cache-lines, this heuristic is not appropriate for structured stencil codes (but might possibly find use in unstructured domains). We do not experiment and evaluate the heuristics H_2 and H_3 as exploring tiling is not the focus of the current work (but we add this investigation as a candidate in our future work).

6.8 Experimental Results

We first carry out a set of performance evaluations using various topologies on a single node, followed by multiple nodes. Our sequence of experiments is as follows:

- Evaluate and analyze the Independent Compute (IC) for increasing grid sizes and process numbers for characterizing the shared L3 cache behaviour on ARC2
- Optimize the IC using established techniques
- Evaluate and analyze plane communication times for increasing grid sizes with two different intra-node process placement policies on ARC2
- Validate the inferences from our model by combining the IC and plane communication times on ARC2
- Evaluate a light-weight, dynamic, tiling heuristic against exhaustive tiling and compiler switches for on-node Parallel Geometric Multigrid on ARC2 and ARC3
- Present performance results for multiple nodes on ARC2 and ARC3

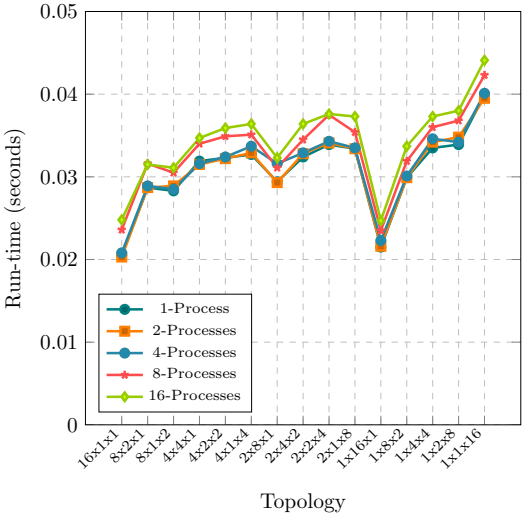
- Observe the relationship between the frequency and size of Z-planes passing through a hierarchy of networking elements and optimal partitions on ARC2
- Present Weak Scaling and Strong Scaling results for ARC2 and ARC3

6.8.1 Single Node

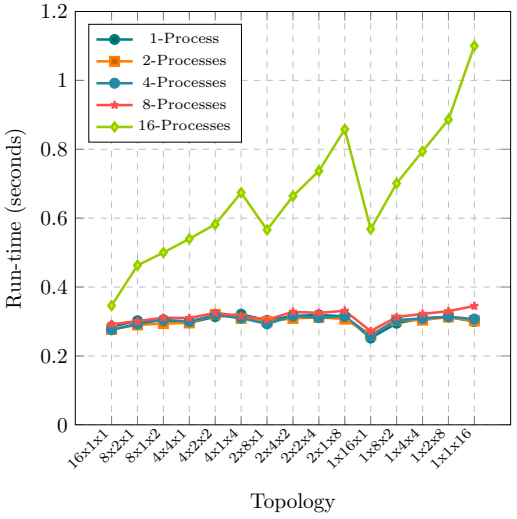
With the growing number of cores in a single node, it becomes important to characterize the intra-node behaviour of applications. Further, in a shared cluster, the traffic generated by multiple user applications does not affect the on-node communication latencies. A single node of our cluster ARC2 consists of 8 cores per-socket with a total of 2 sockets. The default scheduling policy is `--bind-to-core --bysocket` which maximizes the bandwidth per core (the first process is assigned to core 0 in socket 0, the second process is allocated to core 0 in socket 1, the third process is allocated core 1 in socket 0 and so on). With OpenMPI 1.6.5, `mpiexec --report-bindings` displays the default binding in the standard error file. As the number of processes increase, the contention for the LLC (20 MB/socket) and main memory (16 GB/socket) per socket increases. To study this behaviour, we weakly scale a problem of given size per core but with no communication. Thus, the problem size per process remains constant as we increase the number of processes. The average execution time of n processes should ideally remain constant as each core executes a same-sized but completely independent problem. In particular, each core updates the Independent Computation (IC) zone of a sub-domain using a 7-pt stencil. This is equivalent to performing smoothing operations on the IC at the finest grid level only.

6.8.1.1 Weak Scaling the IC

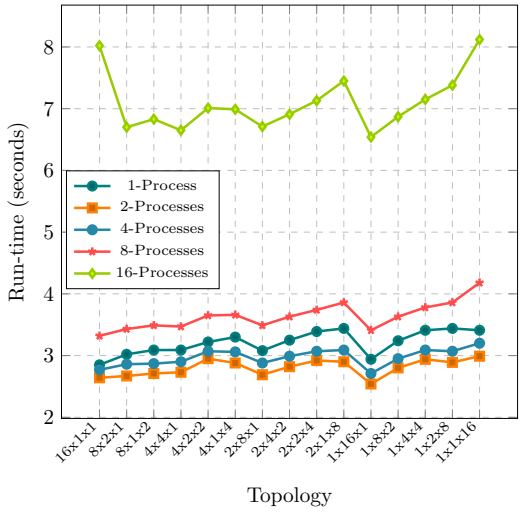
Figure 6.8 shows the maximum execution times of the Independent Compute kernel on any process, with each core (or process) having a sub-domain of size $\frac{64^3}{16}$ (Figure 6.8a), $\frac{128^3}{16}$ (Figure 6.8b), $\frac{256^3}{16}$ (Figure 6.8c) and $\frac{512^3}{16}$ (Figure 6.8d), respectively. If we run a single process on a 64^3 domain within a 16-core node, then that process handles a sub-domain of size $\frac{64^3}{16}$. If we run 8 processes on the 16-core node, then each process handles a sub-domain of the same size i.e. $\frac{64^3}{16}$. Similar cases are used for other domains i.e. 128^3, 256^3 and 512^3. Further, the shape of the sub-domain varies with the different topologies obtainable with $P = 16$. For example, with a topology of $16 \times 1 \times 1$ (and domain 64^3), a sub-domain having dimensions $P_x \times P_y \times P_z = 4 \times 64 \times 64$ is produced, whereas the topology $4 \times 4 \times 1$ produces sub-domains each having shape $16 \times 16 \times 64$. Since there is no communication between processes and each core operates independently on given equal sized sub-domains, the time for the Independent Compute should ideally be equal for all processes, irrespective of the number of cores (or processes) we utilize. However, in practice, this is not true as increasing the process count leads to an increase in the contention for shared resources such as the Last Level Cache and main memory per-socket. The following discussion elaborates how, with an increasing process count,



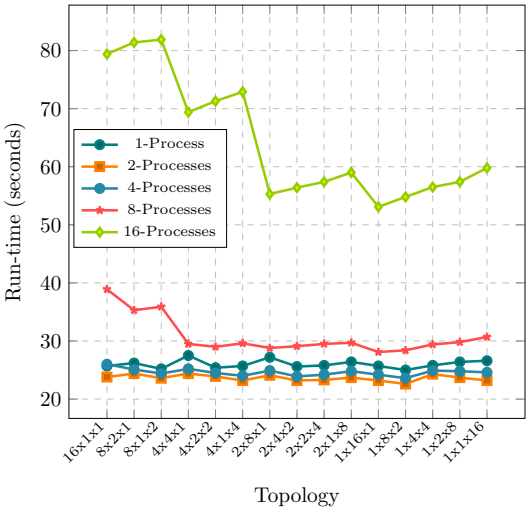
(a) IC run-times for 16384 cells per core and 384 KB working set per process



(b) IC run-times for 131072 cells per core and 3 MB working set per process



(c) IC run-times for 1048576 cells per core and 24 MB working set per process



(d) IC run-times for 8388608 cells per core and 192 MB working set per process

Figure 6.8: Weak Scaling Independent Compute (IC) for P=1,2,4,8 and 16 processes with $\frac{64^3}{16}$, $\frac{128^3}{16}$, $\frac{256^3}{16}$ and $\frac{512^3}{16}$ cells per core (with no communication) to measure impact of shared Last Level Cache per-socket contention on execution times on ARC2

the contention for the above-mentioned shared resources leads to a deterioration of performance within a node, even when the processes operate on independent sub-domains.

With a Working Set Size (WSS) of approximately 384 KB, i.e. 3 arrays of type `double` with 16384 elements ($=\frac{64^3}{16}$) each, the total WSS remains less than the shared *Last Level Cache* (LLC) per core i.e. 2.5 MB/core. It can be seen from Figure 6.8a that the characteristics of the curve indicate the unchanging behaviour of the topologies as the process count is increased from one to sixteen. Further, the heavy overlapping indicates that the execution times are approximately equal even when the LLC and shared memory contention increases with an increasing process count. This is expected as the *size(WSS-per-process) < size(LLC-per-core).* An anomaly is that the execution time of a single process is more than that of two and four processes. A plausible reason could be that *CentOS* and *OpenMPI 1.6.5* do not pin the single process [150] to a single core. But since we never run a single process per-node in the actual application, we do not investigate this any further. Figure 6.8b shows the same experiment but with a domain size of 128^3 per core and 131072 cells/core creating a WSS of $\approx$ 3MB per core. With a per socket shared LLC of 20 MB and with 8 processes per node (i.e. 4 per socket due to the binding `--bind-to-core --bysocket` configuration), the combined WSS of 4 processes is small enough to fit into the per socket LLC. Thus, Figure 6.8b shows no sudden jumps in the execution times up to 8 processes. But with 16 processes, the cumulative WSS of 48 MB exceeds the LLC and the penalty of accessing the main memory can be clearly seen in the baseline implementation running 16 processes. Figures 6.8c and 6.8d show the Weak Scaling of the Independent Computation kernel with no communication for domains of sizes 256^3 and 512^3, respectively. In both the cases the WSS per process exceeds the shared LLC per core. The change in the shape of the curve from eight to sixteen processes in Figure 6.8c shows that the execution timings need not necessarily remain fixed with respect to each other (i.e. the execution pattern of topologies in going from a smaller process count to a higher process count may not follow similar curves). A similar change can be seen in Figure 6.8d for the plot of 4, 8 and 16 processes. It is to be noted that even with the baseline implementation, there are many topologies at each domain size which outperform the sub-domain created by the standard topology, i.e. the topology returned by `MPI_Dims_create()` (henceforth referred to as MDC or the standard topology). From the results it can be seen that process topologies which have a higher value of D_y outperform other topologies in executing the Independent Computation kernel (IC) with growing data size as predicted in Section 6.6.4 (see Independent Compute (IC)). The only exception to this is the execution times of a $\frac{128^3}{16}$ sub-domain with 16 processes (see Figure 6.8b). In this case, the topologies having $D_x > D_y$ outperform other topologies.

We further elaborate the discussion to include the effects of memory bandwidth on Weak Scaling the Independent Compute without communication. For the purposes of this discussion, we reproduce the memory bandwidths of various cache levels of the Intel Sandy Bridge E5-2670

processor from [7]. The various approximate per-core bandwidths are:

1. L1 read: 16 GB/sec,
2. L2 read: 15.9 GB/sec,
3. L3 read: 15 GB/sec,
4. Main memory read: 10 GB/sec.
5. Stream benchmark: 14 GB/sec (only 1 core active), 3.8 GB/sec (all 16 cores active)

Thus, from the above, we can infer that the aggregate Stream bandwidth in fully subscribed mode is $16 \times 3.8 = 60.8$ GB/sec. The aggregate bandwidth of 60.8 GB/sec for a single node is approximately 59% of the peak (theoretical) memory bandwidth of 102.4 GB/sec.

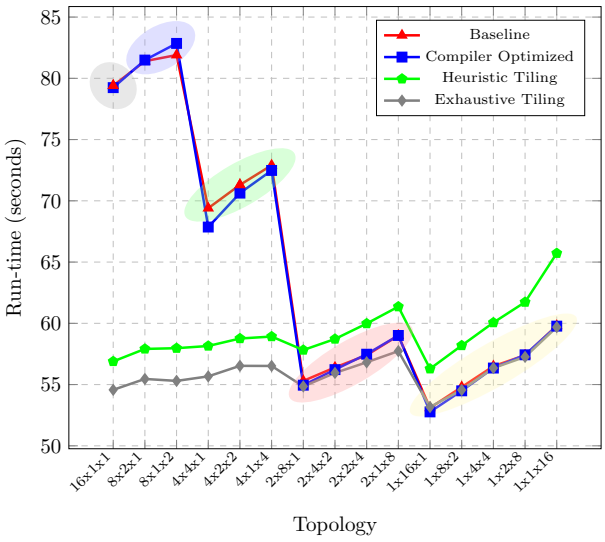
The topologies in Figure 6.8a scale i.e. the time taken for P=1, 2, 4, 8 and 16 processes is approximately the same. Having a WSS of 384 KB per process means that with P=16 the L1d cache (32KB) and unified L2 cache (256 KB) are filled to capacity and approximately 384KB - 32 KB - 256 KB = 96 KB of data resides in the shared L3 cache for each core. This clearly means that most requests (read or write) are fulfilled from the L2 cache (because it contains $\frac{256}{384} = 66\%$ of data of each process). The least time of execution for P=16 processes is $t = 2.47 \times 10^{-2}$ sec in Figure 6.8a. Hence, the bandwidth $\mathcal{B} = \frac{WSS}{t} = \frac{384 \times 1024}{2.47 \times 10^{-2}}$ bytes/sec = 15.91 MB/sec. The execution timings shown in Figures 6.8a, 6.8b, 6.8c and 6.8d are for 1000 iterations and hence the actual bandwidth becomes 15.91 MB/sec $\times 1000$ = 15.91 GB/sec. This value is in almost perfect agreement with the maximum L2 read bandwidth of 15.9 GB/sec (see enumerated list above). It is to be noted that we only consider the read bandwidth here as the write bandwidth is only 11% of the total bandwidth. This is so because in the weighted Jacobi algorithm having a RHS term, the total write bandwidth is $= \frac{1}{(1+1+6+1)} = \frac{1}{9} = 11\%$ of the total read-write traffic.

Figure 6.8b exhibits a jump in the execution timings when going from P=8 to P=16 processes. With P=8, there are 4 processes in each socket because of the default process placement policy of ARC2. Hence, with a WSS = 3 MB/process, we get the aggregate WSS as $4 \times 3 = 12$ MB/socket, which is less than the hardware capacity of 20 MB LLC/socket. With 16 processes, each socket contains 8 processes and thus a combined WSS of $8 \times 3 = 24$ MB becomes greater than the hardware capacity of 20 MB LLC/socket. This clearly indicates that the jump in execution times is because of the penalty of accessing the main memory. Further, the total WSS at P=16 is now 24 MB + 24 MB = 48 MB. Since approximately $\frac{40}{48} = 83\%$ of the requests are fulfilled by the L3 cache, we expect that the main memory bandwidth does not saturate even with P=16. From Figure 6.8b we note that the least time taken by P=16 processes is $t = 3.46 \times 10^{-1}$ sec. Hence, $\mathcal{B} = \frac{WSS}{t} = 9$ MB/sec. Since the execution times are for 1000 iterations, the actual bandwidth becomes $9 \times 1000 = 9$ GB/sec. This is much below the max 15 GB/sec read bandwidth of the L3 cache per-core but very close to the 10 GB/sec of main memory read-bandwidth

per process. The aggregate bandwidth for P=16 is then $16 \times 9 = 144$ GB/sec - a value which is greater than the bandwidth of 60.8 GB/sec for 16 processes obtained using the Stream triad benchmark. This illustrates that the main memory bandwidth is not saturated and hence the jump in execution times from P=8 to P=16 is purely because we exceed the total LLC capacity.

Figure 6.8c shows the same experiment with a WSS per process of 24 MB. Hence, even with P=2 processes, the total LLC capacity (40 MB) is exceeded. It can be seen from Figure 6.8c that the execution times scale almost up-to P=8 processes but with P=16 the times show a large increase. With P=8, the least execution time is $t = 3.32$ seconds for 1000 iterations. Hence, for P=8, $\mathcal{B} = \frac{WSS}{t} = \frac{(24\times1024\times1024\times1000)}{(3.32)} = 7.58$ GB/sec per process. For P=8 processes, the aggregate bandwidth then becomes $7.58 \times 8 = 60.6$ GB/sec. This value almost coincides with the Stream bandwidth of 60.8 GB/sec for the main memory. Further, with P=8, the total WSS is $8 \times 24 = 192$ MB and hence only $\frac{40}{192} \approx 20\%$ i.e. read requests are fulfilled from the LLC as a rough approximation. Thus, the main memory provides the remaining 80% of the requests and the aggregate main memory bandwidth calculated as 60.6 GB/sec indicates the saturation of bandwidth. With P=16, since we have already saturated the bandwidth, the execution times as seen in Figure 6.8c do not scale. For P=16, the lowest execution time is $t = 6.54$ sec and hence $\mathcal{B} = \frac{WSS\times1000}{t} = \frac{(24\times1024\times1024\times1000)}{6.54} = 3.84$ GB/sec. Thus, the aggregate bandwidth of 16 processes is $3.84 \times 16 = 61.5$ GB/sec (which again is extremely close to 60.8 GB/sec derived from the Stream benchmark). Clearly, this jump in execution times from P=8 to P=16 is because of the main memory bandwidth saturation.

