Phase-based Adaptive Dynamic Load Balancing for Parallel Tree Computation

by

Fazilah Haron

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

School of Computer Studies
The University of Leeds

August 1998

The candidate confirms that the work submitted is her own and the appropriate credit has been given where reference has been made to the work of others.

Abstract

Dynamic load balancing (DLB) is a technique for the parallel implementation of problems which generate unpredictable workloads by migrating work units to lightly loaded processors based on run-time workload measurement. *Adaptive* DLB is a refinement where aspects of the load balancing system itself are modified in the light of measured workloads.

This thesis investigates *phase-based* adaptive DLB, a version of adaptive DLB in which a parallel computation moves through different load balancing phases identified on the basis of run-time workloads. The idea is explored through a case study of parallel tree computation, in which three distinct phases with intervening transitions are identified. Two major variants of phase-based adaptivity are distinguished. In *parametric adaptivity*, parameters of the DLB algorithm are adapted between phases; in *algorithmic adaptivity*, different DLB algorithms are utilised in each phase. These concepts are investigated quantitatively through a simulator for parametric adaptivity and discussed in detail for algorithmic adaptivity.

The simulator permits a range of processor topologies, parameterises the performance of the underlying network, includes two different network performance models, and allows a wide range of simulated tree-structured workloads, parameterised by depth, fan-out, node granularity and imbalance. It was extensively validated in relation to the performance of two DLB algorithms on a 512-processor Cray T3D.

The simulator was used to evaluate the benefit of parametric phase-based adaptivity. Preliminary experiments with non-adaptive algorithms revealed performance to be sensitive to the interval between load-balancing invocations, so this parameter was prioritised for subsequent adaptivity experiments. A performance metric called Improvement Through Adaptivity (ITA) was discussed. Two DLB algorithms were used as test cases; the well-established Generalised Dimension Exchange Method and a novel Loadserver algorithm, whose implementation is described in the thesis.

Results were obtained for all combination of the transitions, and the relationships between ITA and various parameters (processor sizes, node granularity, tree imbalance and network performance) were established. Similar relationships were observed for both algorithms, though with some differences in detail. Positive values of ITA were obtained with both algorithms, for at least one transition combination, over a range of all the parameters. Thus, the potential benefits of phase-based parametric adaptivity are confirmed, justifying future work in implementing this technique.

To my parents,

Rokiah Abdul Rashid and Haron Hamid,

for their unconditional love and care throughout my life.

Acknowledgments

I would like to deeply thank all of those who helped me with this thesis.

My first supervisor, Dr John R Davy for his invaluable advice, guidance and encouragements during the course of this work, especially during the last nine months. My second supervisor, Prof Peter M Dew, also offered valuable comments at the early stage of this work.

The Parallel Group members, Dr Jonathan Nash, Dr Wissal Essah and Dr Don Goodeve have also provided help in various ways. Their help and support are gratefully acknowledged. Thanks are also due to my postgraduate friends for their help and assistance throughout my study. I would also like to thank Mike Parsons, a final year project student, who assisted the initial implementation of the load balancing algorithms.

I am grateful to my friends and housemates for their friendship, help and encouragement. My special friend, in particular, who has always been a great support and encouragement during the last stage of my study.

Finally, my special thanks and appreciations go to my parents, for their constant support and advice, encouragement and prayers, to whom this thesis is dedicated. Thanks are also due to my brother and sisters, brother-in-law, my niece and nephews who have always been wonderful to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Dynamic Load Balancing

Parallel processing is believed by many to be the wave of the future in computing [39, 23]. Fundamental physical limitations on processing speeds will eventually force high-performance computations to be targeted principally at the exploitation of parallelism. Just as the fastest cycle times are approaching their fundamental barriers, new generations of parallel machines are emerging. Examples of such machines include Cray T3E, IBM SP2, Intel Paragon, Convex machines, Ncubes and Meiko CS2 [77].

Achieving good performance from these machines is a non-trivial task. Factors such as load imbalance, inherent serial sections, contention for shared resources, synchronisation and communications may inhibit good performance. These issues are central to the development of many parallel applications. In the case of load imbalance, research has led to many load balancing techniques to optimise the performance of parallel applications.

The issue of load balancing exists not only in parallel systems, but also in distributed environments, with one common objective – to improve the performance. However, the nature of the performance objective differs. In a distributed system the objective is usually to reduce the average response time of a mix of independently submitted jobs, while in a parallel system the aim is usually to minimise the total execution time of a single program.

For some applications it is possible to make *a priori* estimates of the work distribution; for example, the standard dense matrix multiplication. The assignment of tasks to processors can be done before program execution begins. Such an off-line *a priori* determination is called *static* load balancing [60, 69]. By contrast, a strategy which attempts to balance work during an execution is referred to as *dynamic* load balancing (DLB).

Such techniques are used when no prior estimate of load distribution is possible, so static methods are inappropriate. It is only during actual program execution that it becomes apparent how much work is being assigned to individual processors. This is due to the dynamic or non-uniform computational nature of the problem.

The key feature of dynamic *load balancing* is that units of work are migrated from heavily loaded processors to lightly loaded processors in order to achieve a well balanced load throughout the system. Some dynamic load balancing algorithms only aim to ensure no processor remain idle when there are useful work to be performed in the system. This objective is normally referred as *load sharing*. The decisions as to when and where to migrate tasks are typically based on run-time measurements of the system load. DLB algorithms improve performance by exploiting short-term fluctuation in this system state (or load). Since they must collect, store, and analyse state information, DLB algorithms incur more overhead than their static counterparts, but this overhead is often well spent.

The popularity of DLB is increasing with the continuous demands for better processing speed. The last decade has been one of the most exciting period for DLB in parallel computing. Extensive research has been done covering a wide range of topics from theoretical background [4, 11] to the practical state-of-the-art load balancing techniques [9, 32, 33, 37, 38, 74, 80, 82, 85, 88, 89]. DLB has become extremely important in many disciplines. Some typical areas include combinatorial search [44], optimisation problems [17], climate modeling [80], finite element methods [76], N-body problems [72], to name just a few.

1.2   Adaptivity in Dynamic Load Balancing

Dynamic load balancing can be characterised as *adaptive* or *non-adaptive* [1] (see Figure 1.1). An adaptive dynamic load balancing algorithm automatically responds to the system state in order to be operating at (or close to) its optimal level [7]. An excellent description of adaptive DLB system was provided by Shiva *et al.* in [70];

> "Adaptive load-distributing algorithms are a special class of dynamic algorithms. They adapt their activities by dynamically changing their parameters, or even policies, to suit the changing system state. For example, if some load distributing policy performs better than others under certain conditions, a

---

[1]Some authors used the terms *adaptive load balancing* and *dynamic load balancing* interchangeably, whereas the first is best seen as a special class of the latter.
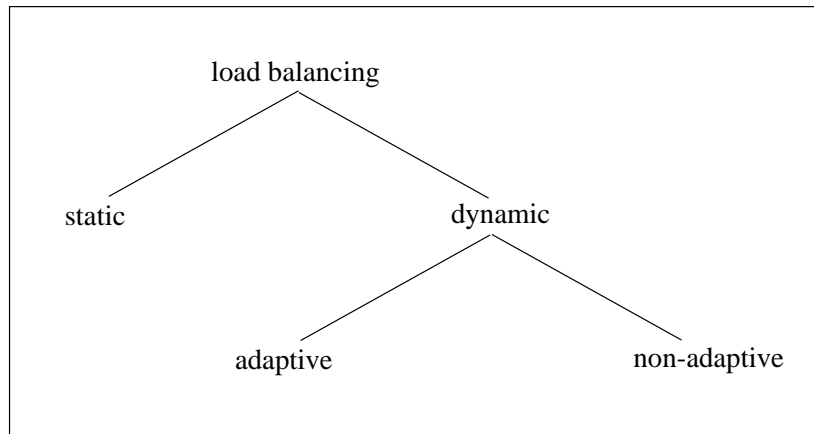
Figure 1.1: The characteristics of load balancing algorithms.

> simple adaptive algorithm might choose between these policies based on ob-
> servations of the system state. Even when the system is uniformly so heavily
> loaded that no performance advantage can be gained by transferring tasks,
> a non-adaptive dynamic algorithm might continue operating (and incurring
> overhead). To avoid overloading such a system, an adaptive algorithm might
> instead curtail its load balancing activity when it observes this condition."

Most research in adaptive DLB is in the context of distributed systems [3, 13, 20, 41, 64, 68]. A seminal work on adaptive DLB in distributed system can be found in Krueger [41]. In contrast, sporadic work have been carried out on adaptive DLB in parallel computing, examples include [9, 80, 89].

Most of the adaptive work in the literature alters the DLB parameters to improve the performance. These parameters include the workload threshold [13, 68, 89], the interval in between DLB invocations [80, 89] and the migration factor [9], which determines the amount of load migrated between processors. There is little work which exploits adaptivity by changing the DLB algorithms. One example is by Ramamritham and Zhao [64] which switches between two DLB algorithms, bidding and focus-addressing, in a distributed real-time system. Their work include both approaches to adaptivity, namely adapting parameters and algorithms.

1.3   Phases in Dynamic Load Balancing

Most of the studies on DLB in distributed systems attempt to balance the workload at the system level where the load balancing involves the movement of generic processes without

any knowledge of the application they represent. The workloads are typically processes (recognised by the operating system) which can be characterised as having different sizes and varying arrival rates, but where nothing is known by the load balancing system about the computational tasks represented by these processes. Hence, it is not feasible to use any knowledge of application characteristics in order to improve the performance of load balancing.

The scenario in the parallel world is rather different. Most DLB systems are built directly into the application [17, 44, 76, 72, 80]. Migration usually involves tasks or unit of data specific to that particular application. Since the DLB is integrated within the application, there is a potential to optimise for that application.

This thesis will consider the potential for one particular way in which DLB may be improved by taking into account particular characteristics of the application. Specifically we will consider the possibility that the execution of an application may proceed in distinct *phases*, detectable at run-time, where different load balancing approaches are applicable in each phase. This form of adaptivity will be called *phase-based adaptive dynamic load balancing*. Clearly, the approach requires a prior knowledge of workload characteristics in order for phases to be identifiable.

This idea will be pursued by consideration of a class of parallel application which can be characterised as *tree computation*, because the execution generates a tree of tasks. This class of computation includes divide-and-conquer and branch-and-bound algorithms. Parallel tree computation commonly make use of DLB because the shape of the tree of tasks is frequently not predictable and may be highly imbalanced. However, the knowledge that the task graph will take the form of a tree may be exploited to identify different phases in the computation. In particular, this thesis will explore the use of three main phases: an initial phase in which the expansion of the task tree gradually leads to all processors being utilised, a central phase in which DLB maintain all processors fully utilised, and a final phase in which the machine gradually empties as the number of remaining tasks reduces.