Figure 6.8d shows the execution times of P = 1, 2, 4, 8 and 16 processes for a WSS-per-process = 192 MB. Thus, even a single process exceeds the total LLC of 40 MB. The execution times scale up-to P=8 processes approximately. The least execution time for P=8 processes is $t = 28.1$ sec for 1000 iterations. Hence, the bandwidth per process is $\mathcal{B} = \frac{WSS}{t} = \frac{192\times1024\times1024\times1000}{28.1} = 7.16$ GB/sec. For 8 processes the aggregate bandwidth is then $7.16 \times 8 = 57.31$ GB/sec. This value indicates that we have almost reached the limit indicated by the Stream benchmark i.e. 60.8 GB/sec and P=16 should not scale. With P=16, the least $t = 53.1$ sec for 1000 iterations and hence $\mathcal{B} = \frac{WSS}{t} = \frac{(192\times1024\times1024\times1000)}{53.1} = 3.79$ GB/sec. The aggregate bandwidth is then $3.79 \times 16 = 60.6$ GB/sec, which coincides with the maximum main memory bandwidth obtained using the Stream benchmark. In summary, the sudden increase in execution times in going from P=8 to P=16 processes is because of exceeding the LLC capacity when considering Figure 6.8b. Further, the sudden increase in execution times between P=8 to P=16 is because of the saturation of main memory bandwidth when considering Figures 6.8c and 6.8d.



(a) Execution times

Topology	WPSS
16x1x1	786432
8x2x1	393216
8x1x2	393216
4x4x1	196608
4x2x2	196608
4x1x4	196608
2x8x1	98304
2x4x2	98304
2x2x4	98304
2x1x8	98304
1x16x1	49152
1x8x2	49152
1x4x4	49152
1x2x8	49152
1x1x16	49152

(b) WPSS for Topologies

Figure 6.9: Baseline/naive implementation, Compiler optimized run-times with `-O3 -xHOST -ip -ansi-alias -fno-alias`, Heuristic square tile for X/Y dimensions (based on Rivera and Tseng [6] square tiles), Exhaustive Tiling for domain of size 512^3 and 16 processes on ARC2, default `MPI_Dims_create()` $= 4 \times 2 \times 2$

6.8.1.2 Compiler Switches and Heuristic Tiling (H_1)

The performance of the topologies can be enhanced by using techniques such as optimal compiler switches, cache tiling, vectorization with appropriate alignment and exclusive SIMD directives. We compare the execution times of various topologies with a domain size of 512^3 with these optimizations. The objective is to optimize the bulk of the computation, i.e. the Independent Computation kernel of the sub-domain. The results are presented in Figure 6.9 where the tiled code generally performs better than the code exploiting optimal compiler switches. We create a light-weight, run-time, space tiling heuristic H_1 (see section 6.7.1) based on the size of the LLC per core and a working set (WSS) of three equal sized arrays. Following the work of Rivera and Tseng [6], we assume that square tiles should be used in the X and Y direction i.e. $CX = CY$. Thus, for a single 3-D array having $CX = CY = k$ and an un-cut unit-stride dimension $P_z + 2$, the number of elements should equate to:

$$k^2 \times (P_z + 2) = \frac{2.5}{3} \times \frac{1024 \times 1024}{8},$$

yielding

$$k = \left\lceil \sqrt{\frac{104857.6}{(P_z + 2)}} \right\rceil.$$

Although with exhaustive tiling we are able to find tile sizes CX and CY (with $CX \neq CY$ for

the majority of the topologies) which outperform the heuristic that we create, the tile iteration space becomes huge and thus finding the optimum becomes a time consuming process. The task of optimizing stencils depends heavily upon the hardware parameters such as cache sizes, cache-line size, prefetch policies, stencil order, data size, and the algorithm employed, etc. [151]. The range of relative error between the execution times found using the heuristic and the optimal tile size is $\approx 4-10\%$. An observation is that most process topologies outperform the `MDC` topology in the cases of exhaustive and heuristic tiling. Specifically, the compiler optimized version of $1\times16\times1$ outperforms the Independent Computational kernel created by the standard topology by $\approx 25.2\%$ (see Figure 6.9a).

6.8.1.3 Working Planes Set Size (WPSS)

To understand the difference in the run-times of the baseline version of the different sub-domains, we group the various process topologies on the basis of the Working Planes Set Size (WPSS). The WPSS for a 7-pt stencil is the number of elements in the three planes which are required to update a single plane. Thus, the total elements (`double` type) contained in three planes are $3\times(P_y+2)\times(P_z+2)\approx 3\times P_y\times P_z$. We cluster the topologies having the same WPSS into a single group (see Figure 6.9b). To compare the execution times of the IC kernel of two topologies T1 and T2, their WPSS is computed. The WPSS of both T1 and T2 may or may not fit into the $\frac{LLC-per-core}{3}$, where the denominator indicates that the LLC is assumed to be equally divided between three arrays namely, the write array (v), the read array (u) and the array representing the source (or RHS) term (f) (see Equation (6.1) in Section 6.6). We can distinguish between at least three cases:

1. *WPSS(T1)* $\neq$ *WPSS(T2)* and both $> \frac{LLC-per-core}{3}$: In this case, more weight is given to the WPSS as compared to the Vectorization factor (i.e. the length of P_z - the larger the better).

2. *WPSS(T1) = WPSS(T2)* and *WPSS* $> \frac{LLC-per-core}{3}$: The topology with a higher value of P_z outperforms the other.

3. *WPSS* $< \frac{LLC-per-core}{3}$: Here the demarcation between the performance of topologies becomes blurred and needs more investigation.

The topology $16\times1\times1$ in Figure 6.9a deviates from the first rule above and outperforms topologies $8\times2\times1$ and $8\times1\times2$ despite having a larger WPSS. Empirically, it is very difficult to exactly determine the working set brought into the different cache levels but still the rules formulated above provide a substantially accurate insight into the relative baseline performance of various topologies.

6.8.1.4 Communication times of Dependent Planes

Figures 6.10a, 6.10b, 6.10c and 6.10d show the individual maximum time for sending and receiving X/Y/Z planes to/from neighbouring processes within an SMP (Symmetric Multiprocessor) of ARC2 for $P = 16$ and increasing plane sizes. The communication times of topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ form the basis of the following observations:

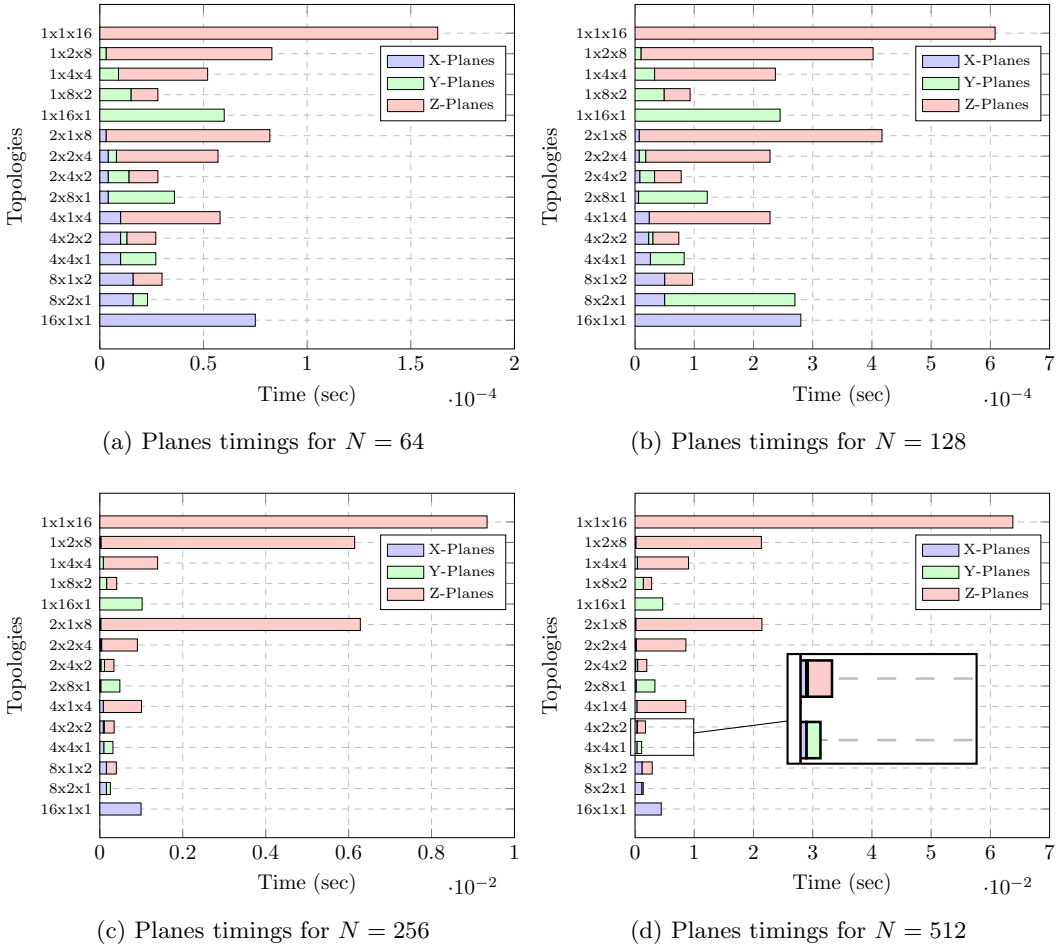
1. For the same sized X, Y and Z planes, the Z-plane takes the maximum amount of time (as indicated in Table 6.3). For example, topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ all pass equal-sized inter-socket X, Y and Z planes. At $N = 64, 128$, same sized Z-planes take about `3x` the time as compared to the X/Y planes. At $N = 256$ and 512, they take `9x` and `12x` the time, respectively. Our predictions in Table 6.3 show that the Z-plane communication is `8x` more expensive than its siblings.

2. At $N = 64$ and 128, the same sized X-planes on an average take a factor of `1.2` more time than the Y-planes but at $N = 256$ and 512 the Y-planes take a factor of `1.03` more time than the X-planes. Our predictions show that same sized X and Y-planes should take the same amount of time (see Table 6.3 in Section 6.6).

3. When the surface area of planes is quadrupled, the communication times of inter-socket X planes increases by factors of `3.3-4.5`, the inter-socket Y-planes by `4-4.68` whereas the factor is between `3.73-15` for the inter-socket Z-planes. These ranges of times for the X/Y planes are as expected, but the `15x` jump in timings from $N = 128$ to $N = 256$ for the Z-plane is much greater than the expected, theoretical, `4` times increase.

We consider the topology $1 \times 1 \times 16$ to understand the abnormal increase in the communication timings of the Z-plane. The topology $1 \times 1 \times 16$ produces $P_z = 8$ for $N = 128$ and $P_z = 16$ for $N = 256$. The distance between any two adjacent mesh points in the Z-plane ($Z_{adj} = P_z + 2$) then becomes 10 and 18, respectively. The L1 streaming hardware prefetcher (DCU - Data Cache Unit) fetches only one extra cache-line with ascending addresses. Thus, effectively two cache lines or $\frac{64+64}{8} = 16$ `double` elements are fetched. For the discussion that follows, we assume that the cache is initially empty and we are accessing the elements of the Z-plane. With $Z_{adj} = 10$, after an initial cache-miss, there is no cache-miss to access the second element as the prefetch mechanism is able to fetch an extra 8 `double` elements which include the next element on the Z-plane. However, with $Z_{adj} = 18$, a cache-miss occurs when accessing both the elements i.e. a cold cache-miss followed by a cache-miss when the second element of the Z-plane is accessed. As discussed in Section 6.6.4, this illustrates how prefetching affects the communication times for a particular choice of sub-domain dimensions.

In summary, the majority of timings for various topologies can be explained and compared on the basis of the following: (i) Size of the plane being passed (ii) Number of planes being



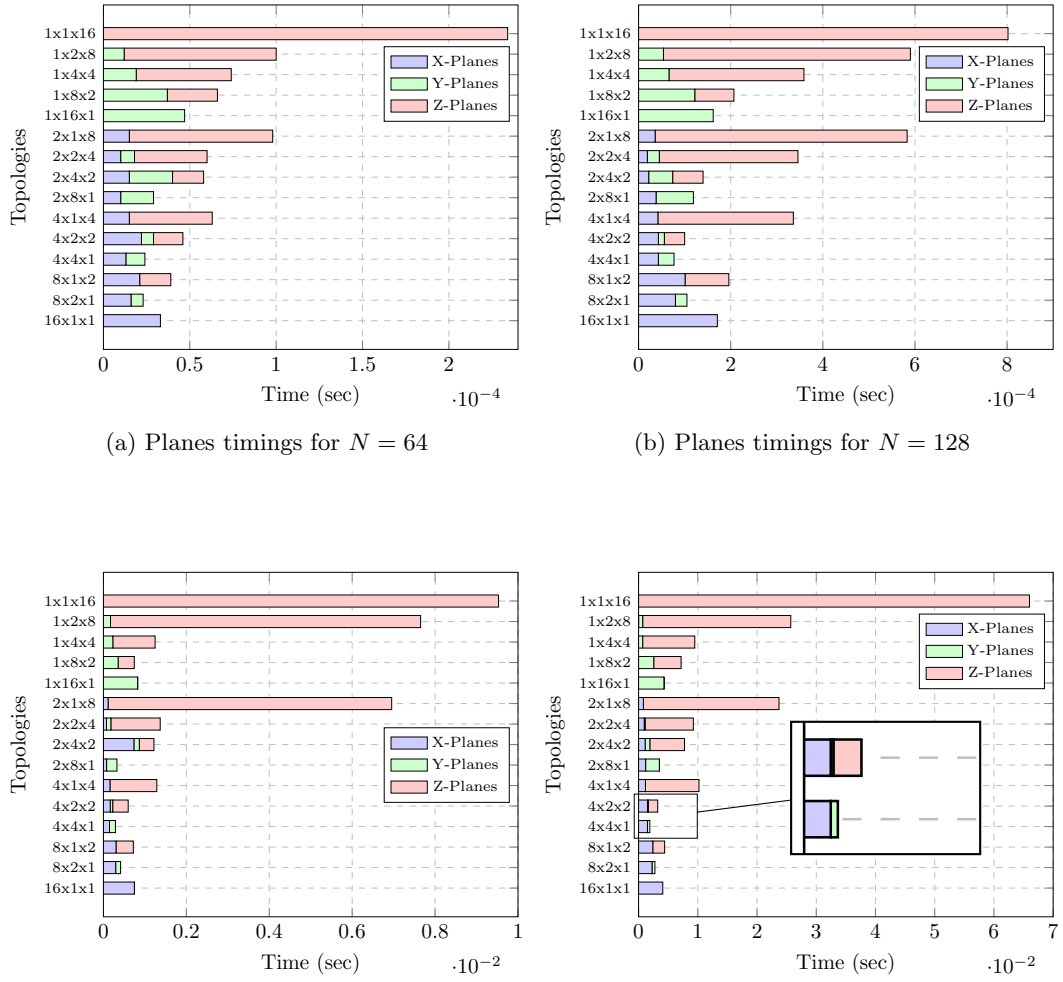
(a) Planes timings for $N = 64$

(b) Planes timings for $N = 128$

(c) Planes timings for $N = 256$

(d) Planes timings for $N = 512$

Figure 6.10: Maximum average time (maximum time over processes and average of runs) to send and receive X/Y/Z planes separately within a 16-core node for topologies (`--bind-to-core -bysocket`) using Intel 16.0.2 and OpenMPI 1.6.5 on ARC2, default `MPI_Dims_create()` = $4 \times 2 \times 2$



(a) Planes timings for $N = 64$

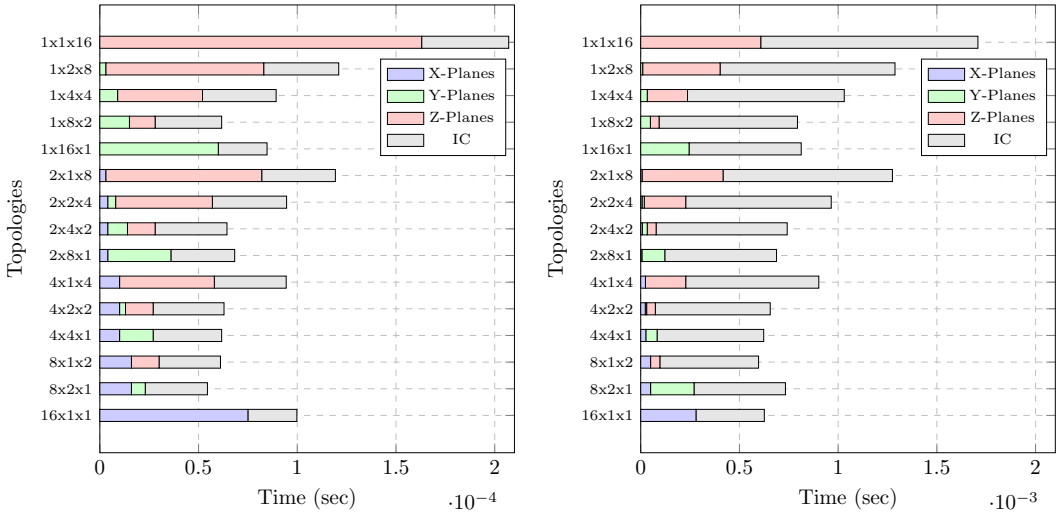
(b) Planes timings for $N = 128$

(c) Planes timings for $N = 256$

(d) Planes timings for $N = 512$

Figure 6.11: Maximum average time (maximum time over processes and average of runs) to send and receive X/Y/Z planes separately within a 16-core node for topologies ((`--bind-to-core -bycore`))

exchanged (iii) Region of movement of plane i.e. intra-socket or inter-socket and (iv) Cache-misses during packing/unpacking of plane (depends on whether it is an X/Y/Z plane). The timings in Figures 6.10a, 6.10b, 6.10c and 6.10d do not exactly reflect the actual timings in the real scenario since the X/Y/Z planes in these simulations are being passed and received separately i.e. a single type of plane (either X or Y or Z) is being handled separately. In a real application, all types of planes are passed simultaneously depending on the implementation. Thus, the latter should produce an increased number of simultaneous send/receive requests per process and hence deteriorate the total communication timings further.



(a) Combined IC and plane timings, $N = 64$ (b) Combined IC and plane timings, $N = 128$

Figure 6.12: Relative plane communication and Independent computation times for $N = 64$ and $N = 128$ with $P = 16$ ((`--bind-to-core -bysocket`)) using Intel 16.0.2 and OpenMPI 1.6.5 on ARC2, plane update execution times are not shown, default `MPI_Dims_create()` $= 4 \times 2 \times 2$

Another intra-node process binding scheme, namely `--bind-to-core --bycore`, fills up a single socket with increasing ranks instead of a round-robin policy of utilizing sockets. The key idea is to reduce the cost of communication by increasing the possibility of neighbouring ranks residing on the same socket. Figures 6.11a, 6.11b, 6.11c and 6.11d show the timings for sending X/Y/Z planes within an SMP with the core bindings being `--bind-to-core -bycore`. For the topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$, and with increasing plane sizes, the communication time of X planes increases by a factor `4-6`, for Y planes by `3.5-5` and Z planes by `3.5-12` (compared to a `3.73-15x` increase in `--bind-to-core -bysocket`). The abnormal jump by a factor of `12` for Z-planes occurs when the planes size increases from 128×128 to 256×256 elements. Thus, with Z_{adj} changing from 10 to 18, the L1 Streaming hardware prefetcher (Data Cache Unit) is unable to prefetch the cache-line which contains the next mesh point, resulting in a miss for every mesh point when $Z_{adj} = 18$. For equal sized X/Y/Z planes, the communication of the Z plane is a factor `6-15` more expensive than X/Y planes when the binding policy is `--bind-to-core --bycore` (compared to a `3-12x` increase in `--bind-to-core -bysocket`).

6.8.1.5 Combining IC and DP timings

Figures 6.12a and 6.12b show the relative/combined times for the Independent Compute (IC) and communication of planes. At $N = 64$ and $P = 16$ (16-core node), the communication cost in almost every topology exceeds the IC cost, clearly indicating that the communication

Table 6.5: h-independence: of Parallel Geometric Multigrid, Coarsest Grid tolerance = 10^{-8}, Finest Grid tolerance = 10^{-5}

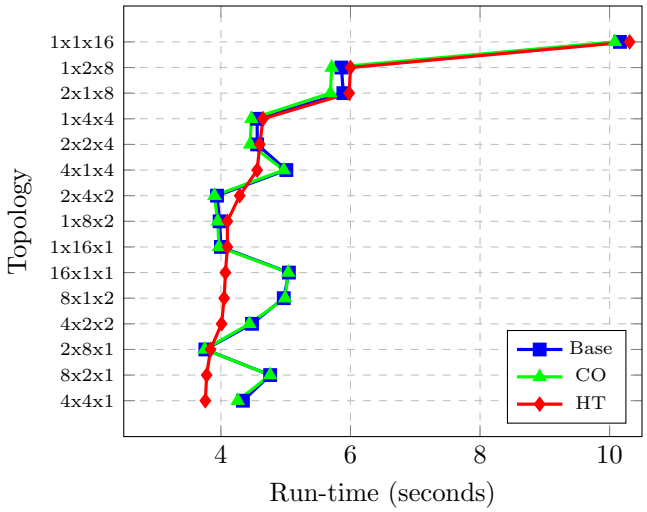
Fine Grid	Levels	Coarsest Grid	V-cycles V(3,3)
$32 \times 32 \times 32$	2	$16 \times 16 \times 16$	2
$64 \times 64 \times 64$	3	$16 \times 16 \times 16$	2
$128 \times 128 \times 128$	4	$16 \times 16 \times 16$	2
$256 \times 256 \times 256$	5	$16 \times 16 \times 16$	2
$512 \times 512 \times 512$	6	$16 \times 16 \times 16$	2
$1024 \times 1024 \times 1024$	7	$16 \times 16 \times 16$	2
$2048 \times 2048 \times 2048$	8	$16 \times 16 \times 16$	2

cannot be completely hidden within computation at coarser levels of a multigrid solver. We would expect the communication to remain completely hidden within computation at finer grid levels as shown by the larger computation times in Figure 6.12b but this overlap is completely governed by the OpenMPI implementation and the underlying hardware. These two figures further show that a topology which has the least IC computation time may not yield the optimal partition as it may have a higher communication time as compared to other topologies. For example the topology $1 \times 16 \times 1$ has the least IC execution time at $N = 64$, $P = 16$, as can be seen in Figure 6.12a, but its total execution time (disregarding overlap) is more than a topology such as $4 \times 4 \times 1$ or $4 \times 2 \times 2$. This observation lends support to our model, as the latter topologies have a much more balanced D_x and D_y.

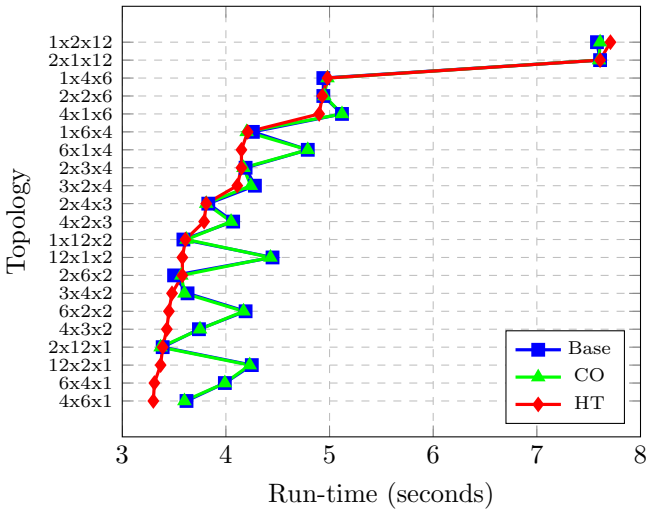
6.8.1.6 Multigrid

Before we describe the experimental results of Parallel Geometric Multigrid, we demonstrate below the correctness of our Multigrid implementation by demonstrating the property of h-independence of the convergence rate of Geometric Multigrid. The h-independence condition means that the number of V-cycles in Geometric Multigrid should approximately remain constant, and independent of the fine grid size, if we fix the coarsest grid size. Table 6.5 shows the number of V-cycles with three pre and post smoothing steps, i.e. V(3,3), which attains this property. As can be seen from the table, when the problem size increases from $32 \times 32 \times 32$ to $2048 \times 2048 \times 2048$ i.e. an increase of 2^{18} times, the number of V-cycles remain constant for a tolerance of 10^{-8} for the coarsest grid solve and 10^{-5} for the finest grid. The tolerance is the ratio of the initial l_2 norm of the residual to the current norm. As described in [25], the restriction stencil operator must be modified near the Neumann boundaries to prevent the convergence rate of Multigrid from deteriorating.

Figure 6.13a shows the Baseline (Base) (`-O2`), aggressively Compiler Optimized (CO) (`-O3 -xHOST -ip -ansi-alias -fno-alias`) and Heuristically Tiled (HT) versions of Parallel Geometric Multigrid for the largest problem that we could fit into a 16-core node of ARC2 i.e.

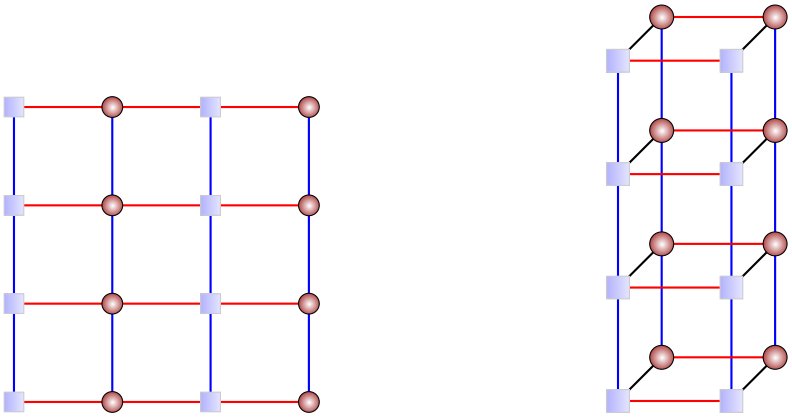


(a) Topology Run-times for $P = 16$, $N = 512$, Levels = 6, Coarsest iterations = 100, 5 V(3,3) cycles, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` $= 4 \times 2 \times 2$



(b) Topology Run-times for $P = 24$, $N = 576$, Levels = 5, Coarsest iterations = 400, 5 V(3,3) cycles, Intel 17.0.1, OpenMPI 2.0.2, ARC3, default `MPI_Dims_create()` $= 4 \times 3 \times 2$

Figure 6.13: Intranode execution times of Parallel Geometric Multigrid using Baseline (Base), aggressive Compiler Optimization (CO) and Heuristically Tiled (HT) versions on ARC2 and ARC3



(a) Topology $4 \times 4 \times 1$ (b) Topology $4 \times 2 \times 2$

Figure 6.14: 16 processes in a single node of ARC2 arranged by `--bind-to-core -bysocket`, Blue squares represent socket 1, Red balls represent socket 2, thick black lines are Z-planes, thick blue lines are X-planes, thick red lines are Y-planes.

approximately 8 million cells/core (or 0.13 billion *dof*). We use the H_1 heuristic for the HT version (see Section 6.7.1). It can be noted that a topology such as $4 \times 4 \times 1$ outperforms the standard topology $4 \times 2 \times 2$ in all three versions even though the former sends (or receives) a maximum of two inter-socket Y-planes per process that are two times larger than the Y-planes of the latter topology, which sends (or receives) only intra-socket Y-planes. The situation is shown in Figure 6.14a and 6.14b where the different cores of a particular socket are shown by means of squares and circles, respectively. Each of the X/Y and Z planes are shown by means of a different coloured line in these figures. The performance gap between topologies stems from the fact that the topology $4 \times 2 \times 2$ send and receives inter-socket Z-planes which are absent in $4 \times 4 \times 1$. Thus, the cost of packing/unpacking the Z-plane for $4 \times 4 \times 1$ is zero whereas the standard topology has to pack/unpack/communicate the high cost Z-plane (see the magnified section of Figure 6.10d).

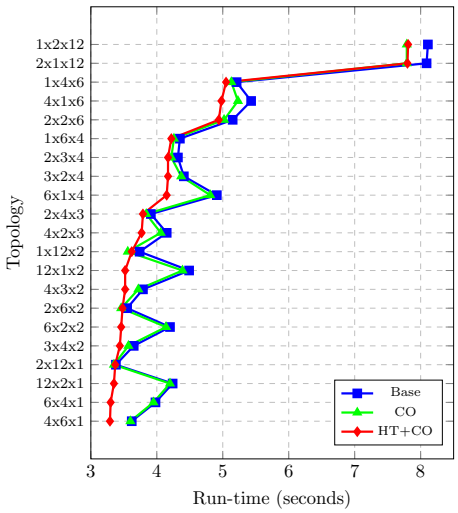
Figure 6.13b shows the execution time of Parallel Geometric Multigrid on a single node of the ARC3 cluster using the Intel 17.0.1 compiler and OpenMPI 2.0.2 with approximately 8 million cells/core (or 0.19 billion dof). A significant difference between the OpenMPI 1.6.5 and OpenMPI 2.0.2 implementations is the change of the shared memory module (`-sm` module) to the `-vader` module, the latter offering performance benefits over the former. With 24 cores, the Heuristically Tiled version of $6 \times 4 \times 1$ and $4 \times 6 \times 1$ outperform the `MPI_Dims_create()` topology of $4 \times 3 \times 2$. The WPSS of $4 \times 6 \times 1$ is less than that of $6 \times 4 \times 1$ and thus is the major factor in contributing to the improved performance of the former within a single node (as process placement effects within a single node can be ruled out). It may be noted that although having a large P_z offers an enhanced opportunity for Vectorization, it decreases the

probability of the data remaining in the cache before that data is accessed again because of the Least Recently Used (LRU) eviction policy (see Figure 6.7). The intranode execution trends of topologies on ARC2 and ARC3 show that our predictions, and the behaviour of topologies, are consistent across different hardware.

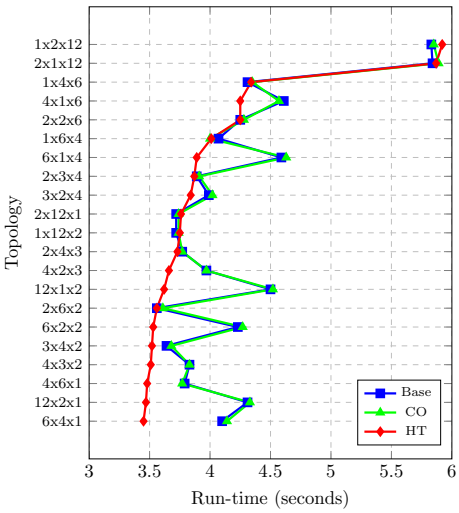
Figures 6.13b, 6.15a, 6.15b and 6.15c show the effect of different combinations of compilers and MPI implementations using a combination of Intel 17.0.1 + OpenMPI 2.0.2 (henceforth called I17O2), GNU 6.3.0 + OpenMPI 2.0.2 (henceforth called G6O2), Intel 17.0.1 + Mvapich2/2.2 (henceforth called I17M2) and GNU 6.3.0 + Mvapich2/2.2 (henceforth called G6M2), respectively, on a domain of size 576^3 on a single node of ARC3. For each of these, three variations in the form of Base version for Intel 17.0.1 (`-O2`) and GNU 6.3.0 (`-O2`), aggressive CO for Intel 17.0.1 (`-O3 -xHOST -ip -ansi-alias -fno-alias`) and GNU 6.3.0 (`-O3 -march=native`) and HT were tested. Since Heuristic Tiling alone provided negligible benefits without aggressive compiler based optimization with GNU 6.3.0, it was coupled with the latter (i.e. HT+CO - see Figure 6.15a and 6.15c). The curves in I17O2, I17M2, G6O2, and G6M2 are a characteristic of the compiler which is used. The experiments with the Intel 17.0.1 compiler, irrespective of the MPI implementation version, showed negligible difference between the Base version and the CO version while showing the best timings with HT alone. The optimal timings were obtained with a combination of HT+CO with GNU 6.3.0. Overall, the optimal execution timings were obtained with topologies $D_x \times D_y \times D_z = 4 \times 6 \times 1$ and $D_x \times D_y \times D_z = 6 \times 4 \times 1$ - the topologies which are predicted with our model. For every version (Base, CO, HT, HT+CO) of I17O2, I17M2, G6O2, and G6M2, one of the predicted topologies i.e. either $4 \times 6 \times 1$ or $6 \times 4 \times 1$, outperformed the default `MPI_Dims_Create()` (MDC) topology of $4 \times 3 \times 2$. The performance gains for the versions using Mvapich2/2.2 (i.e. I17M2 (1.70%) and G6M2 (1.71%)) were smaller as compared to versions using OpenMPI 2.0.2 (i.e. I17O2 (3.79%) and G6O2 (6.53%)) - possibly suggesting a performance sensitivity of topologies on the efficiency of communication routines in the MPI implementations. Interestingly, the optimal run-time of the OpenMPI versions (I17O2 and G6O2) had a performance gain of approximately 4.36% over the best execution timing of the Mvapich2/2.2 versions (I17M2 and G6M2). Figure 6.15d shows the minimum timings for I17O2, I17M2, G6O2 and G6M2. The curves for I17O2 and G6O2 almost overlap i.e. have negligible differences and hence are shown as a single curve. The similarity in the shape of curves in Figure 6.15d shows the software independence of our model. This behaviour is ideally expected as our high level abstract model is derived using only the data layout, as elaborated in Section 6.6, and is independent of any particular software or hardware characteristics.