Some previous work distinguishing the initial phase of tree computation has been described in [62, 67]. Unfortunately, those two examples are limited to a specific applications and do not consider more general possibilities for phase-based load balancing, and indeed do not use the notion of phases explicitly. Other work discussing load balancing or scheduling in phases more explicitly [24, 79] involves situations where the phases are distinct and repeated parts of the computation, rather than representing different workload characteristics developing dynamically within a single computation.

1.4 Research Objectives

The overall goal of the research in this thesis is to explore the concept of phase-based adaptivity in DLB as explained in the previous section.

More specifically, the objectives of the study are:

- To develop a conceptual framework for adaptive dynamic load balancing, including phase-based adaptivity.

- To investigate and evaluate the potential benefits of phase-based adaptivity using simulation.

It is hoped that the study will provide a more systematic view of adaptive DLB in parallel computing, reveal some insights on the new approach, and establish some relationship between the performance benefit and the selected application or machine characteristics.

1.5 Thesis Structure

The thesis is divided into six chapters, including the introduction. Chapter 2 presents the background studies of the entire work; the tree computation, dynamic load balancing algorithms in general and the specific algorithms used in this study. It also presents the implementation of a new DLB approach. The programming environment, that is the message passing library and the platform used, are also described.

Chapter 3 proposes the conceptual framework of phase-based adaptivity in dynamic load balancing in the context of parallel tree computation. It introduces the notion of phase-based adaptivity and the basic concepts involved. It then proceeds to explain the mechanisms to adapt the DLB algorithm. The chapter continues by positioning the contribution of this thesis in relation to the related work on phases and adaptivity. A simple formalism for adaptivity performance and some definition of parameters are also introduced followed by the reasons for adopting simulation to investigate the idea at the end of the chapter.

Chapter 4 is concerned with the design and implementation of the simulator. It first describes the general parallel execution model and the components of the simulator. This is then followed by a discussion on the performance model used for the computation and the load balancing operations, the calibration and validation process.

Chapter 5 starts by presenting the results of three preliminary experiments, whose objectives are to verify the assumption made on the workload pattern, select the type of

the traversal method and experiment the sensitivity of the DLB interval. It then proceeds by defining the metric used to assess the improvement gained from phase-based adaptivity and the experimental plans for parametric and algorithmic techniques. The results of the parametric approach for two applications and one machine parameters are then analysed.

Chapter 6 presents the summary and evaluation of the work, and highlights the insights gained from the whole study. Possible future plans are also detailed in this chapter.

Chapter 2

Background

This chapter serves as a background knowledge to the research presented. It discusses the dynamic load balancing algorithms of interest, the application concerned, and the parallel environment of the experimental work.

The chapter starts by introducing the parallel environment in which the experimental work was carried out (in Section 2.1). It then proceeds to discussing the nature of the application used, that is tree-structured computation, in Section 2.2. Section 2.3 explains in detail two DLB algorithms used in the next two chapters; Diffusion and Dimension Exchange Method. Two other algorithms which are related to the work are also described. Section 2.4 discusses the components of a DLB algorithm. Finally, Section 2.5 discusses the implementation of an alternative hybrid algorithm, the Loadserver.

2.1 Characteristics of the Parallel Environment

The experimental platform used in the study consists of the Cray T3D supercomputer, at Edinburgh Parallel Computing Centre (EPCC), and the Message Passing Interface (MPI) standard.

The Cray T3D [21] is composed of 512 DEC Alpha 21064 processors each rated at 150 MHz with 64 Mbyte of memory, giving an aggregate memory of 32 Gbyte and providing peak performance of 76.8 Gflop/s. The nodes (each node comprises two processing elements) are arranged in a 3-d torus. All arithmetic operations, both integer and floating points, are performed using 64-bit arithmetic.

MPI [78] is a portable, public-domain library standard message passing, which adopts most if not all, common practices from existing communication libraries. A detailed discussion on the implementation of MPI on the T3D is available in [6]. MPI defines func-

tions for sending messages from one process to another (point-to-point communication), for communication operations that involve groups of processes (collective communication, such as reduction, scatter, gather etc.), and for obtaining information about the environment in which a program executes (enquiry functions). The *communicator* construct combines a group of processes and a unique tag space can be used to ensure that communications associated with different parts of a program are not confused. The rich support of a collective operations is one of the key feature of MPI. It makes coding easier and less error-prone. Another advantage is performance – using a single collective communication is faster than using a sequence of point-to-point operations.

For convenience in programming, MPI also provides virtual topology; a high-level method for managing process groups without dealing with them directly. One can conceptualise processes in an application-oriented topology using general graphs and grids. Virtual topology highlights the main communication patterns in a communicator by a 'connection', but at the same time allowing any process within a communicator to communicate with each other.

Facility such as virtual topology is not supported in PVM (Parallel Virtual Machine), the main alternative public domain standard message passing library [73]. Relatively minimal support of collective operations is provided on PVM. For these reasons MPI was preferred over PVM for this work. An overview of message passing library, which includes MPI and PVM, and other portable libraries, is available in [52].

2.2   Parallel Tree Computation

Tree computation starts with a single node, the root, which represents the complete problem to be solved. Nodes are dynamically generated and consumed, through expansion and solving operations. Thus the computation can be viewed as a dynamically growing (and contracting) tree of nodes. Each node forms an independent unit of work, hence multiple nodes can be executed in parallel. Since the shape of the tree is typically irregular, unbalanced and unpredictable, static load balancing is not feasible and dynamic load balancing techniques are commonly employed. Nodes are stored in task queues at each processor and are migrated between processors to improve load balance.

There are many applications which exhibit tree characteristics. Those which use divide-and-conquer and branch-and-bound algorithms are typical examples of tree computation.

2.2.1 Divide-and-conquer

Divide-and-conquer (D&C) algorithms recursively partition a problem into smaller sub-problems until the sub-problems are small enough to be solved directly. The solutions of the sub-problems are combined into a solution for the original problem.

D&C can be used to solve many problems, including mergesort and the well-known fractal image – the Mandelbrot set [61]. The Mandelbrot set is defined as follows. A series of points on the complex plane can be computed using a function $f_c(z) = z^2 + c$, where c and z are a complex number and complex variable, respectively. Repeated application of the function determines whether each point is a member of the set. A point is said to be in the set if it remains bounded, otherwise it is not, that is when the value gets farther away from 0. The image of the set can be made by plotting the points on the screen.

The computations at all points are independent, so the problem is highly parallelisable. D&C can be exploited by recursively dividing the plane into stripes or quadrants. Since points near the middle of the plane need more iterations there is a need for dynamic load balancing.

Various authors (e.g. [16, 46, 53]) show that D&C algorithms can be defined by four functions: *divide*() splits a problem into subproblems, *leaf*() determines whether a problem is small enough to be solved directly, *solve*() computes the results of a 'small' problem, and *combine*() combines the results of sub-problems. Once these functions are defined, it is possible to execute a complete D&C problem in parallel by means of a generic kernel which controls the subdivision and combining, and organises load balancing, communication and synchronisation as required. This implies that load balancing system can be developed for D&C which are independent of any one specific application, an important pre-requisite for the work in this thesis.

2.2.2 Branch-and-bound

Branch-and-bound (B&B) algorithms represent an important technique in solving combinatorial search problems [26]. The basic scheme is to reduce the problem search space by dynamically pruning unsearched areas which can not yield better results than solutions already found. Branching is performed by recursively partitioning the problem into sub-problems. A *lower bound* is computed for each subproblem to determine whether or not further exploration of the subproblem is worthwhile. In other word, B&B is an exhaustive tree search over the solution space using methods aimed at reducing the size of the search tree.

Searching algorithms that use B&B techniques can be characterised by the *search heuristics* used in searching for a solution. The search heuristics determine determines the order in which the algorithm conducts the search in the tree. Popular heuristics include *best-first* and *depth-first* searching. Best-first B&B requires large amounts of storage of intermediate subproblems for non-trivial problem instances, which may either deteriorate its performance by use of secondary storage or prohibit its use altogether. Depth-first B&B has a modest storage requirements, but may search a larger part of the search tree, and never searches a smaller part than best-first search.

A typical B&B problem is the Traveling Salesman Problem (TSP). Here a salesman must visit n cities, returning to the starting point, while minimising the total cost of the trip. There are several solutions to the problem. One of the best known is due to Little *et al.* [49]. Another example of B&B problem is the Knapsack Problem [51].

Just as with D&C, generic interfaces have been devised for B&B problems. Corresponding kernels then organise the parallel implementation details [42, 74].

2.2.3   A Comparison Between Parallel D&C and B&B

Both D&C and B&B generate a tree of independent tasks which may be executed in parallel. Kernels for these applications control the traversal of this tree, executing the interface functions at nodes of the tree. In both cases the structure of the tree is typically irregular and data-dependent, hence it is impossible to determine *a priori* how the tree node computations may be scheduled amongst the processors. For this reason dynamic load-balancing techniques are used, based on run-time evaluation of the loads on different processors. In this respect the load balancing problems of D&C and B&B can be seen as identical.

There are, however, some important differences in detail.

- D&C requires two phases, decomposing problems and composing results, so the tree structure resulting from the computation must be maintained. This is not necessary for B&B.

- B&B must maintain global data to be shared among all tree nodes (i.e. the lower bound [26]), whereas there is no such requirement for D&C.

- In D&C the task tree is normally traversed in depth-first order so as to minimise memory requirements [53], whereas alternative orders, such as best-first, can be applied to B&B.

- Termination of the distributed computation is easily detected for D&C when the result of the root problem is obtained, whereas B&B requires more complex distributed termination detection.

Despite these differences, it is possible to use essentially the same load balancing techniques for these applications.

2.3 Dynamic Load Balancing Algorithms

We are interested in DLB schemes which seek to minimise total execution time of a single application on massively parallel systems. Such parallel systems often allow sharing of data through explicit message passing (e.g. MPI and PVM systems).

Numerous load balancing schemes with different characteristics have been proposed in the literature [9, 11, 13, 40, 48, 50, 56, 71, 86, 89, 85, 82, 81, 84], of which two have been studied extensively; Diffusion and Dimension Exchange Method. These two algorithms will be described in detail in this section since they are used in the subsequent chapters. Two other algorithms will also be described − a random algorithm and an adaptive DLB method. Between them these four algorithms introduce many of the key component of DLB and are representatives of the much wider range of algorithms in the literature.

2.3.1 Diffusion

In *Diffusion* [11, 82, 85] work is moved periodically from heavily loaded processors to more lightly-loaded ones. Since processors use load information only from neighbours and tasks are migrated only between neighbours, both the information domain and migration space are local. Each processor performs load balancing operations independently, leading to a diffusion of work throughout the machine.

Several variants of diffusion have been implemented; one major distinction is between *sender-initiated* diffusion (SID), in which a migration is initiated by a heavily-loaded processor and *receiver-initiated* diffusion (RID), in which a lightly-loaded processor initiates the load balancing. RID is reported to outperform SID and is anticipated to be suitable for large parallel systems [82].

We describe a detailed algorithm for RID as it appears in [82] − the same principle applies to SID. In RID, the balancing process is initiated by a processor, i, whose load drops below a threshold L_{LOW}. The average load of processor i and its neighbours, $\overline{L}_i$, is then calculated. If processor i's load, L_i, is below the average load by more than

another threshold, $L_{threshold}$, the execution proceeds to the next load balancing phase, i.e. calculating the neighbour's excess load (if any). Each neighbour j is assigned an excess weight w_j as follows:

$$w_j = \begin{cases} L_j - \overline{L}_i & \text{if } L_j > \overline{L}_i \ , \\ 0 & \text{otherwise.} \end{cases}$$

The total excess weight for all neighbours, W_i, is also computed,

$$W_i = \sum_{j=1}^{J} w_j$$

The amount of load requested by processor i from neighbour j, δ_j, is:

$$\delta_j = (\overline{L}_i - L_i)\frac{w_j}{W_i}$$

The load requests are sent to appropriate neighbours. A neighbour will transfer a maximum of half of the amount of the current load for each request. This is to avoid instability and reduce the effect of information aging.

All processors inform their near-neighbours of their load levels and update this information throughout execution. The accuracy of the load information highly depends on the frequency of the load update. If the frequency is high the accuracy of the load estimation increases, so does the overhead. The reverse is true for low frequency. Clearly, there exist a tradeoff between the quality of load information and the overhead in achieving it.

Since diffusion make load balancing decisions asynchronously, in overlapping balancing domains, the balancing operation may suffer from an inaccurate load information, i.e. information aging. A processor may end up transferring an excessive or insufficient number of tasks from one processor to another. This situation is called processor thrashing which will lead to system instability. An adaptive RID strategy which automatically adjust the load diffusion factor according to the system load fluctuation may help rectify the problem [9].

Diffusion has several benefits: it is entirely distributed, thus avoiding bottlenecks; it requires communication only between neighbours; and it has provably-good long-term balancing properties [11]. On the other hand, movement of work is slow; for instance an idle processor with no heavily-loaded neighbours can not immediately obtain work, even if there are heavily-loaded processors elsewhere in the system.
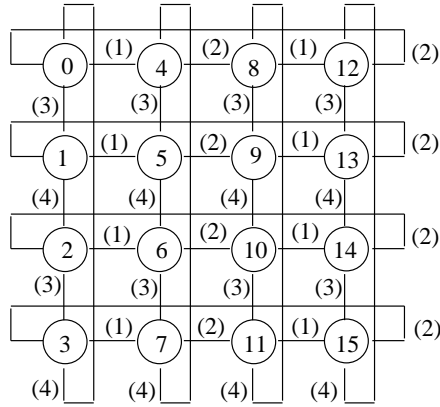
Figure 2.1: Edge colouring for a 2-d torus.

2.3.2 Dimension Exchange Method

Dimension Exchange Method (DEM) was initially proposed as a load balancing algorithm for the hypercube structure [11]. In DEM a single load balancing operation consists of $\log p$ pairwise balancing steps, where p is the number of processors in the machine. Each balancing step corresponds to each of the $\log p$ dimensions. All node pairs of the same dimension exchange their load information and average out the number of tasks. The whole system is balanced after a single load balancing iteration.

Hosseini *et al.* generalised the method to non-hypercube topology based on edge colouring of undirected graphs [34]. With edge colouring, the edges of a given graph are coloured with some minimum number of colours such that no two adjoining edges are of the same colour. A *dimension* is then defined to be a collection of all edges of the same colour. Figure 2.1 shows an example of a colour graph of a 2-d torus. The maximum colour is four. Hence, all processors complete one load balancing operation after four consecutive exchange steps.

For a non-hypercube topology, DEM can no longer yield a uniform workload distribution in a single iteration, but will eventually converge to a uniform distribution [34]. The number of iterations required is linearly proportional to the total processors for the chain and to the dimension order k for k-ary n-cubes topology, where k is the number of processors along the dimension. To rectify the problem Wu and Shu [84] proposed a direct method which balance the load in one iteration by allowing each node to obtain the global system state through *sum* reduction operation.

Xu & Lau parameterised DEM according to the amount of workload to be migrated between processor pairs [88]. They refer to the parameter as the workload exchange

```
for colour := 1 to max_colour do
    node-pair, i and j, exchange load information l_i and l_j
    if (l_i - l_j) > 1 then
        send ⌊ (λl_i - λl_j) ⌋ tasks to node j
    endif
    if (l_j - l_i) > 1 then
        send ⌊ (λl_j - λl_i) ⌋ tasks to node i
    endif
endfor
```

Figure 2.2: Dimension exchange algorithm.

parameter, λ. Optimal λ (or λ_{opt}) leads to the fastest convergence of a balancing process. Equal splitting of the total workload (i.e. $\lambda_{opt} = 1/2$) between processor pairs only yields optimum results for the hypercube structure. For other topologies and network sizes the value of λ_{opt} varies. For a mesh, λ_{opt} is $1/(1 + sin(\pi/k))$ and for torus is $1/(1 + sin(2\pi/k))$ where k refers to the maximum dimension length of the topology.

Figure 2.2 shows (with some notational changes) an integer version of DEM algorithm for an arbitrary topology, as given in [84]. For integer workloads absolute load balance can not be achieved; a pair of processors is regarded as balanced if the difference is no larger than one, a very small difference in the case of real workloads.

Although the theoretical results showed the supremacy of GDEM[1] over Diffusion, GDEM is anticipated not to scale well for very large parallel system [82]. This is because the theoretical studies do not take into account the global synchronisation overhead which incurs every time the load balancing is invoked.

2.3.3 Other Example DLB Algorithms

Random Algorithm

Random algorithms studied by several authors are quite simple and effective. There are two variants of random algorithms; a newly created task is sent to a randomly chosen processor in the system [8, 71] or a task is migrated only when the workload reaches above a certain 'heavy' limit as in [20]. The major advantages of this strategy are its simplicity and topology independence. No (or minimum) load information needs to be

---

[1]Since this thesis adopts the generalised DEM algorithm - in terms of the topology and the exchange parameter - we use GDEM as our abbreviation for the rest of this thesis.

maintained, nor is any load information sent to any processors. Results have shown that random algorithms have a respectable performance [71]. However the lack of locality may cause a performance degradation due to large overhead and communication traffic since the probability of the task being transferred to remote processors is high. This is especially true for a random algorithm which migrates task immediately upon creation.

An Adaptive DLB Method

The adaptive DLB method by Xu and Hwang [89] illustrates an adaptive system which adapts two important DLB parameters, namely the frequency of load information update and threshold value. The latter determines whether a processor is heavily loaded.

The system uses a dedicated processor which acts as a load information collector that periodically collects and broadcasts load information. The frequency between two successive updates is varied according to the current system load variation. The duration between two updates is referred as the *time window*.

The transfer and location decisions are completely distributed and are based on two heuristics; a processor with minimum load or the one which is the least migrated to so far is the candidate for receiver. The adaptive threshold value and the migration domain are determined based on a local or global range policy. A local policy dictates migration to immediate neighbours only and the threshold value is the average load among neighbours including itself. Global policy consider the whole system load.

Although the load information collector periodically updates the load information to all processors, the load balancing decision may not be based on accurate load information. Any load changes that occurs within a time window are only known to the processors that are involved, but not the rest. For this reason the balancing decision may not be correct, let alone optimal.

Melab *et al.* [55, 56] extend the method by adding the local delay apart from having the global delay. These values determines when the processor should send its load information and when the information collector should broadcast the system state. This is to finely detect the changes in the local load which leads to the system load fluctuation.

2.4   Components of a Dynamic Load Balancing Algorithm

Typically a load balancing algorithm consist of three main components [3]:

(i) Information policy.

The *information policy* specifies what information about the states of other processors in the system is to be collected, when the information is to be collected, from where and how it is to be collected.

The *information space* defines the domain from where the load information is to be collected. A *global* information space dictates the maintenance of the whole system state. One example algorithm is the adaptive DLB systems proposed by Xu and Hwang above. Many algorithms gather load information from directly connected neighbours (e.g. Diffusion and GDEM). These algorithms are said to have a *local* information space. There exist algorithm which does not use any load information at all; an example includes Random algorithm. The frequency of gathering the load information (also referred as update frequency) can be periodic (e.g. Diffusion) or adaptive (as in the adaptive DLB approach).

Most load balancing require evaluation of load on a per-processor basis. The measure of load can be some weighted tasks or just a simple queue length. A study in [47] concluded that a simple metric (e.g. queue length) is as effective as the more complex metric such as the CPU or memory usage.

(ii) Transfer policy.

The *transfer policy* determines the condition under which migration should take place and whether the processor is a sender or a receiver.

Many algorithms use a threshold-based transfer policy [43, 20]. These algorithms use the knowledge of the individual load of the processor in deciding whether transfer is necessary. No exchange of state information is required. A task will be transferred if the load is greater than the threshold value.

RID and the local adaptive DLB method (by Xu and Hwang above) use the average load between neighbours to initiate the migration. GDEM algorithm uses a relative transfer policy – migration only occurs when the load difference between a pair of processor is greater than one (assuming the workload is integer).

(iii) Location or placement policy.

The *location policy* determines the 'transfer partner' to which a task should be assigned. The region of the transfer could be just the neighbouring processors, or a cluster of processors or the entire system. The first two are typically referred as *local migration space* while the third as having *global migration space*.

The rule used to select a partner ranges from a simple random probability to a designated processor. Random algorithms transfer the task to a randomly chosen processor. If the destination processor is itself the task will be processed locally. GDEM, on the other hand, transfers the tasks to the designated processor pair in each dimension. In RID the underloaded processor receives tasks from any of the directly connected neighbours whose load are in excess.

The load balancing activities may be *centralised* in a single processor or *distributed* among all the processing elements that participate in the load balancing process. In a centralised approach, a dedicated node gathers global information about the state of the system and assigns tasks to individual nodes. In other word, the three mentioned policies are executed by a single node. With a fully distributed approach each node executes its own scheduling policy by exchanging state information with other nodes; all the three policies are fully distributed.

2.5   A Hybrid Approach to DLB

Most algorithms to date follow a purely centralised or purely distributed approach. A common example of the centralised approach is a master-slave model [59]. A dedicated process, the master, distributes the workload to the slave processes which carry out the actual computation. When idle, a slave requests more work from the master. Such a system will have limited scalability because of bottlenecks at the master. Any centralised approach will tend to suffer from similar problems.