6.8.2 Multiple Nodes

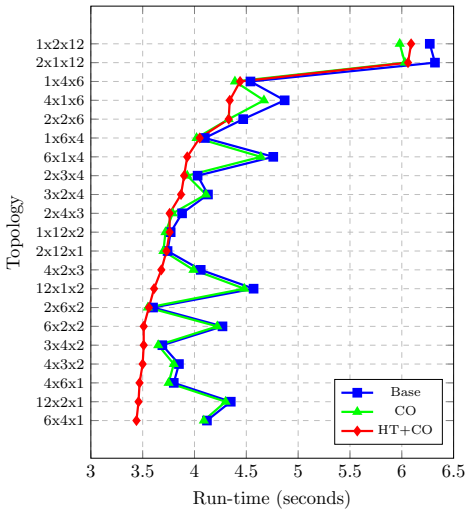
Figure 6.16a shows the total run-time of parallel geometric multigrid for various topologies which are feasible when the global fine grid size is $512 \times 512 \times 512$ (0.13 billion dof) and the global coarsest grid is $16 \times 16 \times 16$ (i.e. 6 levels) for $P = 64$ on ARC2. As predicted by our



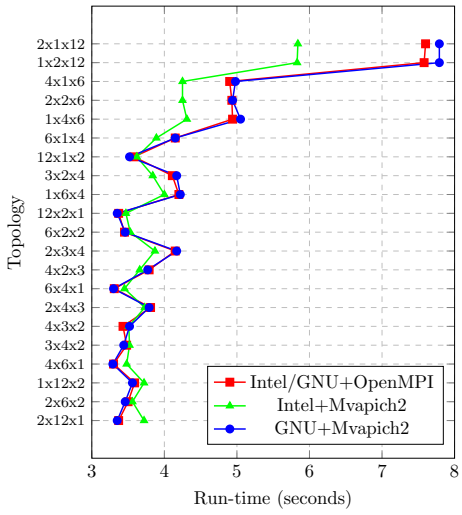
(a) GNU 6.3.0 and OpenMPI 2.0.2



(b) Intel compiler 17.0.1 and Mvapich2/2.2



(c) GNU 6.3.0 and Mvapich2/2.2



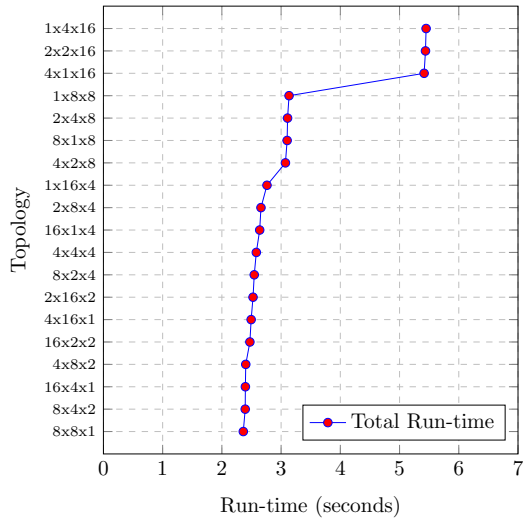
(d) Minimum execution times

Figure 6.15: Topology Run-times for $P = 24$, $N = 576$, Levels = 5, Coarsest iterations = 400, 5 V(3,3) cycles and the minimum run times for various combinations of compilers and MPI implementations on ARC3, default `MPI_Dims_create()` $= 4 \times 3 \times 2$

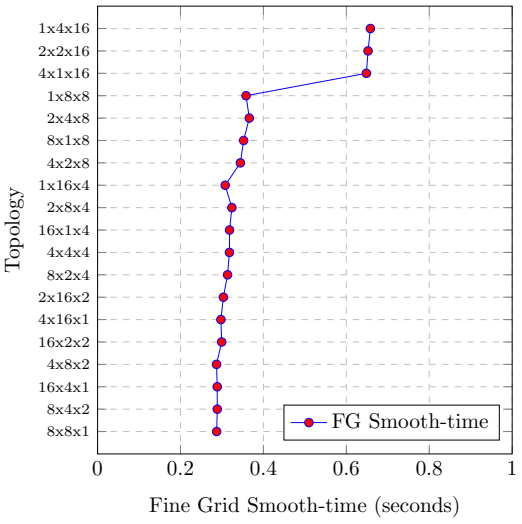
model, there are cache-minimizing topologies which outperform the standard topology $4 \times 4 \times 4$ returned by `MPI_Dims_create()` with $P = 64$. Figure 6.16b and Figure 6.16c show the corresponding fine grid smooth times and coarsest grid run-times, respectively. The performance improvement of the best performing topology $8 \times 8 \times 1$ over $4 \times 4 \times 4$ in the total run-time is 8.5% whereas in the fine grid smooth time it is 9.7%. As the fine grid smoothing time is the major contributor to the total running time, Figure 6.16a and Figure 6.16b bear a striking resemblance. The coarsest grid run-times are very small in comparison and appear to be irregular at this level. The cache misses at the coarsest level will have a lesser effect on the running time as compared to the communication time due to process placement and message latency as the local work-set of the three arrays used in Jacobi updates is 5.1 KB (including the halo cells) for $4 \times 4 \times 4$ and 6.75 KB for $8 \times 8 \times 1$, which can easily fit into the L1d cache. The latter topology passes a maximum of four planes as opposed to a maximum of six by the former. Assuming perfect cache hits (as the local-work set fits into L1d cache), it is the message latency which becomes the primary factor in the $8 \times 8 \times 1$'s superior performance over $4 \times 4 \times 4$ at the coarsest level. Our implementation uses *persistent* point-to-point communication at the coarsest level as the number of halo exchanges at the coarsest level $>> (\nu_1 + \nu_2)$ and thus we can expect to see a benefit in not destroying the MPI send and receive handles every time data is communicated.

Figure 6.17a shows the number of intra-node Z-planes being passed for each topology for $P = 64$ on ARC2 at the fine grid level when the topologies are arranged in the ascending order of their total run-times. The number of intra-node/inter-node X/Y/Z planes at all levels for a particular topology are equal except for at the coarsest grid. The communication volume decreases by one-fourth in going from a finer level to the next coarser level. Further, we only count the total number of Z-planes which are sent, as it includes the number of Z-planes which will be received. It can be seen from Figure 6.17a that as the number of Z-planes increase, so does the size of the communicated Z-plane. The number of planes however should not be related directly to the time being taken by a topology as these planes are exchanged simultaneously. The majority of the best performing topologies in this case again are the ones which pass a smaller sized Z-plane or do not pass a Z-plane at all.

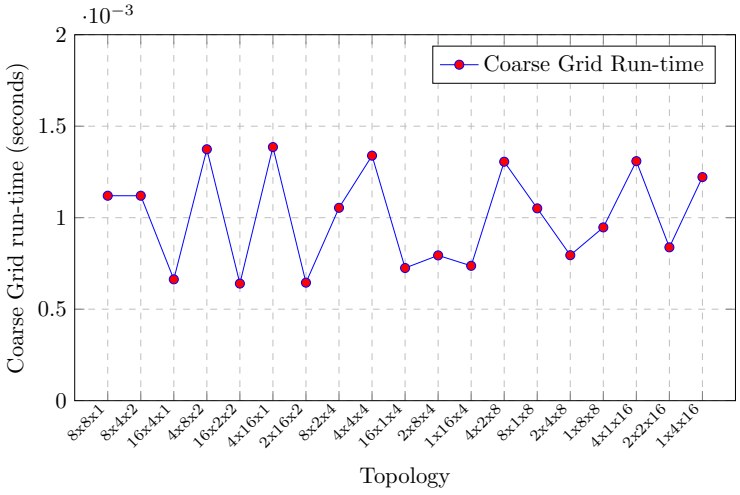
For $P = 64$, the maximum time taken by any process to communicate X/Y/Z planes was measured on ARC2. In Figure 6.17b, whenever the time taken by X-planes is greater than the time taken by Y-planes, the X plane was larger than the Y plane or the X plane was passed between *racks* and thus the switch hop latency contributed to the total time. Further, whenever equal sized X/Y and Z planes were passed, irrespective of whether it was an intra-node or inter-node plane, the Z-plane communication time exceeded its siblings. The exceptional case was with the topology of $4 \times 4 \times 4$, where an equal sized Y-plane (intra-node) took more time than the X-plane (inter-node). More research is needed to determine the reason for this



(a) Total run-time

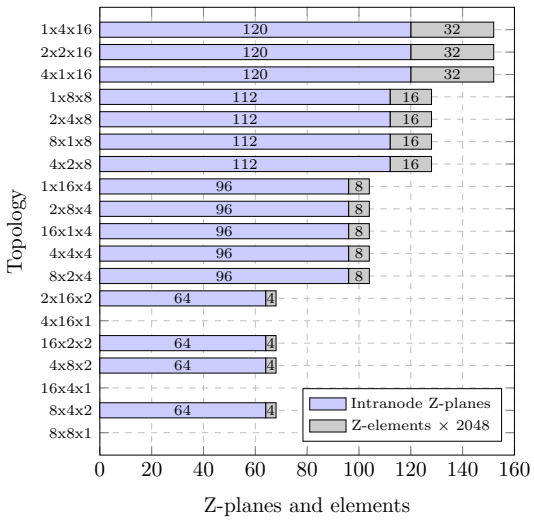


(b) Fine Grid (FG) smooth time

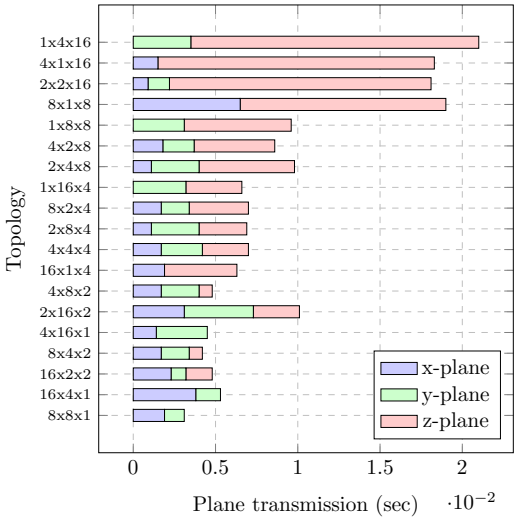


(c) Coarsest Grid run-time

Figure 6.16: Execution times of Geometric Multigrid for $P = 64$, Fine Grid $= 512^3$, Levels $= 6$, Global Coarsest Grid $= 16^3$, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations $= 100$, Vcycles $= 5$, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` $= 4 \times 4 \times 4$



(a) Intra-node Z-planes and elements for topologies with $D_z > 1$



(b) Communication times for individual X/Y/Z Planes

Figure 6.17: $P = 64$, Fine Grid = 512^3, Levels = 6, Global Coarsest Grid = 16^3, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations = 100, Vcycles = 5, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` $= 4 \times 4 \times 4$

deviation from the normal.

We can differentiate between various plane categories depending on the hierarchy of network they interact through. Table 6.6 divides the X/Y/Z planes into 4 categories each depending on their region of movement. The cheapest communication is intra-node communication and the costliest communication is inter-rack communication. Considering the case of extreme topologies with $P = 64$ processes or cores i.e. $1 \times 1 \times 64$, $1 \times 64 \times 1$ and $64 \times 1 \times 1$, we recorded the exchange of planes on ARC2 as listed in Table 6.7. It can be seen from Table 6.7 that exactly the same number and size of planes are passed in a particular category. The corresponding running times at the fine grid level and coarsest grid level (Global Coarsest Grid (GCG) = 64^3) is shown in Table 6.8 where it can be seen that the time taken by the X and Y partition is almost equal but the Z partition is outperformed by a factor of ≈ 3 and 3.5 at the fine grid level and coarsest levels, respectively. This shows that in addition to process placement (which is the same for all partitions in this case), cache-misses play a very important factor in the packing/unpacking/update times of these planes. The Cache Line Utilization (CLU) factor for the Z-plane is *0.125* at the fine grid level where $P_z = 8$ for $1 \times 1 \times 64$, whereas it is one for the X/Y planes. Thus, even when the DCU and IP-based stride prefetcher in the L1d cache are able to hide the latency by prefetching the needed lines, a penalty must be paid as the Z-plane elements reside in different cache lines.

Table 6.6: Plane Types: Categories of planes based on network elements that they pass through, namely, node/shelf/rack

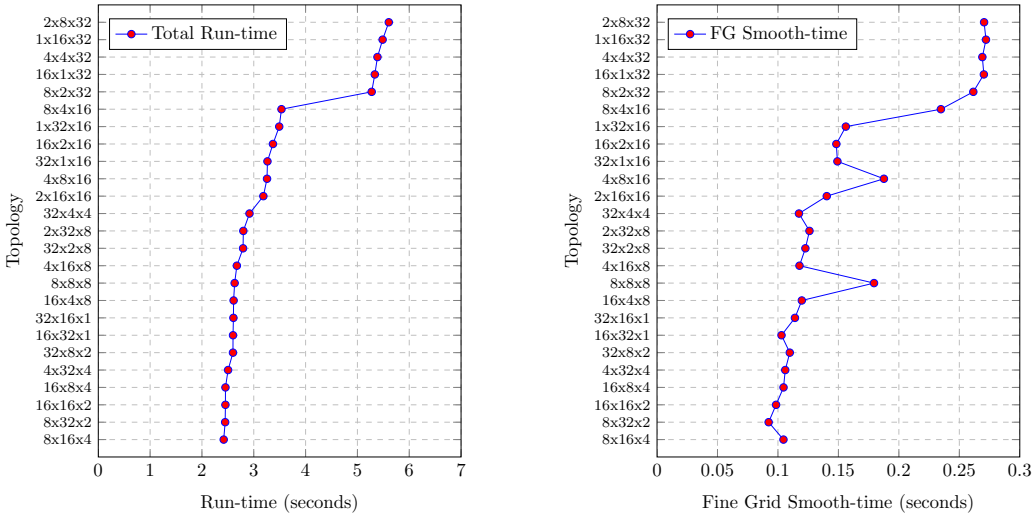
Category	**Description**
C0	Intra-node X-plane
C1	Inter-node Intra-shelf Intra-rack X-plane
C2	Inter-node Inter-shelf Intra-rack X-plane
C3	Inter-rack X-plane
C4	Intra-node Y-plane
C5	Inter-node Intra-shelf Intra-rack Y-plane
C6	Inter-node Inter-shelf Intra-rack Y-plane
C7	Inter-rack Y-plane
C8	Intra-node Z-plane
C9	Inter-node Intra-shelf Intra-rack Z-plane
C10	Inter-node Inter-shelf Intra-rack Z-plane
C11	Inter-rack Z-plane

Table 6.7: Plane Frequency: Number of X/Y/Z Intranode/Intra-shelf/Intra-rack planes for 1-D topologies on ARC2

Topology	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
1x1x64	0	0	0	0	0	0	0	0	120	0	4	2
1x64x1	0	0	0	0	120	0	4	2	0	0	0	0
64x1x1	120	0	4	2	0	0	0	0	0	0	0	0

Table 6.8: Extreme topologies: Run-times for $N = 512^3$, $P = 64$, GCG $= 64^3$, Coarsest iterations = 100, Vcycles = 5, $\nu_1 = \nu_2 = 3, \omega = 1$, FG (Fine Grid), CG (Coarsest Grid), Intel 16.0.2, OpenMPI 1.6.5, ARC2

Topology	**Level**	**FG Smooth-time**	**CG run-time**
1x1x64	4	1.08 sec	0.028 sec
1x64x1	4	0.39 sec	0.010 sec
64x1x1	4	0.36 sec	0.008 sec



(a) Total run-times

(b) Fine Grid smooth-times

Figure 6.18: Total run-time and Fine Grid smooth-times for $P = 512$, Fine Grid $= 1024^3$, Levels $= 6$, Global Coarsest Grid $= 32^3$, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations $= 800$, Vcycles $= 5$, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` $= 8 \times 8 \times 8$

Figure 6.18a shows the total run-times for topologies with $P = 512$ and a fine grid of size 1024^3, i.e. ≈ 1 billion degrees of freedom. The standard topology of $8 \times 8 \times 8$ is outperformed by several topologies which have $D_z \leq 8$. The best performing topology outperforms the standard by 8%, whereas in Figure 6.18b, which shows the fine grid smoothing times only, it outperforms the standard by 41%. Further investigation is needed to ascertain the exact cause of this difference. Although all the topologies were examined on the same set of cores, a possibility of increased congestion in the network due to other user jobs cannot be ruled out as the allocated partition by the job scheduler on our test machine ARC2 is not independent. Thus, our reproducible single node experiments are crucial to testing the validity of our model.