For this reason most dynamic load balancing follows a purely distributed model. A common technique in a purely distributed approaches is diffusion. The problem with this approach is it may lead to sub-optimal load migration decisions. A heavily-loaded process cannot offload to a lightly-loaded non-local process, hence a process may remain idle for a substantial time, even when there are surplus tasks in the system. This effect is particularly significant for tree-structured computation because in both cases the computation starts

from a single task. Thus, global balancing may be relatively slower to achieve compared to the centralised approach.

Since both centralised and distributed approaches have limitations in terms of performance, recent researches have begun to consider hybrid approaches, aiming to combine the scalability of distributed models with the better load information of centralised models [2, 83]. Ahmad and Ghafoor [2] presented a semi-distributed approach which partitions the network into independent symmetric regions centred at some control points. The central points are schedulers which optimally schedule tasks within their spheres. The schedulers consider task migration to other spheres only if the load of the most lightly loaded node in its sphere is greater than the threshold limit, *threshold-1*. Tasks are migrated based on a significant difference (i.e. *threshold-2*) between the accumulative load of the local sphere and that of remote sphere.

Wu and Kung [83] proposed an algorithm in which individual processes execute task subtrees locally in depth-first fashion, but maintain a *global pool* of tasks from which a process out of work can obtain a further task. Under certain simplifying assumptions they show that their algorithm is optimal in terms of the number of tasks which need to be migrated, but the maintenance of the global pool is costly and difficult to implement on a distributed memory machine. However, a subsequent simulation of a related algorithm by Nash *et al.* on a scalable shared memory platform showed good scalability [57].

Wu and Kung's work shows that the distributed scheduling of tasks in combination with a limited centralised component has the potential for scalable performance, though the specific algorithm is unrealistic for the current generation of distributed memory computers because of the complexity of the centralised function.

The remainder of this section describes a novel hybrid algorithm, the Loadserver, which also combines distributed scheduling with a centralised component, but in a very different way; the central component is extremely simple and lightweight. The aim is to make it feasible for implementation on the current generation of distributed memory machines while benefiting from the better load balancing made possible by global load information. This is consistent with the conclusion of Eager *et al.* [20] that very simple load balancing policies using small amount of information are the most promising.

2.5.1   Loadserver Algorithm

The Loadserver (LDSV) algorithm was originally proposed in [14] where an initial implementation was described. The distributed characteristic of the LDSV algorithm is

captured through its autonomous local decision making in scheduling the workload. Each process maintains its own task queue, from which it selects tasks to execute and where newly divided tasks are placed. Migration of tasks between queues is facilitated by a centralised LIS which maintains information about the lightly loaded processors. LDSV is an example of an asynchronous load-balancing algorithm where each processor performs balancing operations discretely based on their own local workload distributions and invocation policies. Note that in Chapter 4 we describe the implementation of the synchronous version of LDSV for the purpose of experimenting adaptivity.
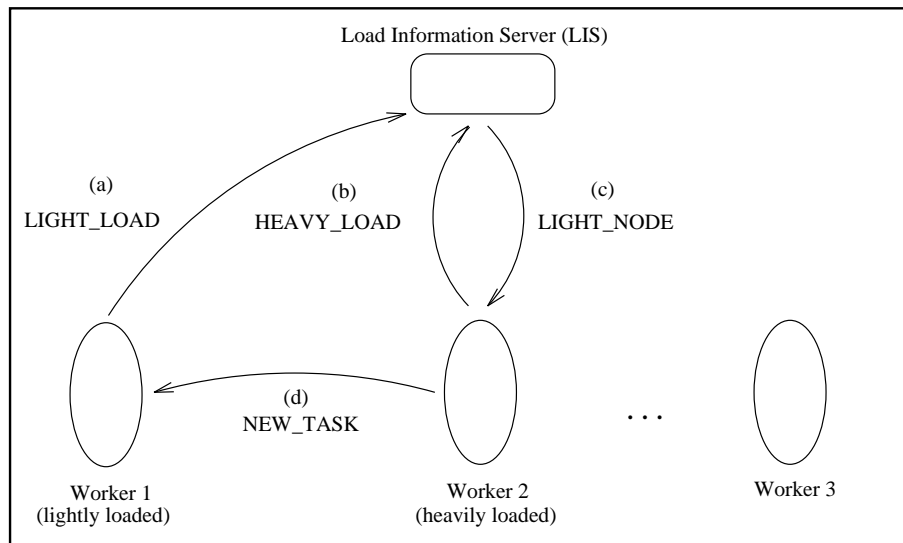
Figure 2.3: The principles of the Loadserver algorithm.

In LDSV one process maintains the LIS while all other processes run a *worker* process which executes the application kernel. The principle of the algorithm is shown in Figure 2.3. When a worker's load falls below a specified 'light' threshold it sends a LIGHT_LOAD message (a) to LIS, which enqueues the sender's id. After executing a task each worker checks whether its load is above a 'heavy' threshold. If so it sends a HEAVY_LOAD message (b) to LIS, which dequeues the process id of a lightly loaded worker and returns a LIGHT_NODE message (c). A task is then migrated to the lightly loaded worker using a NEW_TASK message (d). Offloading is repeated until either the worker's load falls below the heavy threshold or there are no more lightly loaded processes, when the LIS replies with NO_LIGHT_NODE. Initially the root task is executed on a selected worker and the process id of the other workers are queued in LIS. After this task is subdivided the load balancing mechanism ensures work is distributed to all the workers.

Thus, by using central load information, LDSV makes better load balancing decisions
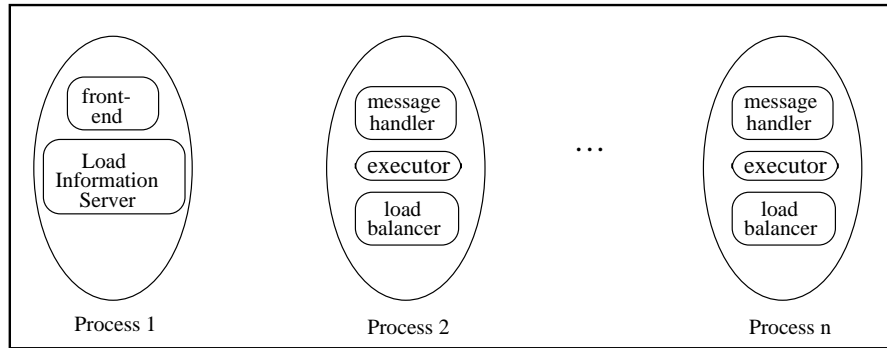
Figure 2.4: The process and thread structure of the D&C systems.

compared to purely distributed approaches. In particular, a lightly-loaded worker will always be serviced if there is surplus work in the system. The cost of maintaining this information is small: LIGHT_LOAD and HEAVY_LOAD messages are short, a lightly-loaded process notifies the LIS only once, a heavily-loaded node stops trying to offload as soon as a NO_LIGHT_NODE message is received, and the computation load of the LIS is very small (enqueue/dequeue). Most scheduling is done locally, leading to good scalability, with the extra benefit of preserving a high degree of data locality.

2.5.2 Implementation

LDSV has been incorporated in a D&C kernel using C with MPI to aid portability. To exploit maximum concurrency within the kernel it is desirable that the worker's functionality be divided among three separate processes as shown in Figure 2.4. *Executor* executes the interface functions at the tree nodes, the *message handler* deals with incoming messages containing tasks, their associated data and other system information, the *load balancer* carries out load balancing operations. These processes need a common address space enabling them to share task queues and other data structures required by the kernel. Since MPI has no lightweight thread facility and permits only one process per processor, an in-house co-routine package [27] was used to provide this service. All processes are identical except the first and second process, which acts as a front-end and LIS processes, respectively.

The task queues are double-ended; local scheduling operates in depth-first LIFO order, to minimise memory usage. Task migration uses breadth-first FIFO order, thus choosing the task likely to be largest.

Termination detection is straightforward. When the result of the root task is received

the computation is complete and a FINISHED message is broadcast.
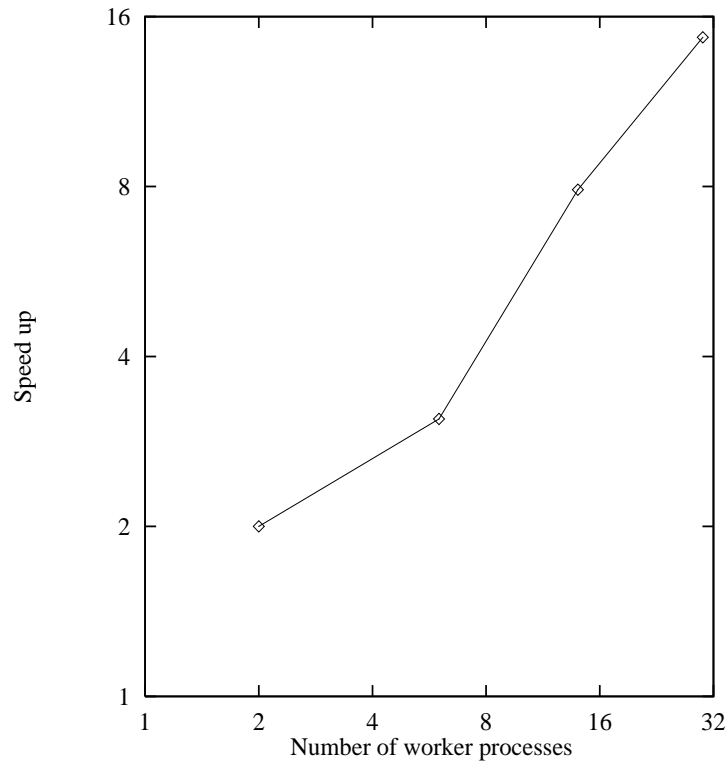
Figure 2.5: The speed up of Loadserver algorithm for Mandelbrot problem.

The LDSV algorithm was tested on the Cray T3D with the Mandelbrot problem with image size 512×512. Maximum iteration for each pixel is 1000. Figure 2.5 shows the speedup achieved by the LDSV algorithm. Each point represents the results of the number of processes instead of processors. Total worker processes is always lesser by two since one processor acts as LIS and the other the front-end. Note that the graph uses a log-log scale for x- and y-axis. Log-log plot captures the qualitative changes and relative performance well for small and large processor sizes when compared to the commonly used linear plot [10]. Hence, all the graphs on performance in this thesis used log-log plot.

2.6 Summary

In this chapter, several different existing dynamic load balancing algorithms for distributed-memory computers have been discussed. The load balancing components and the applications of interest were also described.

The chapter concludes with the implementation of a hybrid approach which combines a

centralised information and location policy, and distributed scheduling policy. Loadserver is an attempt to find a tradeoff between the better load-balancing decisions enabled by global information and migration, and the consequent overheads. The LIS provides an inevitable hot-spot but it is a very lightweight process and messages to and from it are short and relatively infrequent. All decisions in regards to transfer are done locally, aiming at maintaining scalability.

Chapter 3

Phase-based Adaptive Dynamic Load Balancing: A Framework

This chapter introduces the notion of phase-based adaptive dynamic load balancing and provides a more systematic view of the area by means of the proposed framework.

The chapter is organised as follows. Section 3.1 explains and motivates the use of phase-based adaptivity. Section 3.2 provides a simple formalism for adaptivity performance and points to future experiments. Section 3.3 compares the approach with existing related work, both on phases and adaptivity. Section 3.4 discusses practical issues that may arise. Finally, Section 3.5 justifies the method of further study.

3.1 Phase-based Adaptivity: Concepts

In this section we introduce three basic concepts in phase-based adaptivity: the workload phases, transitions and the types of adaptivity. We illustrate our idea by reference to tree-structured computation as generated by D&C and B&B algorithms (detail discussion can be found in Section 2.2). Our discussion is based on the assumption that the workload of the tree computation evolves in an idealised manner, that is a consistent increase followed by a consistent decrease in total workload forming a 'bell-shape' pattern. Practical difficulties with this assumption are discussed in Section 3.4 and Chapter 5. We then highlight that the tree computation can be divided into distinct phases and each phase has different load balancing requirement and objective.
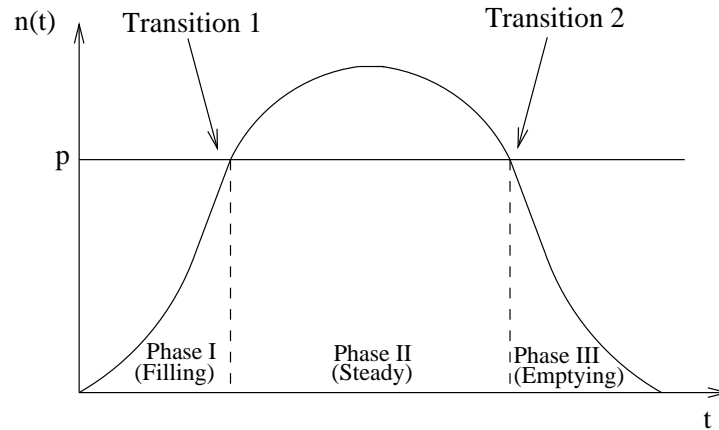
Figure 3.1: The workload phases of tree structured computation.

3.1.1 Workload Phases

The execution of a tree application typically produces a workload which evolves from one phase to another. Below we illustrate how phases can be identified in such application.

Suppose a tree application is executed on p processors and that at time t there are altogether $n(t)$ nodes available for execution; thus $n(0) = 1$, since the computation starts with the root task. It is possible to identify three main *workload phases* in the computation (see Figure 3.1).

(i) Phase I

During this phase the machine cannot be fully utilised, since $n(t) < p$. In this period the main objective of DLB is to migrate newly created tasks as quickly as possible, so as to make use of all processors. We call this the *filling phase*.

(ii) Phase II

When $n(t) \geq p$ there is enough work for all processors potentially to be busy. Unbalanced growth of the computational tree means that some processors become idle while there is still a large amount of work to be done. In this time the main objective of DLB is therefore to ensure that all processors remain busy or at least that idle periods are kept short. We call this the *steady phase*.

(iii) Phase III

Eventually the workload becomes so low that it is not possible to use all processors; as in the filling phase, $n(t) < p$. There seems little benefit to be gained from load balancing during this time. We call this the *emptying phase*.

We will argue that for tree-structured computation there is a case for considering load balancing in three distinct stages: a fast workload distribution for the filling phase, maintaining the processors busy for the steady phase, and no load balancing for the emptying phase. It should, however, be emphasised that the ability to adapt a DLB at predictable stages depends critically on the knowledge of the application as generating a tree. Our above argument would not apply, for instance, to general purpose distributed processing, where prior knowledge of pattern of workload is not feasible. Our approach involves recognising the patterns or regularities of workload and using this knowledge to change the behaviour of a DLB algorithm to optimise performance.

3.1.2 Transitions

As defined above there are three workload phases, hence we can identify two *transition* points (Figure 3.1). The first transition, t_1, marks the phase change from filling to steady phase, while the second transition, t_2, between steady and emptying phase.

In principle, both t_1 and t_2 are detected when the number of tasks reaches the total number of processors; that is when $n(t) \geq p$ for t_1 and $n(t) \leq p$ for t_2. However, in practise the best transition point may not be exactly at $n(t) = p$. It can be at any point earlier or later than $n(t) \geq p$ or $n(t) \leq p$. Hence, we introduce two variables, c_1 and c_2, which may be used to adjust these transition points. Thus, the transitions are encountered when $n(t) = c_1 \times p$ and $n(t) = c_2 \times p$, for t_1 and t_2, respectively. The values of the two variables are real positive numbers and the best values are determined experimentally.

3.1.3 Types of Adaptivity

Based on the types of adaptivity found in the literature, we classify the mechanisms for adaptivity into two categories: parametric and algorithmic adaptivity.

Parametric Adaptivity

Parametric adaptivity simply means that the values of the load balancing parameters are adapted to the dynamically changing system workload. Below are the parameters which are typically adjusted:

- Workload threshold. The workload *threshold* determines the workload state of a processor. Some algorithms, for example, LDSV and Gradient Model [48], differentiate the processor state into two; light and heavy (or low- and high-water-mark in

Gradient Model). Hence, we have *light* and *heavy* thresholds. The heavy threshold usually indicates that the load balancing algorithm should proceed to a migration phase while the light threshold indicates the node requires (more) work.

Xu and Hwang [89] use an adaptive threshold to determine if a processor is heavily loaded. The value of the threshold is periodically calculated using system load information, which is broadcasted on an adaptive *time window* basis. A similar idea on adaptive threshold can be found in [68]. The values of the threshold changes according to the large and fast load changes of distributed systems. Dasgupta in [13] used an adaptive threshold which adapts to the limited bandwidth of the shared bus architecture.

- Migration factor. The *migration factor* controls the amount of load to be migrated. It can be a 'fraction' of the total workload or a single task count. GDEM aims at equally balance the workload. Hence, it requires 'fraction' of the heavy processor's load to be transferred to its light processor pair. LDSV migrates a single task at a time since its objective is only to ensure that no processor is idle.

  Adaptive RID strategy in [9] uses an adaptive migration factor which attempts to avoid thrashing problem due to information aging which stems from a long update interval. They refer to the migration factor as adaptive diffusion factor.

- Load balancing interval. The *interval* is defined as the 'duration' between two load balancing invocations. Wikstrom *et al.* [80] use an interval of time to control the frequency of invocation. This frequency is determined based on the estimated time of the total execution time, both during the present state and the perfectly balanced state.

  For application such as the tree computation, time interval may not be applicable since the tree workload consist of distinct indivisible nodes. An interval of the number of nodes is more appropriate [42]. Assuming that the interval is i, a processor will invoke the load balancing after every i nodes.

The key idea in using the parametric approach in the phase-based technique is to ensure the objective at each phase is achieved by means of adjusting the values of the DLB parameter(s).

Ensuring a fast load distribution during phase I suggests the use of a low interval value, possibly one. If the migration factor is to be adapted, this may mean that a small number of tasks ought to be migrated (for example a single task). If the workload thresholds are to

be manipulated, a low value of the heavy threshold (e.g. one) may be appropriate so that the heavy processor can quickly off-loaded the task. A low value of the light threshold is recommended too, for example a value of zero, possibly, to quickly receive the load. The basic idea behind the low values of DLB parameter is to minimise the idle time.

During the steady phase, there is less need for load balancing, therefore, the frequency of the DLB invocation ought to be reduced. This can be done by increasing the interval or the heavy threshold. Using the latter may incurs the extra cost of load evaluation. So, increases the interval is probably to be preferred when possible. A larger value of migration factor may be necessary to reduce the level of load imbalance.

During the emptying phase the interval (or the heavy threshold) can be set to infinity (or in practise the value of MAX_INT) to avoid invoking the load balancing completely. Manipulating the migration factor, however, will not give the disabling effect.

It may well be that adapting any one parameter may not lead to the best performance. Perhaps the combination of a few (or all the parameters) may yield the best results.

Algorithmic Adaptivity

In *algorithmic adaptivity* completely different algorithms or different DLB policies (e.g. receiver-initiated followed by sender-initiated policies) are used during a single program execution.

We shall now illustrate the use of algorithmic adaptivity in the context of parallel tree computation by making use of the knowledge of phases defined earlier. During the filling phase the DLB algorithm must facilitate rapid migration of work. The local information and migration space of Diffusion is quite restrictive, whereas the global information and migration space of LDSV are more effective.

This can be simply demonstrated in the case that all node executions lead to expansion. Suppose that the diameter of the underlying communication network is D and the degree (or the fan-out) of the computation tree is f (i.e. each node expands to give f children). Suppose also, for simplicity, that the computation proceeds in 'rounds', with each processor executing one task (if it has one) and then carrying out its load balancing operations. With LDSV, in the first round $(f - 1)$ tasks can be offloaded to distinct processors, whatever the topology, allowing f processors to expand during the second round. It is clear that after $\log_f p$ rounds p tasks will have been generated and also that all processors will now have one task, so the machine is fully utilised. Thus, LDSV is optimal in terms of the number of rounds in the filling stage. One may expect other DLB algorithms with a global

benefit of
load balancing

cost of
load balancing

t
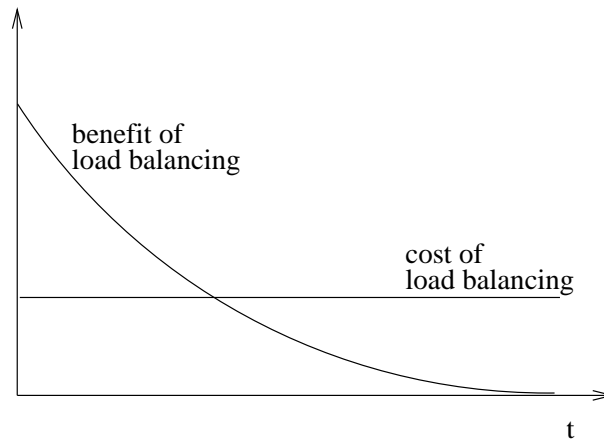
Figure 3.2: Cost-benefit of load balancing.

information and migration space to be similarly optimal.