To elaborate on the trend of topology execution times, Figure 6.19a and Figure 6.19b show the multiple node scenario with $P = 96$ and $P = 576$ on ARC3. The Baseline (Base) versions of the predicted topologies $12 \times 8 \times 1$ and $8 \times 12 \times 1$ in Figure 6.19a are both outperformed by the `MPI_Dims_create()` topology (MDC) of $6 \times 6 \times 4$ by 23.93% but the aggressive CO version of $8 \times 12 \times 1$ outperforms the MDC by 6.89%. The Baseline version suggests that as P_z increases to large values (768 in this case), the LRU policy (see Figure 6.7) results in the eviction of data in the cache when $D_z = 1$, as a much larger number of cache lines are accessed before the data is utilized again. For example, with $N = 768$ and $D_z = 1$, $P_z = 768$ so approximately $\frac{768}{8} = 96$ cache lines must be accessed before the data point at $u_{i,j+1,k}$ is accessed again after utilizing it to update $v_{i,j,k}$. With Heuristic Tiling and explicit Vectorization (HT+Vec), the compiler is forced to vectorize as opposed to issuing only a request for Vectorization at optimization levels

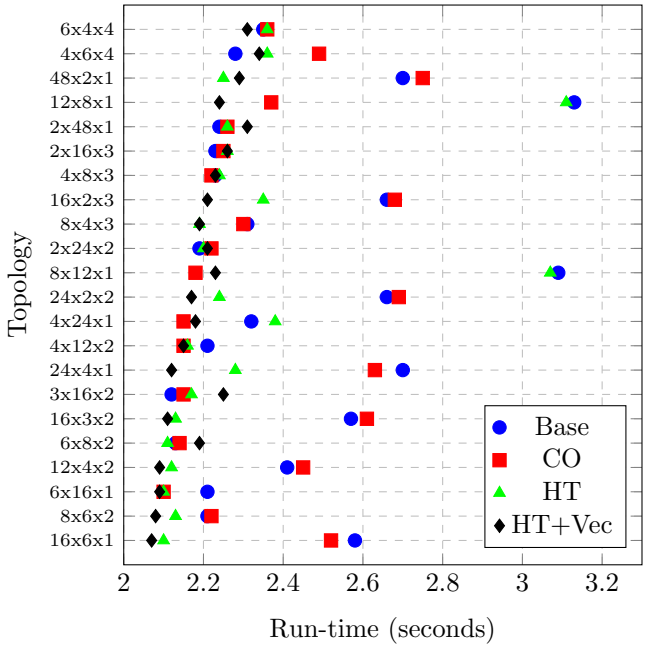
Table 6.9: Weak Scaling Design Experiment: Fixed 2 V(3,3) cycles, 717 coarsest grid iterations for first V-cycle and 712 coarsest grid iterations for second V-cycle

Global Fine Grid	**Cores**	**Levels**	**Global Coarsest Grid**
$128 \times 128 \times 128$	1	4	$16 \times 16 \times 16$
$256 \times 256 \times 256$	8	4	$32 \times 32 \times 32$
$512 \times 512 \times 512$	64	4	$64 \times 64 \times 64$
$1024 \times 1024 \times 1024$	512	4	$128 \times 128 \times 128$

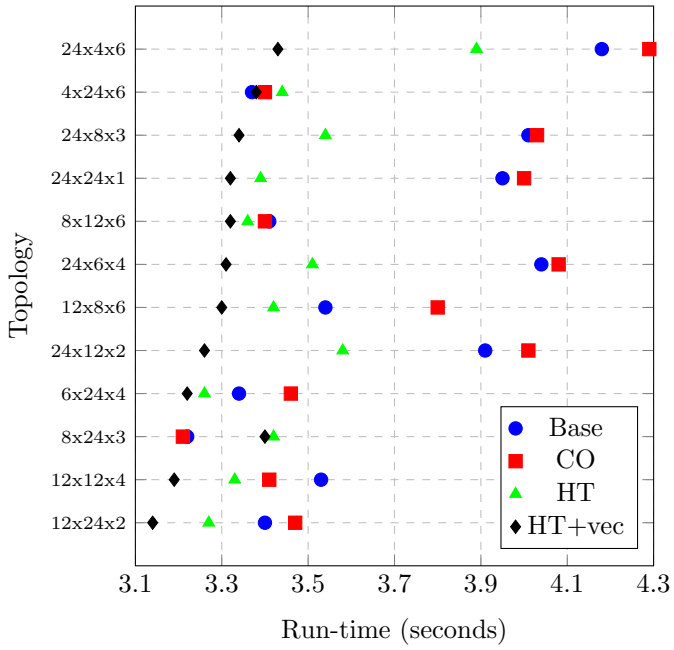
`-O2` and `-O3`: the effect of which is evident with the best execution timings being obtained under a combination of Heuristic Tiling and Vectorization. For explicit Vectorization we use the `#pragma ivdep` option before the innermost loop and this forces the compiler to ignore vector dependencies. The Vectorization report must be checked to ensure that the loop has been vectorized. With $P = 576$, the optimal value of D_z shifts to a value of two and again shows that for extremely large domain sizes, an upward shift in the minimal base value of D_z might be needed to avoid mispredictions.

We now describe the experiment designed for the purpose of *Weak Scaling* Parallel Geometric Multigrid. In Weak Scaling, the problem size per process should remain constant. Given an initial problem size, we increase it by a factor of 8. The number of cores is also increased by a factor of 8. The only exception to this scheme is when our test machine does not have enough cores (as shown in Table 6.10). We further fix the number of levels of the problem, the number of V-cycles and iterations of the coarsest grid in each V-cycle. One method of fixing is to allow the initial problem to converge and to record the number of V-cycles and the coarsest grid iterations in each V-cycle. Next the same experiment is repeated on the same grid after fixing the V-cycles and the number of coarsest grid iterations at each V-cycle but removing the call to `MPI_Allreduce()`. Now the problem size and the number of cores is increased by a factor of eight but the number of levels remain the same. The number of V-cycle and coarsest grid iterations in each V-cycle is kept the same as in the initial problem. This process is repeated for increasing problem sizes and core counts. Table 6.9 illustrates this concept by increasing the problem and processes by a factor of eight, keeping the coarsest grid level the same, fixing the number of V-cycles and also the number of coarsest grid iterations for each V-cycle. It is important to notice that the number of mesh points on each process remain constant and are equal to $= 128^3 + 64^3 + 32^3 + 16^3 = 2396160$. Further, for all configurations, the number of iterations on the first three levels is $= 2 \times V(3,3) = 2 \times (3+3) = 12$ and the number of coarsest grid iterations is $= 717 + 712 = 1429$.

Table 6.10 summarizes the Weak Scaling results by comparing the average performance gain of best performing topologies with respect to the MDC on up to 1024 cores. Our experiment shows that we are always able to find a topology with a $D_z < D_{sz}$, where D_{sz} is the



(a) $P = 96$, $N = 768$, 5 levels, Coarsest iterations = 800, 5 V(3,3) cycles, default `MPI_Dims_create()` $= 6 \times 4 \times 4$



(b) $P = 576$, $N = 1536$, 7 levels, Coarsest iterations = 400, 5 V(3,3) cycles, default `MPI_Dims_create()` $= 12 \times 8 \times 6$

Figure 6.19: Baseline (Base), Compiler Optimized (CO), Heuristically Tiled (HT) and HT + Explicit Vectorization (Vec) total run-time of topologies with Intel 17.0.1, OpenMPI 2.0.2 on ARC3

Table 6.10: Weak Scaling on ARC2: Highest performing Vs standard topology percentage performance gain, Intel 16.0.2, OpenMPI 1.6.5

Cores (Cells/core)	Total Run-time	Fine Grid Smooth
64 (≈2 million)	11.1%	14.4%
512 (≈2 million)	17.3%	36.4 %
1024 (≈1 million)	9.6%	8.8%

Z-dimension returned by MDC, that outperforms the standard topology. With $P = 1024$ and $P = 512$, the Z-planes in the MDC topology are still communicated within a node and the cache-minimizing topologies send/receive larger X/Y planes to/from different racks. Despite inter-rack latencies and larger X/Y planes with cache-minimizing topologies, the cost of sending large-sized Z-planes contributes to the higher execution times of the standard topology. As our test facility does not have 4096 cores, we only weak scale up to 1024 cores (with $\approx$ 1 million cells/core). As opposed to the smaller problem size chosen on ARC2, where tiling and Vectorization yield negligible benefits, we choose a larger problem on ARC3 for Weak Scaling, i.e. 18 million cells/core ($\approx$ 29 billion dof for our largest case). We separately report the Weak Scaling results for the Base, CO, HT and HT+Vec versions as Heuristic Tiling has a significant effect at this problem size (see Table 6.11). Our HT+Vec scheme decreases the overall run-time of the standard topology by 18.45% but also decreases the gain that cache-minimizing topologies have over the standard topology to approximately 4%. Nonetheless, it is important to note the large gain of approximately 19% in the CO versions.

Table 6.12 shows that *Strong* Scaling cache minimizing topologies in Parallel Geometric Multigrid on ARC2 still lead to performance gains up to $P = 256$. The maximum value of EPWSS (Effective Plane Working Set Size) = WPSS + P_yP_z + P_yP_z (for arrays u, v and f in Equation (6.1), Section 6.6, respectively) is $\approx$ 2.5 MB at $P = 128$ but reduces to $\approx$ 1.25 MB at $P = 256$. Since the actual inclusive L3 cache/core is 2.22 MB, similar behaviour of the cache minimizing and standard topology is expected due to the EPWSS completely fitting in the shared Last Level Cache (L3). The Strong Scaling results for ARC3, as shown in Table 6.13, again show that even with a shrinking problem size per core, the cache-minimizing topologies generally outperform the communication volume minimizing topology and thus are also suitable for Strong Scaling until the cores reach a number at which the EPWSS completely fits in the LLC.

6.8.3 19-pt Stencil

The experiments in Parallel Geometric Multigrid till now were based on using a 7-pt stencil for the Smoothing phase and a 27-pt stencil for the inter-grid transfer operators. In this section we use a 19-pt stencil in 3-D in the Smoothing phase and a 27-pt stencil for the inter-grid transfer operators. Figures 6.20a, 6.20b, 6.20c and 6.20d show execution timings of topolo-

Table 6.11: Weak Scaling on ARC3: Highest performing Vs standard topology percentage performance gain, TR (Total Run-time), FG (Fine Grid), Base (Baseline), CO (Compiler Optimized), HT (Heuristically Tiled), Vec (explicit Vectorization), Intel 17.0.1, OpenMPI 2.0.2, Coarsest iterations = 200, $\approx$ 18 million cells/core, Global Coarsest Grid = 48^3

	Base (%)		**CO** (%)		**HT** (%)		**HT+Vec** (%)	
Cores	TR	FG	TR	FG	TR	FG	TR	FG
24	18.56	25.24	18.94	25.81	5.06	2.81	4.10	4.43
192	19.03	25.81	19.51	27.79	4.49	3.81	3.74	3.96
1536	16.71	20.79	18.86	19.75	4.49	1.49	3.76	0.68

Table 6.12: Strong Scaling on ARC2: % performance gain of Cache Minimizing Topology over Standard Topology for Baseline, Compiler Optimized and Heuristically Tiled versions, N=512, 20 V(3,3) cycles, Coarsest iterations = 100, Levels = 6, Intel 16.0.2, OpenMPI 1.6.5

Cores	**Baseline**	**Compiler Opt.**	**Heuristic Tile**
16	15.00%	16.14%	6.16%
32	3.88%	4.04%	8.96%
64	12.69%	12.24%	13.30%
128	7.98%	7.29%	7.85%
256	0.82%	-0.82%	5.50%

Table 6.13: Strong Scaling on ARC3: % performance gain of Cache Minimizing Topology over Standard Topology for Baseline, Compiler Optimized and Heuristically Tiled with Explicit Vectorization versions, N=768, 5 V(3,3) cycles, Coarsest iterations = 400, Levels = 6, Intel 17.0.1, OpenMPI 2.0.2

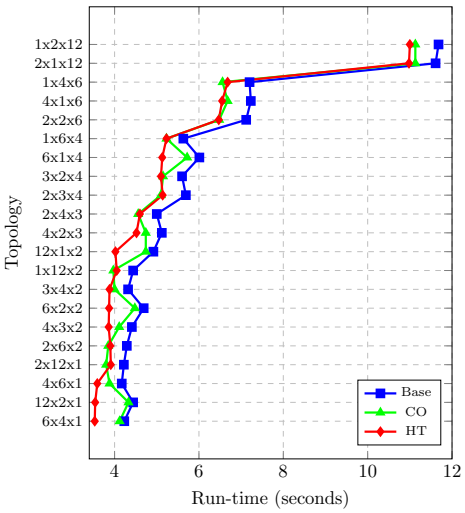
Cores	**Baseline**	**Compiler Opt.**	**Heuristic Tile + Vectorization**
48	9.75%	10.10%	8.58%
96	9.05%	9.48%	8.44%
192	14.06%	13.17%	7.62%
384	7.46%	9.09%	6.25%

gies using a combination of Intel 17.0.1 + OpenMPI 2.0.2 (henceforth called 19I17O2), GNU 6.3.0 + OpenMPI 2.0.2 (henceforth called 19G6O2), Intel 17.0.1 + Mvapich2/2.2 (henceforth called 19I17M2) and GNU 6.3.0 + Mvapich2/2.2 (henceforth called 19G6M2), respectively, on a domain of size 576^3 using a single node of ARC3. For each of these, three variations in the form of Base version for Intel 17.0.1 (`-O2`) and GNU 6.3.0 (`-O2`), aggressive CO for Intel 17.0.1 (`-O3 -xHOST -ip -ansi-alias -fno-alias`) and GNU 6.3.0 (`-O3 -march=native`) and HT (Heuristic Tiling H_1) were tested. As was the case with a 7-pt stencil, Heuristic Tiling alone provided negligible benefits without aggressive compiler optimization with GNU 6.3.0 and hence it was coupled with the latter (i.e. HT+CO (see Figure 6.20b and 6.20d)).

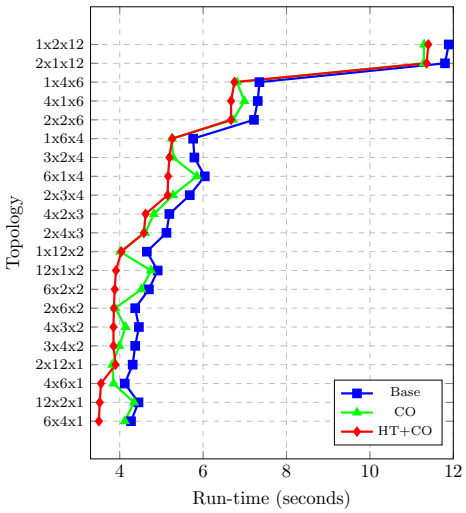
The experiments with the Intel 17.0.1 compiler (19I17O2 and 19I17M2) showed significant difference between the Base version and the CO version. Such a difference was not present in the case of a 7-pt stencil. A plausible reason is the presence of nine data streams in the 19-pt stencil as compared to the five data streams present in the 7-pt stencil. These additional data streams increase the bandwidth pressure on the memory system and hence the optimization level `-O3` is able to generate more efficient code. To study the assembly code obtained after compiling is beyond the scope of the current thesis. In the 19I17O2 version (see Figure 6.20a), the predicted topology of $4 \times 6 \times 1$ outperforms the default MDC of $4 \times 3 \times 2$ by 5.4%, 5.5% and 6.9% in the Base, CO and HT versions, respectively. In the same experiment the topology $6 \times 4 \times 1$ outperforms the default MDC for the Base and HT version and equals its performance in the CO version. As discussed in the section on optimal sub-domain dimensions (see Section 6.6.4), a larger value of D_y as compared to D_x, when $D_x = D_y$ cannot be found, is beneficial as it reduces the WSS (Working Set Size). This can be seen in in all four combinations of compilers and MPI implementations for the Base and CO cases by comparing the $4 \times 6 \times 1$ and $6 \times 4 \times 1$ topology run-times. Interestingly, for 19I17M2, the topology $12 \times 2 \times 1$ surges ahead of the topology $4 \times 6 \times 1$ in the HT version but loses to the latter in the Base and CO versions. Table 6.14 consolidates the results on a single node of ARC3 for various compiler and MPI implementations in terms of the best topologies and performance gains over default MDC for the Base, CO, HT (+CO) versions.

The following observations can be made from Table 6.14:

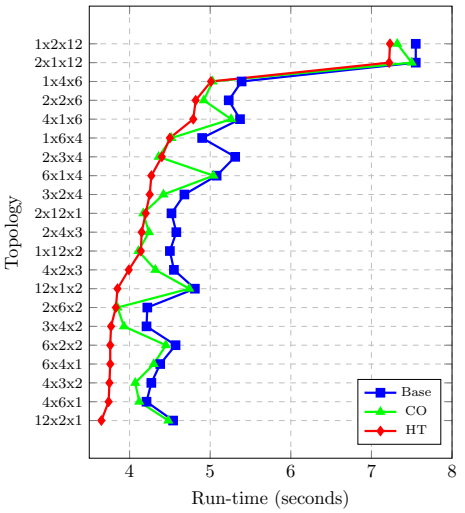
1. The default MDC topology $4 \times 3 \times 2$ is not the optimal for any compiler and MPI implementation (Base, CO, HT or HT+CO versions).
2. For the Base versions, our predicted topology $4 \times 6 \times 1$ is the optimal topology.
3. For the CO versions of 19I17O2 and 19G6O2, we can predict the topology of $2 \times 12 \times 1$ by considering $\frac{4}{2} \times (6 \times 2) \times 1$.
4. For the CO versions of 19I17M2 and 19G6M2, the optimal topology of $2 \times 6 \times 2$ has a higher $D_y = 6$ than the $D_y = 3$ of $4 \times 3 \times 2$ (the former creating a smaller WSS).



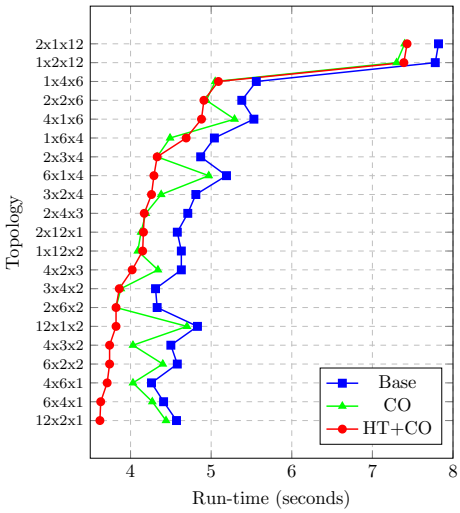
(a) Intel Compiler 17.0.1 and OpenMPI 2.0.2



(b) GNU 6.3.0 and OpenMPI 2.0.2



(c) Intel Compiler 17.0.1 and Mvapich2/2.2



(d) GNU 6.3.0 and Mvapich2/2.2

Figure 6.20: Topology Run-times for $P = 24$, $N = 576$, Levels $= 5$, Coarsest iterations $= 400$, 5 V(3,3) cycles for various combinations of compilers and MPI implementations on ARC3 using a 19-pt stencil in the smoother, default `MPI_Dims_create()` $= 4 \times 3 \times 2$

Table 6.14: Best Topologies and Percentage Gains: Best topologies for Base (Baseline), CO (Compiler Optimized), HT (Heuristically Tiled) versions and percentage gain over the default MDC on a single node of ARC3 for N=576, 5 V(3,3), Levels = 5, Coarsest iterations = 400, 19-pt Parallel Geometric Multigrid

Combination	**Best Base** (% Gain)	**Best CO** (% Gain)	**Best HT(+CO)** (% Gain)
19I17O2	$4 \times 6 \times 1$ (5.4)	$2 \times 12 \times 1$ (7.5)	$6 \times 4 \times 1$ (8.5)
19G6O2	$4 \times 6 \times 1$ (7.6)	$2 \times 12 \times 1$ (7.7)	$6 \times 4 \times 1$ (9.0)
19I17M2	$4 \times 6 \times 1$ (1.4)	$2 \times 6 \times 2$ (5.4)	$12 \times 2 \times 1$ (2.70)
19G6M2	$4 \times 6 \times 1$ (5.30)	$2 \times 6 \times 2$ (5.2)	$12 \times 2 \times 1$(3.20)

5. The MPI implementation does affect the communication timings, in turn affecting the overall performance gains.

Figure 6.21 shows the performance of various topologies for Parallel Geometric Multigrid using $P = 96$ cores when the number of unknowns is $768 \times 768 \times 768$. The figure only shows a subset of topologies for which execution times were obtained but as can be seen, there are many topologies which outperform the default MDC of $6 \times 4 \times 4$ in the Base, CO and HT versions. Although the MDC outperforms the topologies $T_1 = 12 \times 8 \times 1$ and $T_2 = 8 \times 12 \times 1$ in the Base version, it is outperformed by the latter topologies in the CO and HT versions. The T_1 and T_2 topologies suffer from a large P_z and hence the LRU policy can adversely affect the performance. The `-O3` option is able to optimize these topologies to a significant extent and hence these outperform the MDC topology by 7.9% (T_1) and 11.9% (T_2) in the CO version. The highest performing topology of $16 \times 6 \times 1$ in the HT version can be obtained from $8 \times 12 \times 1$ by considering $(8 \times 2) \times \frac{12}{2} \times 1$ and outperforms the default MDC by 17.7%. In summary, the results obtained by using a 19-pt stencil in the Smoothing phase of Parallel Geometric Multigrid yields results similar to those obtained by using the 7-pt stencil except for appropriate quantitative differences due to increased data streams and the changing communication pattern.

6.9 Model Accuracy

We define the accuracy of our model as the fraction of those topologies predicted to outperform the default topology that do outperform it in practice. Formally, let n_p be the total number of predicted topologies for P cores and let t_p be the execution time of the predicted topology and t_{MDC} that of the MDC topology. If $\widetilde{n_p}$ is the number of predicted topologies for which $t_p < t_{MDC}$, then the accuracy of the model is $\frac{\widetilde{n_p}}{n_p} \times 100$ with P processor cores. A topology is predicted to be better than the default MDC if the predicted cache-misses is fewer in comparison to the MDC. The best predicted topology is the one with the fewest cache-misses, irrespective of the communication volume.

For calculating the accuracy of the model, we categorize the topologies into four classes.

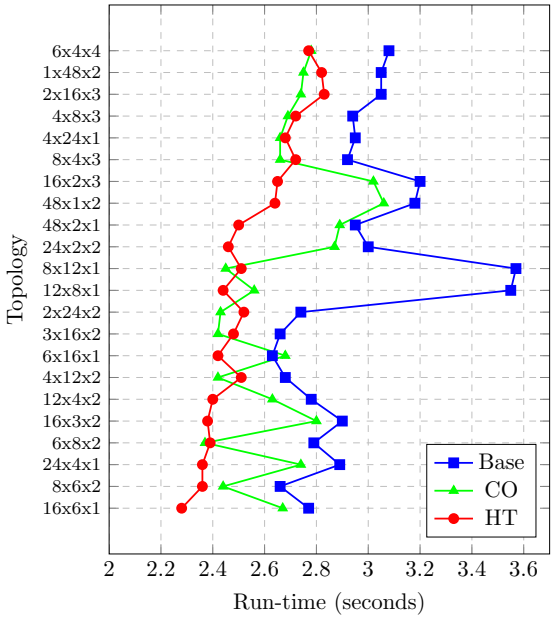


Figure 6.21: 19-pt Smoother in Multigrid, Cores=96, N=768, Levels=5, Coarsest iterations=800, 5 V(3,3) cycles, Intel Compiler 17.0.1, OpenMPI 2.0.2

The four classes are True Positive (True + or TP), True Negative (True - or TN), False Positive (False + or FP) and False Negative (False - or FN). The meaning of these classes is explained below.

1. *True Positive*: A topology which is predicted to outperform the default MDC topology and also experimentally outperforms it.

2. *True Negative*: A topology which is predicted to outperform the default MDC topology but does not experimentally outperform it.

3. *False Positive*: A topology which is predicted not to outperform the default MDC topology and also experimentally does not outperform it.

4. *False Negative*: A topology which is predicted not to outperform the default MDC topology but experimentally outperforms it.

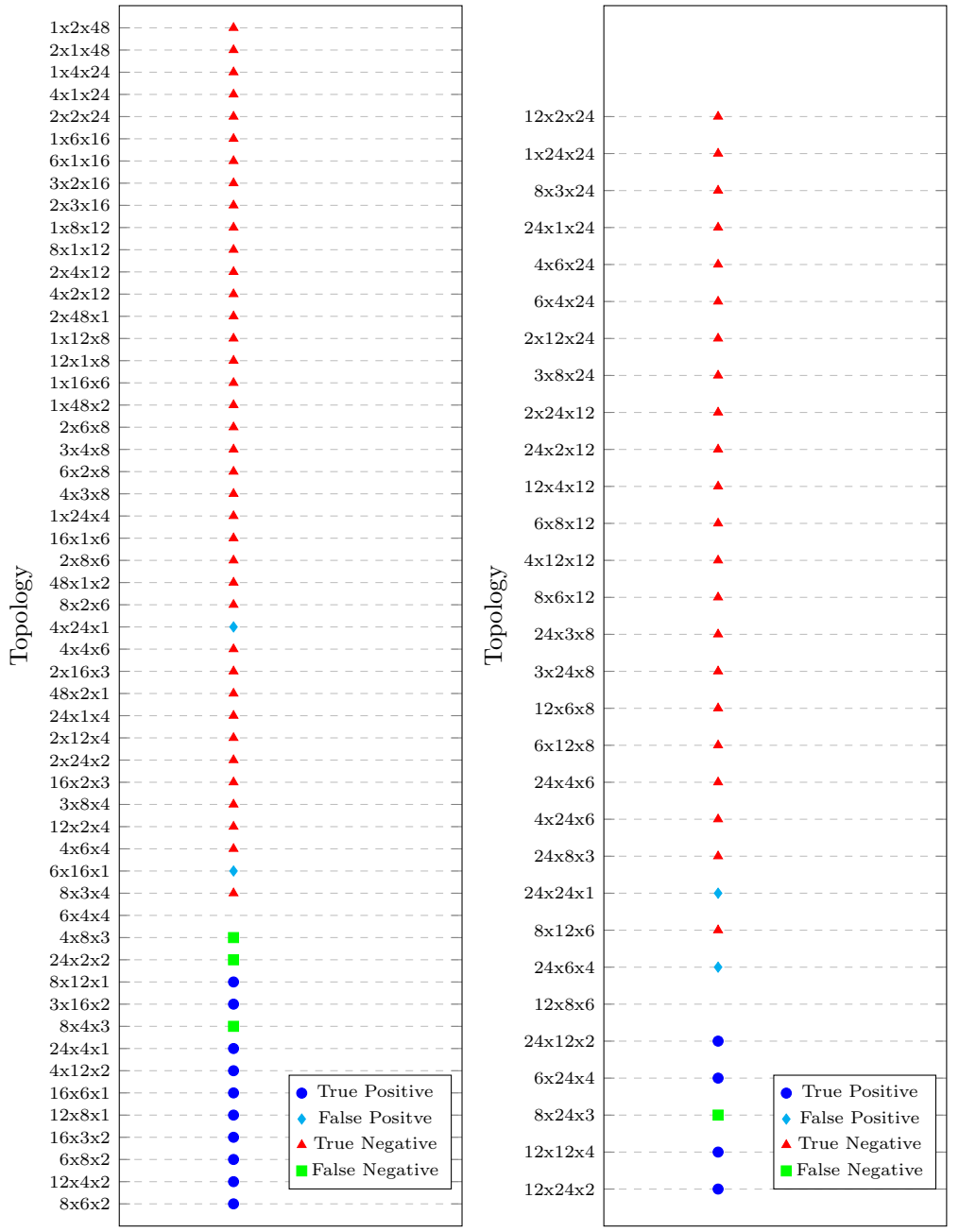
Table 6.15 shows the accuracy of the model with respect to the hardware clusters ARC2 and ARC3 for various core counts, domain sizes, compilers and MPI implementations. It should be noted that not all the predicted topologies are able to reach a pre-defined Multigrid level with a particular domain size and thus such predicted topologies are not counted towards calculating the accuracy. For example, with $P = 768$ only two predicted topologies were experimentally valid (see Table 6.15). Further, it is the predicted topologies with a large $|D_x - D_y|$ that are outperformed by the MDC topology and constitute the False Positives (FP). For example, the

Table 6.15: Model Accuracy: P = number of cores, N = Domain size, n_p = Number of predicted topologies, $\widetilde{n_p}$ = Predicted topologies for which $t_p < t_{MDC}$, MDC = `MPI_Dims_create()` topology, Accuracy (True +) = $\frac{\widetilde{n_p}}{n_p} \times 100$

							Accuracy	
P	**N**	n_p	$\widetilde{n_p}$	**MDC**	**Compiler**	**MPI**	True +	True -
				ARC2				
16	512^3	3	3	$4 \times 2 \times 2$	Intel 16.0.2	OpenMPI 1.6.5	100%	77.78%
64	512^3	7	7	$4 \times 4 \times 4$	Intel 16.0.2	OpenMPI 1.6.5	100%	90.90%
512	1024^3	9	8	$8 \times 8 \times 8$	Intel 16.0.2	OpenMPI 1.6.5	88.89%	93.34%
				ARC3				
24	576^3	4	4	$4 \times 3 \times 2$	Intel 17.0.1	OpenMPI 2.0.2	100%	100%
24	576^3	4	4	$4 \times 3 \times 2$	GNU 6.3.0	OpenMPI 2.0.2	100%	100%
24	576^3	4	3	$4 \times 3 \times 2$	Intel 17.0.1	Mvapich2/2.2	75%	100%
24	576^3	4	3	$4 \times 3 \times 2$	GNU 6.3.0	Mvapich2/2.2	75%	100%
48	768^3	10	10	$4 \times 4 \times 3$	Intel 17.0.1	OpenMPI 2.0.2	100%	76%
96	768^3	12	12	$6 \times 4 \times 4$	Intel 17.0.1	OpenMPI 2.0.2	100%	88.89%
96	768^3	12	10	$6 \times 4 \times 4$	GNU 6.3.0	Mvapich2/2.2	83.34%	97.14%
192	768^3	6	6	$8 \times 6 \times 4$	Intel 17.0.1	OpenMPI 2.0.2	100%	82.60%
384	768^3	6	5	$8 \times 8 \times 6$	Intel 17.0.1	OpenMPI 2.0.2	83.34%	100%
576	1536^3	6	4	$12 \times 8 \times 6$	Intel 17.0.1	OpenMPI 2.0.2	66.67%	100%
768	1536^3	2	2	$12 \times 8 \times 8$	Intel 17.0.1	OpenMPI 2.0.2	100%	100%
1536	3072^3	6	4	$16 \times 12 \times 8$	Intel 17.0.1	OpenMPI 2.0.2	66.67%	100%

predicted topology of $24 \times 6 \times 4$ with $P = 576$, $N = 1536^3$ (see Table 6.15) is outperformed by the default MDC $12 \times 8 \times 6$ by a thin margin of 0.30% due to a very high $|D_x - D_y| = 18$. We do not count the False Negatives (FN) towards calculating the accuracy. In addition to predicting high performing cache-minimizing topologies, we are also able to successfully prune out inefficient topologies with a high degree of accuracy (True Negative accuracy shown in Table 6.15). Further, the False Negative topologies i.e. topologies whose performance our model predicts to be worse than the MDC performance but which experimentally outperform the MDC, are the ones which are closer in performance to that of the default `MPI_Dims_create()` topology. As an example, for $P = 96$, $N = 768^3$, using GNU 6.3.0 and Mvapich2 on ARC3, the False Negative topology of $24 \times 2 \times 2$ outperforms the MDC by 0.88% only.

As representative cases of the model accuracy, we classify the topologies into these four classes for $P = 96$ and $P = 576$ on ARC3 as shown in Figures 6.22a and 6.22b, respectively. As is desirable, the representative cases show a very small fraction of False Positives and False Negatives. It can be seen from the classification of topologies that the power of the model also lies in correctly predicting and eliminating the true negatives.



(a) P=96, N=768, GNU 6.3.0, Mvapich2 (b) P=576, N=1536, Intel 17.0.1, OpenMPI 2.0.2

Figure 6.22: Prediction classes for representative cases of model accuracy on ARC3, where the entry with no symbol is the default MDC (`MPI_Dims_create()`) partition

6.10 Summary

Traditionally, domain partitioning has been considered as a function of only load-balance and communication volume. Thus, the orthodox approach aims to achieve maximal load balance and minimize communication volume. We challenge this approach and introduce a third dimension to the problem of domain partitioning: Cache-misses at the sub-domain level. Thus, instead of only optimizing cache-misses through spatial and temporal methods after the domain partitioning, we analyze the cache-misses at the sub-domain level before performing domain partitioning and use this to predict optimal domain partitions in parallel Geometric Multigrid (GMG). To this effect, we extend the high level quasi-cache-aware model developed in Chapter 4 with the assumption that the interpolation/restriction is proportional to the smoothing time but dominated by the latter. The model estimates the cache-misses for the update of the Independent Compute and the update/packing/unpacking of the Dependent Planes. Though we develop our model using a 7-pt stencil, the methodology can be applied to a 19-pt or 27-pt stencil. Our numerical tests show the same qualitative results with appropriate quantitative differences. Upon subsequent minimization with respect to sub-domain dimensions, the two most important factors needed to obtain optimal domain partitions that emerge out of the model are (see Chapter 4): (i) The balance between the X and Y sub-domain dimensions and ; (ii) Maintaining a Cartesian process-dimension $1 \leq D_{zoptimal} \leq D_{sz}$, where D_{sz} is the Z-dimension returned by the default `MPI_Dims_create()` function.

We emphasize and elaborate the factors affecting optimal sub-domain dimensions namely, Independent Compute, Plane cache-misses, Prefetch, Vectorization, Communication Volume, and the LRU eviction policy. The two most significant factors out of these factors are: Plane cache-misses and the Communication Volume. We place stress on maintaining a balance between the cost of growing communication volume when maximizing the unit-stride dimension and the growing cost of packing/unpacking/updating the Z-plane when the communication volume is minimized. Our experiments on single and multiple nodes expand on the three most important factors: Independent Compute, Dependent Plane and Communication Volume. The single node experiments further show that, even without communication, weakly scaling a problem on a SMP does not keep the time constant due to the rising contention for the shared Last Level Cache. Topologies efficiently executing the Independent Compute are not optimal when communication is added and thus optimality requires a balance between compute cache-misses and the overhead of communication. Further, we develop a light-weight run-time heuristic for tiling, functioning at all but the coarsest level of GMG, which is close to optimal for high performing topologies, given that exhaustive tiling leads to a combinatorial explosion of the tiling space. The experiments for process placement within a node i.e. `--bind-to-core --bysocket` and `--bind-to-core --bycore` yield similar results for plane communication costs. We further calculate the accuracy of the model by considering different compilers and MPI implementations.

Chapter 7

Conclusions and Future Work

The work in this thesis has focused on *domain partitioning* for single and multilevel *stencil-based* codes. We created a high level mathematical model to establish a relation between *cache-misses* and domain partitioning. In addition, we qualitatively explored factors such as *Cache-Line Utilization* (CLU), *Least Recently Used* (LRU) cache eviction policy, *Vectorization* and their role in determining optimal sub-domain dimensions. Our approach differs from the traditional method of domain partitioning in the sense that the traditional method only aims to minimize the communication volume (assuming a load-balanced problem). Our discussion emphasizes the idea that the problem of domain partitioning is dependent on multiple factors as opposed to communication volume only. Although we used the *finite difference* discretization of *Elliptic PDEs* and the *Jacobi* point iterative method to build our *quasi-cache-aware* model, we discuss its applicability and generality in a multitude of scenarios, ranging from *non-linear PDEs* to a non-overlap of communication with computation. Furthermore, we extended and tested our model on *Adaptive Mesh Refinement* (AMR) and parallel *Geometric Multigrid* (GMG) - both being indispensable tools in *Scientific Computing*. Their importance is further reflected by the fact that they are candidate algorithms for *Exascale*. The current Chapter discusses some more detailed conclusions, followed by suggestions for related future research directions.