With Diffusion, work moves through at most one hop per round, hence $\Omega(D)$ rounds are needed to fill the machine. Clearly, this is a worse result than for LDSV, especially for larger values of f on D. Other approaches with a local migration space will perform similarly.

During the steady phase, however, the advantage appears to swing towards Diffusion. In LDSV , when all the processors are busy, they continue to make requests to LIS after every task execution, without ever gaining any benefit from load balancing, and causing worst case contention at the LIS. By contrast, the distributed, local nature of Diffusion avoids global hot-spots; more significantly, receiver-oriented Diffusion will make requests only when work is actually needed, significantly reducing the overheads of load balancing. Similar results can be expected from other local algorithms, especially receiver-oriented. This accords with the observation by Eager *et al.* who provided evidence for the potential algorithmic adaptivity [19]. They reported that sender-initiated algorithms outperformed receiver-initiated at light to moderate system loads, whereas receiver-initiated algorithms were preferable at high system loads. For their case, perhaps the initiation policy could be adapted according to the system load level.

During the emptying phase, it is impossible to keep all processors busy; if possible, load balancing should be switched off, saving the overheads. More general, load balancing should only be carried out when its benefits outweigh its costs, as shown in Figure 3.2. In the early stages of a tree computation there is a high benefit from load balancing, but as time goes on this decreases to zero. On the other hand there is always a cost for load balancing, conceptually shown as constant. Thus, there will be a stage in any computation
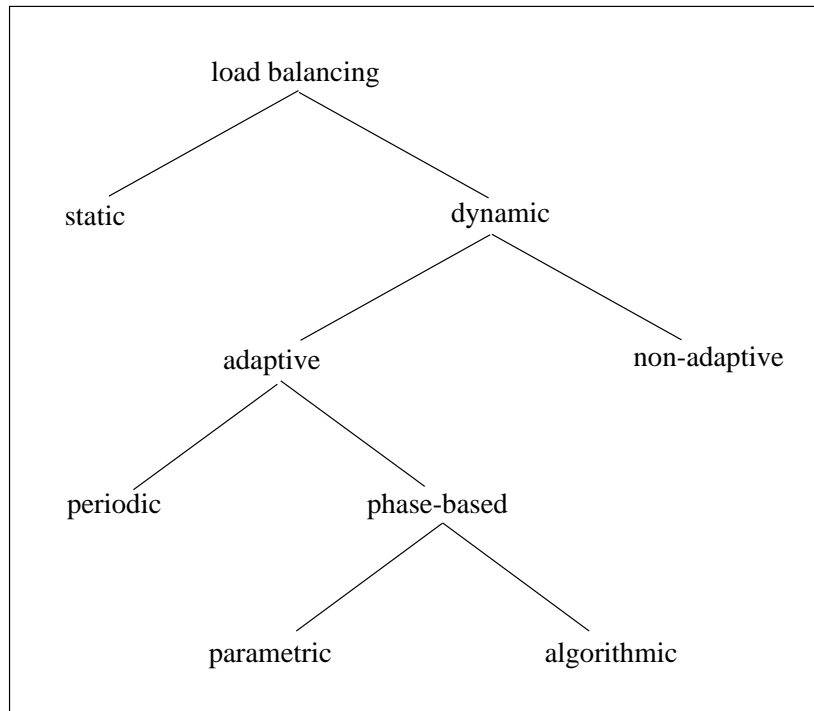
Figure 3.3: The characteristics of adaptive dynamic load balancing algorithms.

where further effort in load balancing no longer brings any net benefit. Discontinuing load balancing at this stage should improve performance.

We conclude that different load balancing algorithms could be used in different stages of a parallel tree computation; an algorithm with global information and migration space for the filling phase, a local algorithm during the steady phase and no load balancing during the emptying phase.

Our new framework described in this section is best seen as a further extension to and a refinement of the present work on adaptive DLB. The relationship between the framework with the existing load balancing work is depicted in Figure 3.3. Notice that the upper part of the classification is actually the taxonomy which appeared in Chapter 1 (Figure 1.1).

In our framework, we refer to all non-phase-based adaptive DLB as *periodic* approach, where the DLB algorithm is invoked periodically through out the execution. Clearly, there is a potential for further characterisation of this class of DLB algorithms, which is beyond the scope of this thesis.

3.2   Parameter Definitions and a Simple Formalism

This section attempts at providing a simple formalism for the performance of a tree application using the phase-based adaptivity approach described in the previous section.

The performance of a parallel tree application is a complex issue; a function of many interrelated parameters. These parameters can be categorised into three categories, namely, machine, application and load balancing parameters. Apart from these, we also include a discussion on parameters which are used in detecting transitions.

For the machine parameters we consider:

- Varying processor numbers, p, which determine the size of the network. The value of p is an integer of a power of two, in the range of 1 to 512.

- Varying network performance or speed, s, which is relative to the communication performance of the T3D. This parameter is defined precisely in Section 5.3.2. Intuitively, a value of s improve network performance by a factor of s compared with the T3D, i.e. increasing s gives a faster network.

In order to support various kinds of task trees. We have the following parameters:

- The level of depth, d, with the root node starting at depth one. In general, the depth of a tree determines the total workload.

- The node grain size, g, which is measured by the number of floating point operations.

- The fan-out, f, which may be fixed or variable. In theory this can be variable, but we have only used fixed fan-out which can either be 2, 4 or 8.

- The degree of tree imbalance, m, which is in the range of 0.0 to 1.0. An imbalance of 0.0 means the tree is completely balanced while an imbalanced of 1.0 implies the worst case of imbalance.

Detailed discussion on application parameters can be found in Sections 4.2.3 and 4.3.1. For the load balancing we define the following parameters:

- The load balancing interval, i.

- The low workload threshold, l.

- The high workload threshold, h.

- The migration factor, r.

All the above DLB parameters have been described in Section 3.1.3.

We shall now define the parameters used for detecting phase transition (further discussion can be found in Section 3.1.2):

- The condition for transition I is $n(t) \geq c_1 \times p$, where c_1 is a positive real value relative to p. For instance, c_1 values of 0.5, 1.0 or 1.5, mean that the transition occurs when the machine is half full, totally full or more than totally full, respectively.

- The condition for transition II is $n(t) \leq c_2 \times p$, where c_2 can be any positive real value. The larger c_2 is the earlier the transition occurs.

Assuming that a parallel tree application runs in a non-adaptive setting using the above defined parameters, we could then formalise the total execution time of that application, T_{NA}, as the following:

$$T_{NA} \quad = \quad t_{NA} \quad (\quad \underbrace{p, s,}_{machine} \quad \underbrace{d, f, g, m,}_{application} \quad \underbrace{i, h, l, r}_{loadbalancing} \quad) \tag{3.1}$$

The total execution time in an adaptive environment, T_A, depends on the adaptive mechanism used. For parametric adaptivity all DLB parameters may hold different values during each phase. The above equation then becomes:

$$T_{PA} = \quad t_{PA} \quad (p, s, \quad d, f, g, m, \quad c_1, c_2, \quad (i_1, i_2, i_3) \quad (h_1, h_2, h_3) \quad (l_1, l_2, l_3) \quad (r_1, r_2, r_3) \;) \tag{3.2}$$

where the subscripts of the four DLB parameters refer to phase I, II and III. Note that the above equation assumes that a DLB algorithm has high and low threshold values. If the algorithm has only one threshold, only one of the parameters (either l or h) will appear in the equation.

Due to the wide range of parameters, for this thesis we limit our investigation to one DLB parameter only, that is the load balancing interval, i. The rest of the DLB parameters will not be adapted. Thus, T_{PA}, can be reduced to;

$$T_{PA} = \quad t_{PA} \quad (p, s, \quad d, f, g, m, \quad c_1, c_2, \quad (i_1, i_2, \infty) \quad h, l, r) \tag{3.3}$$

where i_1 and i_2 are the best interval for phase I and II, respectively. i_3 assumes infinity which has the effect of disabling the algorithm during phase III.

For comparison purposes we use the same formalism to express the performance of algorithmic adaptivity, T_{AA}, though recognising that algorithms are not parameters in the same way as the rest of the parameters. Thus, the performance of algorithmic adaptivity can be expressed as:

$$T_{AA} = \quad t_{AA} \quad (p, s, \quad d, f, g, m, \quad c_1, c_2, \quad (a_1, i_1, h_1, l_1, r_1) \quad (a_2, i_2, h_2, l_2, r_2) \quad (nil) \,)$$

$$(3.4)$$

where a_1 and a_2, are the two different DLB algorithms used during phase I and II, respectively. Each algorithm assumes its own DLB parameters i, h, l and r, as defined in the original algorithm. The last parameter "nil" means no DLB algorithm is used during phase III.

As a conclusion, we can say that the adaptivity performance, T_A, assumes either the performance of parametric or algorithmic adaptivity;

$$T_A = T_{PA} \quad or \quad T_A = T_{AA} \tag{3.5}$$

3.3 Related Work

The idea of phase-based adaptivity is the combination of two existing notions; phases and adaptivity. Hence, we separate the related work according to these two concepts.

3.3.1 Phases

There exist several works in the literature which make use of phases to optimise performance. Some of these identify phases based on the different stages of computation which exist in the application. There is also work which does not assume any knowledge of the application *a priori*, but phases are identified by means of program trace after the computation ends.

An example application which has a clearly identified phases is parallel ray-tracing in Mistral in solid modeling system [14]. The execution of the application proceeded in two distinct stages or phases (see Figure 3.4 (a)). In the first phase a distributed octree representation of a solid is built using a D&C method with dynamic load balancing. In the second phase the ray-traced image of the solid is produced by a further dynamically load-balanced D&C method, using a quadtree decomposition of the image space. Between

```
program  Mistral                        program Particle-in-Cell


    computation 1  }  Phase I              loop
    computation 2  }  Phase II                computation 1  }  Phase I
                                              loop
endprogram                                        computation 2  }  Phase II
                                              endloop
                                          endloop
                                       endprogram

              (a)                                      (b)
```
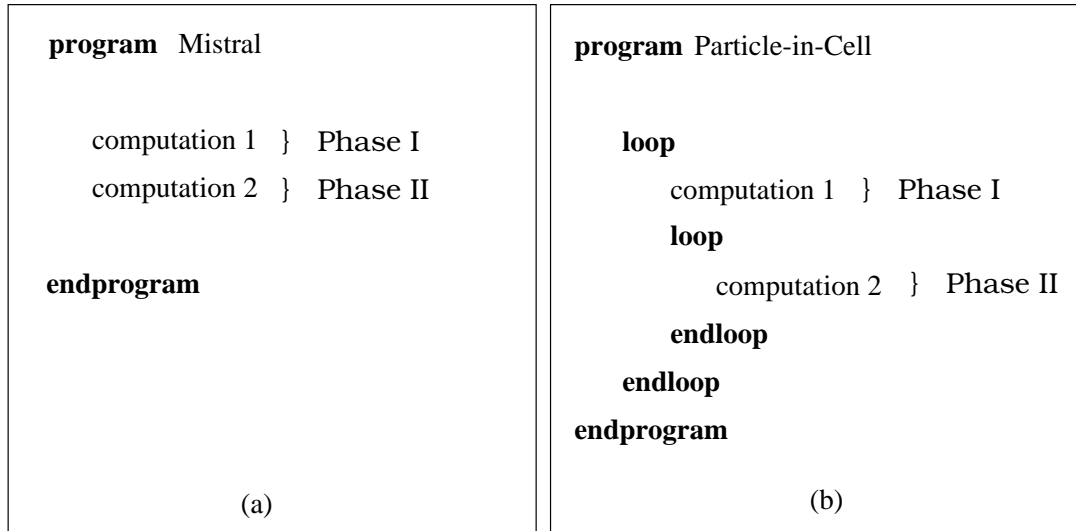
Figure 3.4: (a) Clear-cut phases of Mistral. (b) Nested phases of PIC.

the phases there is a redistribution of data to improve the performance of the second phase.

The system is similar to our proposed framework in that it involves two distinct phases, both based on dynamically load balanced tree computation (i.e. D&C). However, there are two significant differences. Mistral has two separate tree computation, each with its own DLB approach, our framework considers phases within a single tree computation. The two phases are clearly evident in the application and are explicitly within the source code. In our framework, only one computation is visible to the programmer and the underlying DLB system switches between phases depending on run-time load observations. Mistral's phases can be considered *explicit*, in our framework they are *implicit*.

Clearly, Mistral represents a wide range of program which have a clearly identified phases with different pattern of parallelism. From the above observations it should be clear that this is not the scenario with which our framework is concerned.

Problems involved in load balancing multi-phase parallel computation are addressed in [79]. The example here is a Particle-in-cell (PIC) application which is used for simulating highly rarefied particles that flow in the presence of an electromagnetic field. A high level view of the application is shown in Figure 3.4 (b), showing two main phases, with the second nested within the first phase. The two phases have quite different load balancing requirements and involve different load metrics. A load balancing scheme is proposed which balances both phases together, by considering a vector of the loads for individual phases and using the Diffusion algorithm.

Application/System	Key Features
Mistral	Algorithm-based phase identification (explicit phase). Compile-time phase detection. Dynamically load balance in each phase.
PIC	Algorithm-based phase identification (explicit phase). Compile-time phase detection. Dynamically load balance at the end of both phases.
MARS	Program trace-based phase identification (explicit phase). Post-execution phase detection. Statically load balance in each phase.
Our work	Workload-based phase identification (implicit phase). Run-time phase detection. Dynamically load balance in each phase.

Table 3.1: A comparison between our approach and the related work on phases.

As with Mistral the phases are explicitly identified within the application and the same differences from our framework apply.

MARS (Metacomputer Adaptive Runtime System) is a framework for minimising the execution time of distributed application on a wide-area-network [24, 25]. MARS utilises application-specific information by identifying the program phases to improve task-to-processor mapping. An off-line analysis is made from previous execution runs to build a task dependency graph from which phases can be identified. A phase is defined as a sub-graph of the dependency graph which performs a closed subtask, such as loops or frequently called function-bodies. Each phase is statically mapped to the underlying processor network, but there may be remapping between phases implemented by data redistribution. A cost model is used to evaluate whether the cost of remapping will be offset by improved performance in the subsequent phase.

In the case of MARS, phases are detected based on run-time measurements, as in our system. A major difference is that the phase detection is done off-line from the program trace, leading to an optimal static schedule. No dynamic load balancing is ever involved. MARS is therefore not suitable for the unpredictable problems which are handled by our framework.

Table 3.1 summarises the similarities and differences between our approach and the related work on phases.

Special mechanisms for the early stage of a parallel tree computation (equivalent to our filling phase) are described in [62, 67]. Both systems use the Iterative Deepening

A* algorithm [66] to solve the 15-puzzle problem. In [62], newly expanded node are always migrated to free processors, though no indication is given of how it is known which processors are free. Since this implies a global migration space, it supports our hypothesis that global migration is useful for this stage. In [67] the root node is broadcast to all processors which redundantly expand the first few levels of the tree, obtain the same set of subtrees n_1, n_2, ... During the main phase of the computation, processor i then starts to expand its sub-tree $n_i, n_{p+i}, ..., n_{2p+i}$, ensuring that all processors expand different subtrees for most of the computation. It is worth noting that in both the above systems there is a very large processor array (16 K-node SIMD and 1024-node MIMD, respectively) which makes the effects of the filling phase quite significant. Neither of these systems explicitly identifies an emptying stage, though [62] includes an adaptive mechanism in its main computational phase which reduces the number of processors involved towards the end of the computation.

3.3.2 Adaptivity

A number of instances of adaptivity in DLB have been described in previous chapters, specifically in Sections 1.2, 2.3.3 and 3.1.3. We further describe two work that are related to ours. The first is an adaptive system which bears some similarities in terms of classifying the adaptive mechanisms. The second is adaptivity in the context of programming.

Recognising that no single scheduling algorithm performs well in all situations, Ramamritham and Zhao proposed a higher level control of scheduling for distributed real-time system [64]. This added layer is referred as *meta-level control*, which controls the selection of a local scheduling algorithm, a global scheduling algorithm and the section of scheduling parameters. The local algorithm is used for scheduling tasks on a node, while global scheduling is to control cooperation among the nodes.

Their work is similar to ours in the sense that both distinguish the mechanisms of adaptivity, into parametric and algorithmic-related mechanisms. However, the context is very different, and the work is not dealing with phases which are identified at run-time by workload measurements.

All the work on adaptivity described so far focuses on improving the performance – aiming for a faster, more reliable and stable system. The work by Gouda and Herman [29] attempts to reason about the behaviour of adaptive programs. They formalised the definition of adaptivity and present some logical properties of the definition and provide operators for combining adaptive programs. They first study an adaptive sequential

program and later adaptive distributed program. Their targeted applications include a system of distributed processes that communicate via a shared bus, a distributed system for traffic control and a resource allocation program to minimise the amortised cost of resource allocation.

The work is similar to ours in that it considers aspects of adaptivity between known phases, based on run-time measurements. The emphasis, however is very different, the work focuses on methods for constructing adaptive programs and ensuring correctness in the transition between phases. Issues of performance and dynamic load balancing are not in consideration.

3.4   Some Practical Implications

The discussion so far has been based on an idealised workload represented by the bell-shaped curve in Figure 3.1. In practise it is unlikely that the workload curve will be as clear-cut; in particular the early and final stages of the curve might not be monotonic, making detection of precise transition points more difficult.

A further difficulty in detecting transition points arises because of the need to evaluate the total workload in the system. Since most DLB algorithms do not maintain global load information, some additional facilities must be introduced, adding to the complexity of transition detection.

Assuming that transition points are correctly detected there remains the challenge of implementing the transitions. For parametric adaptivity there appear no major problem in principle, but for algorithmic adaptivity there may be problems of making a clear transition between DLB algorithms, such as supporting different definition of process groups and communication patterns, and ensuring any outstanding messages are handled properly.

3.5   Use of a Simulator

The previous section indicates that the implementation of phase-based adaptivity is not straight-forward and will involve tackling non-trivial problems. Before engaging in such implementation it is therefore important to be sure that there will be significant performance benefit, particularly as the implementation of phase changes is likely to introduce extra overheads. Clearly, it is not possible to gain such preliminary quantitative data by means of parallel implementation! Therefore, we are left with two methods of continuing

the study, analytical modeling or simulation.

Analytical modeling can gain quick results if the problem is tractable. Unfortunately, the complex behaviour of the parallel execution of a DLB algorithm is difficult to capture analytically, and this approach does not seem promising.

Simulation, on the other hand, is a well-tried technique which avoids the need to resolve the implementation issues noted above, and does not require a full analytical model of the parallel computation. It also permits the performance on different hardware configurations to be obtained by parameterisation, and can study the effect of varying workload by appropriate parameterised models. Simulation also gives greater control over some of the behaviour of the system under study. Some of the tree application (e.g. B&B) produces a different search space from one run to another [65]. In simulation such behaviour can easily be controlled by the use of artificial tree which generates the workload in a deterministic manner – the same shape, depth, fan-out and hence the same total workload. Furthermore, simulation allows unlimited number of factors which affect performance to be monitored (or detail measurements to be collected) without any adverse side effect.

Practical problems such as limited CPU time and interactive runs for parallel machine also make simulation an attractive alternative. The maximum number of processors allowed on the T3D is only 32 processors (i.e. small scale processor size). Medium-scale (64 - 128) and large-scale (256 - 512) processor sizes must be submitted as batch jobs. The latter can only be run on a specified day. Clearly, such limitation is inconvenience for extensive experimentation.

The next chapter describes the design, implementation and validation of a suitable simulator.

Chapter 4

Simulation: Design, Implementation and Validation

This chapter presents the design and implementation of a simulator for the purpose of carrying out systematic experiments on the various aspects of phase-based adaptivity described in Chapter 3. Factors such as varying topologies, different load balancing algorithms, varying workloads and different cost models are also considered.

The chapter begins with a description of the general model in Section 4.1 and the components of the simulator in Section 4.2. This is then followed by a discussion on the performance models used and the calibration of the models in Section 4.3. Section 4.4 details the validation process. Finally, Section 4.5 discusses the limitations of the simulator.

4.1 The Model of Parallel Execution

The general model of parallel execution adopted in the simulator is similar to the BSP computational model. In BSP [54] computation is divided into a sequence of phases, called *supersteps*. Each superstep is delimited by a barrier operation, within which each processor performs local computation and global communications independently. The results of the communication in a superstep are used in the subsequent superstep. In this thesis the term *iteration* will be used instead of superstep.