7.1 Conclusions

At a high level, we can concisely state the conclusion of our work as follows: "It is not sufficient to minimize only the communication volume for obtaining optimal domain partitions for single or multilevel stencil-codes, rather it is necessary to optimize the balance between minimizing cache-misses and the communication volume."

In fact our model indicates that the optimal domain partitions are "close to 2-D" rather

than cubic (or almost cubic) as suggested by the orthodox method of domain partitioning. Our *Weak and Strong Scaling* results on single, uniform grids showed that although topologies which are closer to 2-D communicate more elements than the default `MPI_DIMS_CREATE()` topology of MPI, they offer better performance due to fewer cache-misses in packing/unpacking/update of the *Dependent Planes* (DP). Further, cache-misses are a much more significant factor to optimize than the communication volume, when the *Working Set Size* (WSS) does not fit into the cache-hierarchy. As the WSS starts fitting in the cache-hierarchy, cache-minimizing domain partitions still show a performance gain over communication volume minimizing partitions. Our experiments are in agreement with the theory we formulate in the current research, that a Z-plane which is perpendicular to the direction in which data is contiguous in memory is the costliest plane to communicate and update as compared to similar sized X/Y planes. This is a weakness of the traditional method of domain partitioning i.e. it does not take into account the different costs of the various plane types in communication and updating the solution. A weakness of our model is that it is only applicable to structured 3-D grids with cuts parallel to the *Cartesian Axes* and cannot be extended to unstructured grids. As our model does not take into account the architectural details except for the cache-line size, it is not dependent on a specific architecture. This is also verified by experiments conducted throughout the work on two different High Performance Computing clusters. Thus, the model is independent of the software-hardware ecosystem and the optimal partitions only depend on the data layout of the language used to implement the stencil-based application.

The cache-misses minimizing topologies outperform the default communication minimization topology returned by the `mpi_dims_create()` subroutine of the MPI specification in all the cases that we tested for single uniform grids using *BoxLib* - the parallel AMR framework. To this effect we implemented and simulated an MPI Cartesian topology of processes and replaced the default box distribution policy of BoxLib with our topology. Using non-cubic blocks/boxes is the optimal choice for single level, uniform meshes in BoxLib. Thus, even in the absence of overlap of communication with computation, our hypothesis remains true. When the use of non-cubic boxes is extended to load-imbalanced AMR, the performance gain decreases, with no gain in some cases. This can be attributed to the change in communication pattern, non-overlap of communication with computation, the load-balancing criterion, the increase in metadata and the automatic box-distribution strategy in BoxLib. Further, the coding effort needed to adapt BoxLib to use non-cubic blocks is significant. We conclude that for maximum performance gains it is best to used non-cubic partitions in single grids but that the performance gains are not significant, as compared to the coding effort spent in adapting BoxLib to non-cubic blocks for AMR. Nevertheless, it may still be stated that the communication minimizing topology/cubic-partitions are not generally optimal for AMR codes.

Parallel Geometric Multigrid (GMG) shows the same behaviour as uniform meshes in the

sense that our model holds completely for GMG. Thus, the optimality of the domain partition at the finest grid level governs the overall performance. This is in-line with the theory of Multigrid, in that it is at the finest grid that maximum work in Multigrid is performed. In other words, the diminishing gap between the communication volume minimizing topologies and the cache-minimizing topologies at coarser levels does not have a significant impact on the performance gain. Further, all the stencil operators, i.e. the 7-pt or the 19-pt stencil in *smoothing* and the 27-pt stencil in *Restriction*, have similar expressions for cache-miss equations that differ only quantitatively.

In addition to cache-misses, we identified and qualitatively investigated some other *Serial Control Parameters* (SCPs), namely, Vectorization, Cache Line Utilization (CLU) and the Least Recently Used (LRU) cache eviction policy that affect optimal sub-domain dimensions. The LRU policy tries to minimize the unit-stride dimension for optimality. The reason for this is because if the unit-stride sub-domain dimension is small, there is higher probability that the recently used data points will still be in the cache when they are needed again. Both Vectorization and the CLU need the maximal value of the unit-stride dimension for optimal performance because it ensures an uninterrupted stream of data points that do not contain the *ghost* points. While updating the *Independent Compute* (IC) kernel, the ghost points can act as "bubbles" in the data stream, thereby lowering the performance. While updating the solution at mesh points, the Jacobi iterative algorithm utilizing the 7-pt (or 19-pt or the 27-pt) stencil requires data elements from three adjacent planes. We defined a quantity called the *Working Plane Set Size* (WPSS) to indicate the total size of these three planes. If two topologies result in the same sub-domain volume, then the topology which has a smaller WPSS generally results in higher performance. In summary, domain partitioning is a complex function of multiple SCPs and not just a simple function of the communication volume. We emphasize and conclude that in the light of an evolving tightly-coupled software-hardware ecosystem, domain partitioning must be re-investigated for performance.

7.2 Future Work

While creating the model for minimizing cache-misses, we take into account only the *cache-line* size and ignore the other architectural details. The final inferences from our model are also *oblivious* of the cache-line size and depend only on the core count. The model can be improved by taking into account the problem size and architectural details. For example, the cache-sizes and the cache-hierarchy can be incorporated to make the model more specific to an architecture. The overall aim here therefore, is to move from a high level model to a low level model.

We create our model without taking into account any form of *cache-tiling*. Cache-tiling helps to reduce cache-misses by keeping the Working Set Size (WSS) in the cache memory for

re-usability and to reduce main memory accesses. There is a vast literature body investigating the techniques for tiling and the associated frameworks. It would be interesting to see the model adapted to the common practice of tiling. The selection of both shape and size of the tile is an important research problem and hence while evaluating our model we felt that there is a need to create super light-weight and dynamic cache-tiling *heuristics* for auto-tuning the tile size. The reason is that an exhaustive approach of searching for the optimal tile size and shape involves a large number of combinations of tile sizes and may actually take much more time to execute than the actual application which utilizes this tile. Although our model does not incorporate tiling as mentioned above, for some experiments we implemented and evaluated the 2-D tile shape proposed in [6]. Further, we proposed and implemented a simple, light-weight, tiling heuristic based on the number of arrays in the WSS and the shared L3 cache. Our experiments with this heuristic showed that the high performing topologies benefited from the automated tile size selection but at the same time the performance of the low performance topologies degraded. This may be taken as a theoretical but not a practical drawback because in practice, real codes will avoid non-optimal low performing topologies. A thorough evaluation of the proposed (and other) tiling heuristics for various stencil sizes, iterative methods and hardware architectures can form an interesting area of research.

We performed our research based on a single ghost layer and thus the model can be extended to model the cache-misses for multiple ghost layers, as might be required for higher order differential operators. We do not devise any separate theory of cache-misses for multiple sub-domains per MPI process. Such cases are extensively observed with AMR frameworks where the challenge is to optimize communication/load-balance and thus, this forms a research direction. A natural extension to the model is to use it for *Hybrid programming* which may use *MPI* for domain level parallelism and *OpenMP* for thread-level parallelism. This creates a two-level problem in the sense that first an MPI process level sub-domain is created which is again partitioned among threads. Thus, the partition space for optimal combinations of processes and threads needs to be explored using a possible extension of our model. Hybrid programming is being seen as a key programming model at the Exascale level and maximizing performance demands an optimal domain/thread level decomposition. The problem of optimal domain partitioning can also be extended to Graphics Processing Units (GPUs). GPUs use a grid of thread-blocks to simultaneously compute solutions at multiple grid points. These grid of thread blocks can be utilized as 1-D, 2-D or 3-D thread blocks, and thus, the communication pattern is governed by the decomposition. It would be interesting to investigate the challenges associated with domain partitioning on such heterogeneous platforms consisting of multicores and GPUs.

As shown in the current work, a complete support for non-cubic boxes in BoxLib can be advantageous for certain applications and thus there is a need to modify BoxLib routines such

that they support them seamlessly. Specifically, as discussed with the developers, the future implementations of BoxLib can make `cluster_minwidth` a vector. To simulate a *block-structured* behaviour with BoxLib, one needs to tag all the cells and manually adjust the value of `cluster_min_eff`. Instead of tagging all the cells in a block for pure block-structured behaviour, BoxLib can provide a single *boolean* variable to switch this behaviour on or off. Though it is easy to modify a structure in BoxLib, as the source code is openly available, it is still difficult to follow the chain of interaction that such a change will propagate. Further, the behaviour of `cluster_min_eff` should be much more clearly defined and realized. As for refinement, a refinement flag can be associated with each *Fab* object to detect whether it has been refined or not. In the current version of BoxLib, the only way to examine refinement is to check the tagged box array associated with a Fab object, or if performing a geometry based refinement, to check the range of coordinates. Finally, a significant improvement to BoxLib's parallel performance would be to decouple the non-blocking point-to-point MPI operations from the wait calls to enable overlapping of communication with computation.

In parallel Geometric Multigrid, a point that we did not focus on is the solve time for the coarsest grid which can have a significant effect on the overall execution timings. A future direction would be to apply our model on the coarsest grid solve on a subset of processes. It can be noted that choosing both the number of processes and the particular *ranks* to solve the coarsest grid problem is non-trivial. To the best of our knowledge no literature exists on how to choose this optimal subset of processes. There lies a further opportunity in examining the benefits of separating the communicating and computing threads (while using Hybrid programming) when solving the coarsest grid problem on a subset of processes.

Appendices

Appendix A

Eager and Rendezvous Protocols

In Chapter 4 (see Section 4.4.3.1), we discussed how a *Dependent Plane* (DP) is packed at the sending MPI process. Explicit packing involves copying the data from the 3-D solution array to a temporary 1-D user defined array. Implicit packing involves copying the data from the 3-D solution array to an MPI buffer. We ignore the cost of cache-misses when copying the data to this 1-D user array or the MPI buffer as the data from the 3-D solution array can be sent directly to the receiving MPI process, depending on the protocol used by the MPI implementation.

We discuss here the two most common protocols used for transferring data from the sender to the receiver MPI process: the *Eager* protocol and the *Rendezvous* protocol. It is to be noted that these protocols are not part of the MPI standard [152]. The Eager protocol is used for sending short messages and it allows the message to be sent immediately, without checking for a posted receive by the destination. Such messages are then buffered at the destination and copied to the application buffer when a corresponding receive is posted. The Rendezvous protocol is typically used for long messages and is based on a *Request-to-Send* (RTS) and *Clear-to-Send* (CTS) technique. This technique involves the source sending a matching MPI data to the destination, which then responds with a CTS message when it is ready to receive the data. Thus, a Rendezvous protocol involves a round-trip of RTS and CTS messages. This round-trip is avoided by the Eager protocol by directly sending the data to the destination. Thus, the Eager protocol optimizes the *latency* whereas the Rendezvous protocol optimizes the resource consumption [152]. There are other protocols that are used in OpenMPI [127] other than the two mentioned above. OpenMPI uses a variant of the protocols described in [153] and details can be found on the OpenMPI FAQ page [150].

Appendix B

BoxLib - Configuration and Profiling

In Chapter 5, we discussed the set-up and solve phases of our implementation of adaptively refined meshes using BoxLib. We further discussed the major modifications needed to adapt the library to seamlessly support non-cubic boxes. This section discusses some precautions, compilation adjustments and profiling of BoxLib applications using Scalasca.

B.1 Deallocating variables for program re-run

To record and compare the execution times of different box-sizes, the same program is run on the same set of cores using different box-sizes. When the program is run again, some variables need to be deallocated. These should ideally be the ones allocated by the user but a variable named `amr_ref_ratio` in `ml_boxarray.h` needs to be deallocated as well (though it is part of the BoxLib library). This variable is part of the `ml_boxarray_module` declared as shown in Listing B.1.

```
1 integer, allocatable, private, save :: amr_ref_ratio (:,:)
```

Listing B.1: Declaration of amr_ref_ratio

Consequently, a subroutine to deallocate it was written and is shown in Listing B.2.

```
1 subroutine amr_ref_ratio_deallocate ()
2     deallocate(amr_ref_ratio)
3 end subroutine amr_ref_ratio_deallocate
```

Listing B.2: Deallocation subroutine for amr_ref_ratio

B.2 Compiling on ARC3

The BoxLib program does not converge when GNU/6.3.0 compilers are chosen on ARC3 with OpenMPI/2.0.2. The exact reason cannot be determined but is possibly due to an incompatible Fortran90 datatype declared in the `mpi.h` file. Thus, gnu/native was chosen along with OpenMPI/2.0.2. Further, the Fortran library files in OpenMPI/2.0.2 have changed names to `-lmpi_mpifh` instead of `-lmpi_f90` and `-lmpi_f77`. This can be seen by executing `$ mpif90 -show` on ARC2 or ARC3. Thus, the last few lines in the file `Software/BoxLib/Tools/F_mk` must be changed as shown in Listing B.3 before compiling on ARC3:

```
1 ifdef MPI_HOME
2   mpi_include_dir = $(MPI_HOME)/include
3   mpi_lib_dir = $(MPI_HOME)/lib
4   mpi_libraries += -lmpi -lmpi_mpifh    # this replaces -lmpi_f90 and -lmpi_f77
5   CC = mpicc
6   CXX = mpic++
7   FC = mpif90
8 endif
```

Listing B.3: OpenMPI 2.0.2 Fortran library files -lmpi_mpifh

B.3 Profiling BoxLib using Scalasca on ARC3

An application implemented using BoxLib can be profiled using the *Scalasca* [121] profiling tool (see Chapter 3). In Chapter 5, we used Scalasca interfaced with the PAPI [120] library to capture cache-misses for various sub-routines. This section describes how to use Scalasca to profile an application using BoxLib.

First the module Scalsaca and *Score-P* must be loaded using the `module load` command. Two files in the BoxLib library need to be modified, namely,

1. `BoxLib/Tools/F_mk/comps/gfortran.mak`
2. `BoxLib/Tools/F_mk/GmakeMPI.mak`.

For the first file the following lines near the beginning of the file need to be changed to use the `scalasca -instrument` command along with the `--compile` option. The `--compile` option forces the *instrumentation* of user functions (by default, `scalasca -instrument` only captures the MPI functions). This change is shown in Listing B.4.

```
1   FCOMP_VERSION := $(shell $(COMP) -v 2>&1 | grep 'version')
2   FC  := scalasca -instrument --compile $(COMP)
3   F90 := scalasca -instrument --compile $(COMP)
```

Listing B.4: Compiling with Scalasca

Table B.1: MPI Fortran libraries: for Open MPI 2.0.2 and Intel MPI 2017.1.32

OpenMPI 2.0.2	IntelMPI 2017.1.32
`-lmpi -lmpi_mpifh`	`-lmpi -lmpifort`

The second step is to insert the command `scalasca -analyze` in the submitted script file. An example of the script file is shown below in Listing B.5.

```
1 # !/bin/bash
2 #$ -l h_rt=00:30:00
3 #$ -l nodes=1,ppn=24,tpp=1
4 #$ -cwd -V
5 #$ -m be
6 scalasca -analyze mpirun ./main.Linux.gfortran.mpi.exe
```

Listing B.5: Shell script modification with Scalasca

After the program runs successfully, it produces a directory having the prefix `scorep`. A directory for this name must not exist before the program is executed otherwise it interferes with the creation of a new directory. As the third step, the `scalasca -examine` command must be run on this directory. This command produces a filename with the extension `cubex`. These `cubex` files can then be examined using the graphic analyzer software called the CUBE. An example of running this command is shown below in Listing B.6.

```
1 scalasca -examine scorep_main_O_sum/
```

Listing B.6: scalasca -examine

If only a textual output is needed, the user can execute `scalasca -examine -s` at the command line.

B.4 MPI libraries for OpenMPI and IntelMPI

In Chapter 5, our experiments use both OpenMPI 2.0.2 and Intel MPI 2017.1.132 and the appropriate library for each is needed by BoxLib. On ARC3 the libraries for OpenMPI 2.0.2 and Intel MPI 2017.1.132 are different and the file `GMakeMPI.mak` must be changed to reflect this. We can find the correct libraries for the specific MPI by loading the correct module and then executing `$ mpif90 - show`. This command shows the correct libraries which must be linked. These libraries are shown in Table B.1.

B.5 Compiling with Intel compiler

An application in BoxLib can be compiled using different compilers. For example, to use the Intel compiler, the compiler being pointed to in the *GNUMakefile* should be changed to

`comp:=Intel`. Further, as there was no entry for the Intel 17 compiler in the `Linux_intel.mak` file, a manual entry having the same options as the Intel 16 compiler was created. The various compiler flags can be specified in this file as well but there is an option to specify the flags at the linking stage as well (specified in the GNUMakefile).

Bibliography

[1] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer, "Paramesh: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, no. 3, pp. 330–354, 2000.

[2] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for AMR applications design document," *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)*, 2009.

[3] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pp. 33–41, IEEE, 2000.

[4] "GitHub - BoxLib-Codes/BoxLib: Block-Structured AMR Framework." `https://github.com/BoxLib-Codes/BoxLib`. Accessed: 2017-04-22.

[5] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[6] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, p. 32, IEEE Computer Society, 2000.

[7] S. Saini, J. Chang, and H. Jin, "Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pp. 25–51, Springer, 2014.

[8] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of scaling algebraic multigrid across modern multicore architectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 275–286, IEEE, 2011.

[9] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling Hypre's Multigrid Solvers to 100,000 Cores," in *High-Performance Scientific Computing*, pp. 261–279, Springer, 2012.

[10] W. D. Gropp, "Parallel computing and domain decomposition," in *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, pp. 349–361, Publ by Soc for Industrial & Applied Mathematics Publ, 1992.

[11] Y. Notay and A. Napov, "A massively parallel solver for discrete Poisson-like problems," *Journal of Computational Physics*, vol. 281, pp. 237–250, 2015.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 4, IEEE Press, 2008.

[13] K. Datta, *Auto-tuning stencil codes for cache-based multicore platforms*. PhD thesis, University of California, Berkeley, 2009.

[14] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[15] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde, "Memory characteristics of iterative methods," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, p. 31, ACM, 1999.

[16] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *Proceedings of the 2005 Workshop on Memory System Performance*, pp. 36–43, ACM, 2005.

[17] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[18] S. M. F. Rahman, Q. Yi, and A. Qasem, "Understanding stencil code performance on multicore architectures," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, p. 30, ACM, 2011.

[19] "BoxLib/AMReX Case Study." `http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/application-case-studies/boxlib-case-study/`. Accessed: 2018-05-26.

[20] "Home Page - Exascale Computing Project." `https://www.exascaleproject.org/`. Accessed: 2017-11-15.

[21] W. A. Strauss, *Partial Differential Equations: An Introduction*. Wiley, 2008.

[22] G. D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.

[23] W. F. Ames, *Nonlinear partial differential equations in engineering*, vol. 18. Academic press, 1965.

[24] "MPI: A Message-Passing Interface Standard." `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. Accessed: 2018-03-04.

[25] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*. Academic press, 2000.

[26] E. Kreyszig, *Advanced engineering mathematics*. John Wiley & Sons, 2010.

[27] S. C. Chapra and R. P. Canale, *Numerical methods for engineers*, vol. 2. McGraw-Hill New York, 1998.

[28] D. Hutton, *Fundamentals of finite element analysis*. McGraw-Hill, 2004.

[29] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[30] M. Schäfer, *Computational engineering: Introduction to numerical methods*. Springer, 2006.

[31] D. Gottlieb and S. A. Orszag, *Numerical analysis of spectral methods: theory and applications*, vol. 26. Siam, 1977.

[32] C. Canuto, M. Y. Hussaini, A. Quarteroni, A. Thomas Jr, *et al.*, *Spectral methods in fluid dynamics*. Springer Science & Business Media, 2012.

[33] Y. Saad, *Iterative methods for sparse linear systems*, vol. 82. siam, 2003.

[34] R. J. LeVeque, *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, vol. 98. Siam, 2007.

[35] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU Press, 2012.

[36] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[37] G. H. Golub and J. M. Ortega, *Scientific computing: an introduction with parallel computing*. Elsevier, 2014.

[38] L. Adams, "A Multi-Color-SOR Method for Parallel Computation," in *1982 International Conference on Parallel Processing*, pp. 53–56, 1982.

[39] M. H. Gutknecht, "A brief introduction to krylov space methods for solving linear systems," in *Frontiers of Computational Science*, pp. 53–62, Springer, 2007.

[40] H. A. Van der Vorst, *Iterative Krylov methods for large linear systems*, vol. 13. Cambridge University Press, 2003.

[41] V. Simoncini and D. B. Szyld, "Recent computational developments in Krylov subspace methods for linear systems," *Numerical Linear Algebra with Applications*, vol. 14, no. 1, pp. 1–59, 2007.

[42] B. Smith, P. Bjorstad, and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.

[43] P. Bastian, G. Wittum, and W. Hackbusch, "Additive and multiplicative multi-grid - A comparison," *Computing*, vol. 60, no. 4, pp. 345–364, 1998.

[44] I. S. Association *et al.*, "Standard for floating-point arithmetic," *IEEE 754-2008*, 2008.

[45] O. Villa, D. Chavarria-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy, "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems," *Cray User Group, Atlanta, GA, USA*, 2009.

[46] E. Kadric, P. Gurniak, and A. DeHon, "Accurate parallel floating-point accumulation," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3224–3238, 2016.

[47] I. Foster, *Designing and building parallel programs*, vol. 78. Addison Wesley Publishing Company, 1995.

[48] "MPI: A Message-Passing Interface Standard." https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Accessed: 2015-02-21.

[49] P. B. Hansen, *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer Science & Business Media, 2013.

[50] "Home - OpenMP." `http://www.openmp.org/`. Accessed: 2015-03-12.

[51] R. Chandra, L. Dagum, D. Kohr, D. Maydan, R. Menon, and J. McDonald, *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[52] M. J. Quinn, *Parallel Programming*, vol. 526. TMH CSE, 2003.

[53] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[54] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[55] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 427–436, IEEE, 2009.

[56] L. Smith and M. Bull, "Development of mixed mode mpi/openmp applications," *Scientific Programming*, vol. 9, no. 2-3, pp. 83–98, 2001.

[57] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[58] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*, vol. 400. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[59] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde, "Parallel geometric multigrid," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, pp. 165–208, Springer, 2006.

[60] J. L. Träff and F. D. Lübbe, "Specification guideline violations by mpi_dims_create," in *Proceedings of the 22nd European MPI Users' Group Meeting*, p. 19, ACM, 2015.

[61] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[62] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi- and manycore processors," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 96, IEEE Computer Society Press, 2012.

[63] W. L. Briggs, S. F. McCormick, *et al.*, *A multigrid tutorial*, vol. 72. Siam, 2000.

[64] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, pp. 17–17, IEEE, 2006.

[65] M. Jiayin, S. Bo, W. Yongwei, and Y. Guangwen, "Overlapping communication and computation in MPI by multithreading," in *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[66] "Infiniband® Trade Association: Home." `http://www.infinibandta.org`. Accessed: 2015-05-30.

[67] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *Proceedings of the 18th Annual International Conference on Supercomputing*, pp. 298–305, ACM, 2004.

[68] W. Gropp, E. Lusk, and D. Swider, "Improving the performance of mpi derived datatypes," in *Third MPI Developers and Users Conf (MPIDC99)*, pp. 25–30, 1999.

[69] X.-H. Sun *et al.*, "Improving the performance of mpi derived datatypes by optimizing memory-access cost," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 412–419, IEEE, 2003.

[70] P. Balaji, D. Buntinas, S. Balay, B. Smith, R. Thakur, and W. Gropp, "Nonuniformly communicating noncontiguous data: A case study with petsc and mpi," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–10, IEEE, 2007.

[71] "The "vader" shared memory transport in Open MPI: Now featuring 3 flavors of zero copy !." `https://blogs.cisco.com/performance/the-vader-shared-memory-transport-in-open-mpi-now-featuring-3-flavors-of-zero-copy`. Accessed: 2018-07-23.

[72] I. Yavneh, "Why multigrid methods are so efficient," *Computing in Science & Engineering*, vol. 8, no. 6, pp. 12–22, 2006.

[73] P. Wesseling, "Introduction To Multigrid Methods.," tech. rep., DTIC Document, 1995.

[74] "MGNet Home Page." `http://www.mgnet.org`. Accessed: 2015-09-21.

[75] P. Wesseling and C. W. Oosterlee, "Geometric multigrid with applications to computational fluid dynamics," *Journal of Computational and Applied Mathematics*, vol. 128, no. 1, pp. 311–334, 2001.

[76] K. J. Brabazon, *Multigrid methods for nonlinear second order partial differential operators*. PhD thesis, University of Leeds, 2014.

[77] K. J. Brabazon, M. E. Hubbard, and P. K. Jimack, "Nonlinear multigrid methods for second order differential operators with nonlinear diffusion coefficient," *Computers & Mathematics with Applications*, vol. 68, no. 12, pp. 1619–1634, 2014.

[78] T. Gradl, C. Freundl, H. Köstler, and U. Rüde, "Scalable multigrid," in *High Performance Computing in Science and Engineering, Garching/Munich 2007*, pp. 475–483, Springer, 2009.

[79] B. Gmeiner, H. Köstler, M. Stürmer, and U. Rüde, "Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 217–240, 2014.

[80] S. Williams, D. Kalamkar, A. Singh, A. M. Deshpande, B. V. Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Implementation and Optimization of miniGMG - a Compact Geometric Multigrid Benchmark," tech. rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2012.

[81] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A survey of parallelization techniques for multigrid solvers," *Parallel Processing for Scientific Computing*, vol. 20, pp. 179–201, 2006.

[82] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Systematic reduction of data movement in algebraic multigrid solvers," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1675–1682, IEEE, 2013.

[83] "MUMPS: a parallel sparse direct solver ." `http://mumps.enseeiht.fr`. Accessed: 2015-10-10.

[84] "SuperLU: Home Page ." `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/`. Accessed: 2015-10-10.

[85] B. Gmeiner, T. Gradl, H. Köstler, and U. Rüde, "Highly parallel geometric multigrid algorithm for hierarchical hybrid grids," in *NIC Symposium*, vol. 45, pp. 323–330, 2012.

[86] P. K. Jimack, M. A. Walkley, and J. Zhang, "Scalable Parallel Multigrid Preconditioning for High Fidelity Finite Element and Finite Difference Simulations," *Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, 2015.

[87] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[88] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[89] J. Rantakokko and M. Thuné, "Parallel structured adaptive mesh refinement," in *Parallel Computing*, pp. 147–173, Springer, 2009.

[90] L. F. Diachin, R. Hornung, P. Plassmann, and A. Wissink, "Parallel adaptive mesh refinement," *Parallel Processing for Scientific Computing*, vol. 20, pp. 143–162, 2006.

[91] N. Hannoun and V. Alexiades, "Issues in adaptive mesh refinement implementation," *Electronic Journal of Differential Equations*, vol. 15, pp. 141–151, 2007.

[92] J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang, *BoxLib/Docs/UsersGuide at master BoxLib-Codes/BoxLib GitHub*, 2012. https://github.com/BoxLib-Codes/BoxLib/tree/master/Docs/UsersGuide.

[93] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured AMR calculations using the SAMRAI framework," in *Supercomputing, ACM/IEEE 2001 Conference*, pp. 22–22, IEEE, 2001.

[94] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, *et al.*, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014.

[95] B. Van Straalen, J. Shalf, T. Ligocki, N. Keen, and W.-S. Yang, "Scalability challenges for massively parallel AMR applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[96] "MAESTRO." `https://ccse.lbl.gov/Research/MAESTRO/`. Accessed: 2017-04-16.

[97] "CASTRO." `https://ccse.lbl.gov/Research/CASTRO/`. Accessed: 2017-04-10.

[98] "AMReX-Codes: Block-Structured AMR Software Framework and Applications." `https://ccse.lbl.gov/AMReX/index.html`. Accessed: 2017-10-15.

[99] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S156–S172, 2016.

[100] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pp. 655–664, ACM, 1989.

[101] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 215–228, 1999.

[102] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–57, ACM, 2006.

[103] R. Verfürth, "A posteriori error estimation and adaptive mesh-refinement techniques," *Journal of Computational and Applied Mathematics*, vol. 50, no. 1-3, pp. 67–83, 1994.

[104] M. Ainsworth and J. T. Oden, *A posteriori error estimation in finite element analysis*, vol. 37. John Wiley & Sons, 2011.

[105] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies*, pp. 213–232, Springer, 2003.

[106] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 35, 2016.

[107] P. Ghysels and W. Vanroose, "Modeling the Performance of Geometric Multigrid Stencils on Multicore Computer Architectures," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C194–C216, 2015.

[108] "Roofline Performance Model." `https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/`. Accessed: 2018-03-15.

[109] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[110] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[111] M. Sturmer, J. Treibig, and U. Rude, "Optimising a 3D multigrid algorithm for the IA-64 architecture," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, pp. 29–35, 2008.

[112] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 285–297, IEEE, 1999.

[113] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, *et al.*, "ExaStencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops*, pp. 553–564, Springer, 2014.

[114] "ARC2 - Advanced Research Computing." `http://arc.leeds.ac.uk/systems/arc2/`. Accessed: 2018-03-27.

[115] "Intel® Xeon® E5-2670." `https://ark.intel.com/products/91767/Intel-Xeon-Processor-E5-2650-v4-30M-Cache-2_20-GHz`. Accessed: 2018-03-27.

[116] "Intel® Xeon® Processor E5-2650 v4." `https://ark.intel.com/products/91767/`. Accessed: 2017-05-15.

[117] "MPICH |High-Performance Portable MPI." `https://www.mpich.org`. Accessed: 2015-05-31.

[118] "Open MPI: Version 3.0." `https://www.open-mpi.org/software/ompi/v3.0/`. Accessed: 2018-03-28.

[119] "TAU - Tuning and Analysis Utilities." `https://www.cs.uoregon.edu/research/tau/home.php`. Accessed: 2015-05-18.

[120] "PAPI." `http://icl.cs.utk.edu/papi/`. Accessed: 2015-05-18.

[121] "Scalasca." `http://www.scalasca.org/`. Accessed: 2017-08-10.

[122] "VI-HPS:: Projects:: Score-P." `http://www.vi-hps.org/projects/score-p/`. Accessed: 2018-03-30.

[123] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372, Oct 2012.

[124] S. J. Farlow, *Partial differential equations for scientists and engineers*. Courier Corporation, 2012.

[125] J. Peiró and S. Sherwin, "Finite difference, finite element and finite volume methods for partial differential equations," in *Handbook of materials modeling*, pp. 2415–2446, Springer, 2005.

[126] S. Williams, K. Datta, L. Oliker, J. Carter, J. Shalf, and K. Yelick, "Auto-tuning memory-intensive kernels for multicore," *Performance Tuning of Scientific Applications. CRC, USA*, 2010.

[127] "Open MPI: Open Source High Performance Computing." `http://www.open-mpi.org`. Accessed: 2015-05-31.

[128] D. A. Patterson, "Latency lags bandwith," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[129] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pp. 35–43, IEEE, 2007.

[130] "KNEM: High-Performance Intra-node MPI Communication." `http://knem.gforge.inria.fr`. Accessed: 2015-06-05.

[131] S. Pellegrini, T. Hoefler, and T. Fahringer, "On the effects of cpu caches on mpi point-to-point communications," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 495–503, IEEE, 2012.

[132] "Documentation for Intel® C and C++ Compilers |Intel® Software." `https://software.intel.com/en-us/c-compilers/ipsxe-support/documentation`. Accessed: 2018-02-07.

[133] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pp. 1–6, IEEE, 2010.

[134] R. C. OReilly and J. M. Beck, "A family of large-stencil discrete Laplacian approximations in three dimensions," *International Journal for Numerical Methods in Engineering*, pp. 1–16, 2006.

[135] A. Bourached, "Blocking versus non-blocking halo exchange," *arXiv preprint arXiv:1709.06175*, 2017.

[136] H. P. Langtangen, "Solving nonlinear ODE and PDE problems," 2015. unpublished.

[137] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[138] G. Saxena, P. K. Jimack, and M. A. Walkley, "A Cache-aware Approach to Domain Decomposition for Stencil-based Codes," in *International Conference on High Performance Computing and Simulation (HPCS 2016)*, pp. 875–885, 2016.

[139] "CCSE Research: Low Mach Number Combustion." `https://ccse.lbl.gov/Research/Combustion/`. Accessed: 2017-04-11.

[140] "VisIt." `https://wci.llnl.gov/simulation/computer-codes/visit`. Accessed: 2016-05-15.

[141] G. Saxena, P. K. Jimack, and M. A. Walkley, "A quasi-cache-aware model for optimal domain partitioning in parallel geometric multigrid," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4328, 2018.

[142] G. Saxena, P. K. Jimack, and M. A. Walkley, "A Cache-Aware Approach to Adaptive Mesh Refinement in Parallel Stencil-Based Solvers," in *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference*, pp. 364–371, December 2017.

[143] P. Bollada, C. E. Goodyer, P. K. Jimack, A. M. Mullis, and F. Yang, "Three dimensional thermal-solute phase field simulation of binary alloy solidification," *Journal of Computational Physics*, vol. 287, pp. 130–150, 2015.

[144] F. Hülsemann, B. Bergen, and U. Rüde, "Hierarchical hybrid grids as basis for parallel numerical solution of PDE," in *Euro-Par 2003 Parallel Processing*, pp. 840–843, Springer, 2003.

[145] P. N. Brown, R. D. Falgout, and J. E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1823–1834, 2000.

[146] C. Kaltenecker, "Comparison of analytical and empirical performance models: A case study on multigrid systems," Master's thesis, University of Passau, 2016. Master's Thesis.

[147] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP," in *Parallel Processing (ICPP), 2012 41st International Conference on*, pp. 128–137, IEEE, 2012.

[148] G. Romanazzi and P. K. Jimack, "Parallel performance prediction for multigrid codes on distributed memory architectures," in *International Conference on High Performance Computing and Communications*, pp. 647–658, Springer, 2007.

[149] "Intel® 64 and IA-32 Architectures Optimization Reference Manual." `https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`. Accessed: 2016-12-17.

[150] "FAQ: General run-time tuning." `https://www.open-mpi.org/faq/`. Accessed: 2016-11-24.

[151] R. de la Cruz and M. Araya-Polo, "Towards a multi-level cache performance model for 3D stencil computation," *Procedia Computer Science*, vol. 4, pp. 2146–2155, 2011.

[152] "What is an MPI "eager limit" ?." `https://blogs.cisco.com/performance/what-is-an-mpi-eager-limit`. Accessed: 2018-04-11.

[153] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High performance RDMA protocols in HPC," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pp. 76–85, Springer, 2006.