Figure 4.1 shows the outline operation of the simulator while Figure 4.2 depicts a sequence of operations that may occur in an iteration (note that some processors may not execute any computation in a specific iteration). The local computation in this case is the

```
while not finished do
    compute ()
    load balance ()
    barrier ()
end
```

Figure 4.1: An overview of the simulator.

execution of tree computation and the global communication is the information exchange and data migration used in dynamic load balancing. This synchronous approach is simple and easy to simulate, thus making a study on adaptivity tractable.
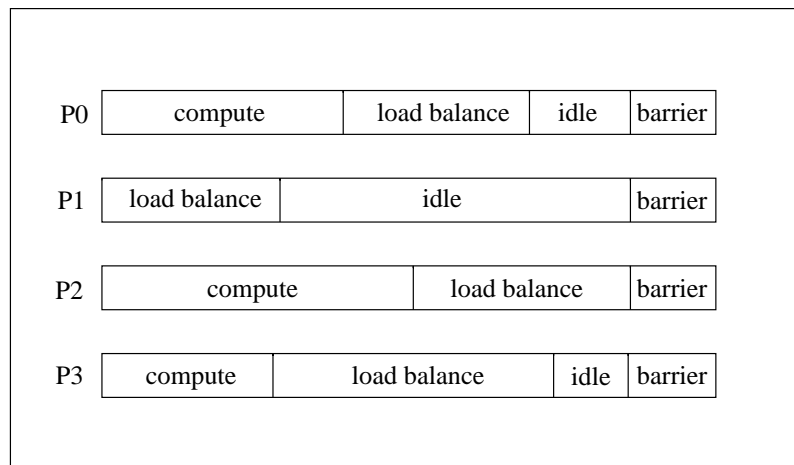
Figure 4.2: Processor operations in a single iteration.

The drawback of this approach is that it assumes a synchronous DLB algorithm. Any asynchronous DLB algorithm, e.g. the LDSV, must be transformed to an equivalent synchronous version. This is achieved by adding barrier synchronisation which inevitably yields poorer performance compared to the original asynchronous version. In short, a synchronous version induces a larger overhead. Despite this, the pattern of results for adaptivity experiments is expected to be similar, whether a synchronous or an asynchronous algorithm is used. The amount gained perhaps may be larger for the synchronous model due to the extra gain from the larger overhead, but this should not invalidate the results. The assumption is that the effects of adaptivity are essentially independent of the model adopted.

4.2 The Load Balancing Simulator

The simulator simulates the synchronous execution of a DLB algorithm on the Cray T3D with tree-based computation as the workload generator. It is written in C and MPI, with a parameterised performance model based on the T3D performance model. It supports a range of topologies and is capable of simulating trees with varying depth, degree and imbalance. This constitutes the generality of the simulator as a load balancing testbed.

In order to support the above capabilities, the simulator has been cleanly implemented in four separate modules with each having its own variants. The modules are:

- the topology,

- the load balancing,

- the tree computation, and

- the cost module.

The remainder of Section 4.2 describes the implementation of the first three components and Section 4.3 discusses in detail the cost models used.

4.2.1 The Topology Module

The topology module supports five different topologies; chain, ring, 2-d mesh, 2-d torus and hypercube (Figure 4.3). They are from the k-ary n-cubes family, where n refers to the dimension of the network and k is the number of processors along each dimension [12]. A ring is a one-dimensional structure with k processors along its only dimension. An n-dimensional hypercube (or binary n-cube), on the other hand, is a mesh with two processors in each dimension. These two topologies define the extremes, or the special cases, of k-ary n-cubes. Stripping off the wrap-around connections of a ring and torus result in a chain and a mesh, respectively. The family of k-ary n-cubes forms the basis of several commercial parallel computers. Examples include the hypercube-structured Intel iPSC/860 and NCUBE/2, the two-dimensional mesh-structured T9000 and the three-dimensional torus Cray T3D [77].

The above five topologies were supported using two main data structures, namely an array of array of neighbouring processors and a structure of topology parameters (Figure 4.4). The indices of the array were used as the processor ids, while the sub-arrays kept the processor ids of the neighbours. The simulator adopts the MPI virtual topology style in labeling the processor id, that is column-major ordering [30]. The topologies in Figure

(a) A chain

(b) A ring

(c) A 2-d mesh

(d) A 2-d torus
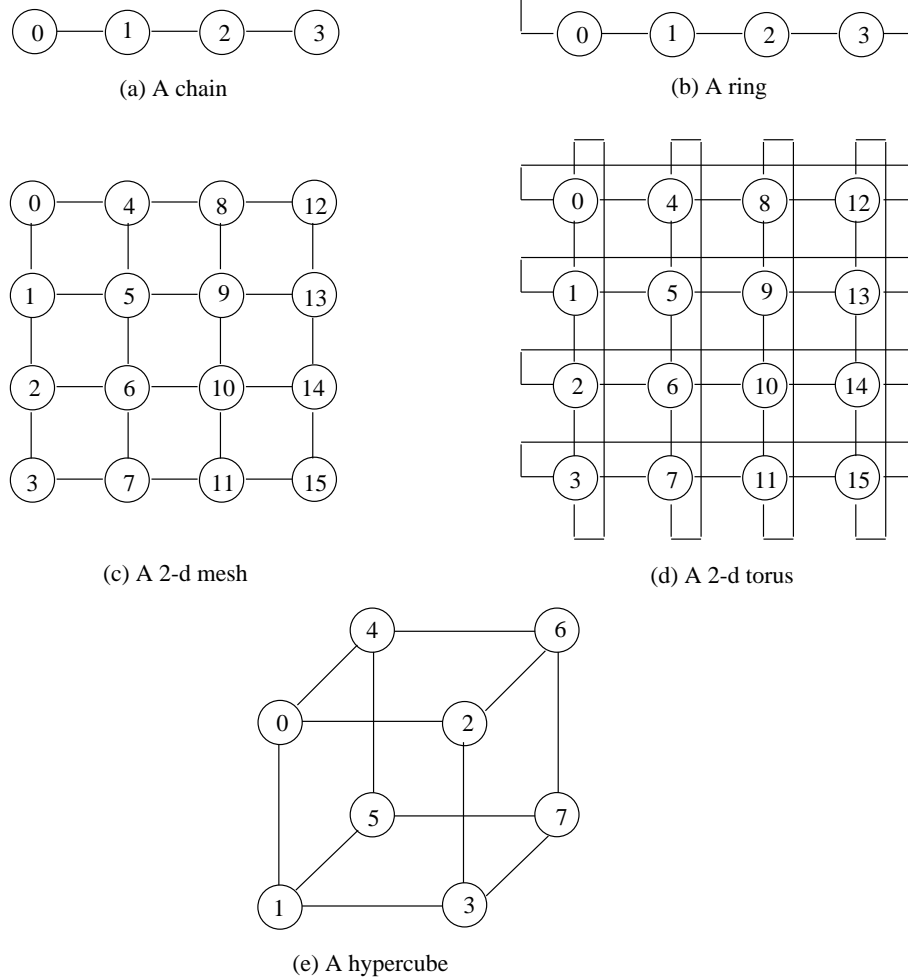
(e) A hypercube

Figure 4.3: Topologies supported in the simulator.

4.3 follow the MPI processor labeling convention. The characteristics of a topology, such as type (e.g. 2-d torus, hypercube etc.), the total number of processors in the network, the length of each dimension and the maximum number of neighbours in the network, are kept in a single data structure.

A load balancing algorithm which is topology-dependent (e.g. GDEM) requires a layer that maps the algorithm requirement to the topology interconnect. For example, GDEM on the 2-d torus uses the knowledge of the four neighbouring processors during each balancing operations. The four neighbours correspond to the four colours in the edge-colouring method (described in Section 2.3.2). Balancing in each dimension (or colour) involves all processor pairs in the same dimension. A separate routine is needed to identify these processor pairs as the balancing operation proceeds.

```
typedef struct {                            /* topology parameters      */

        int   type,                         /* topology type            */
              p,                            /* total processors         */
              x,                            /* total processors along x */
              y,                            /* and y dimension          */
              k,                            /* maximum between x and y   */
              neighbours;                   /* maximum neighbours       */
} to, *to_ptr;


typedef struct {                            /* topology main structure  */
        int   neig[MAX_NEIG];               /* directly connected neighbours*/
} proc, *proc_ptr;

proc    p[MAX_PROC];                        /* all processors in the network*/
```

Figure 4.4: Data structures that support the topologies.

4.2.2 The Load Balancing Module

The load balancing module supports two different load balancing algorithms, GDEM and LDSV, which have been described in Chapter 2. Both were implemented using the synchronous model.

The key issue in simulating these algorithms sequentially is how parallelism can be successfully achieved. The main idea is to 'execute' the load balancing operation of each processor in turn. However, the implementation depends on the specific load balancing algorithm. Below are the description on how GDEM and LDSV are simulated. The algorithms exhibit a different level of complexity and simulation characteristics.

Generalised Dimension Exchange Method (GDEM)

In the real implementation of GDEM, each processor calls the topology setup routine during the initialisation stage, i.e. prior to executing any computation. It then executes the **for** loop in Figure 4.5 to do the balancing, by pairing itself with its partner of the same colour, exchanging the load information and migrating the excess workload (if any). The pairwise balancing is repeated for every colour (or dimension).

The simulated GDEM (Figure 4.6) is very similar to the real version except for two aspects. The first is that it has to have a module to support the topological requirement of the algorithm (see comments in Figure 4.6), which in the real version is provided by

```
procedure Real_Generalised_Dimension_Exchange (pid)
    i := pid
    for colour := 1 to max_colour do          { load balance each dimension    }
        j := get_partner (i)                   { partner in the dimension       }
        exchange_load_information (i, j)
        calculate_load_difference (i, j)
        if load difference > 1 then
            if load (i) > load (j) then
                migrate_task (i, j)            { migrate packed of data         }
            else migrate_task (j, i)
        endif
    endfor
endprocedure
```

Figure 4.5: The real Generalised Dimension Exchange algorithm.

```
procedure Simulated_Generalised_Dimension_Exchange ()
    for colour := 1 to max_colour do
        i := 0
        get_first_pair (i, j)                         { topology support routine       }
        while not finished visiting all pairs do      { parallelise balancing operation }
            exchange_load_information (i, j)
            calculate_load_difference (i, j)
            if load difference > 1 then
                if load (i) > load (j) then
                    migrate_task (i, j)
                else migrate_task (j, i)
            endif
            get_next_pair (i, j)                       { topology support routine       }
        endwhile
    endfor
endprocedure
```

Figure 4.6: The simulated Generalised Dimension Exchange algorithm.

the MPI virtual topology facilities. The second involves the parallelisation of the load balancing. This is achieved using a **while** loop which traverses through each processor pair of the same colour, simulating the parallel execution of pairwise balancing in the **for** loop of the real version. The processor pairs involved in load balancing are determined prior to each balancing operation using function *get_first_pair* (*i, j*) and *get_next_pair* (*i, j*). Thus, the only differences between the simulated and the real GDEM are the additional topology routines which have to be implemented, and the parallelisation of the localised pairwise load balancing.

Loadserver (LDSV)

In the real synchronous Loadserver algorithm (Figure 4.7) [1], each processor checks its load status and if the load is below a 'light' threshold it informs the Load Information Server (LIS), provided that it has not already done so. If the load is above the 'heavy' threshold, the processor will keep on off-loading its load as long as there exists a 'light' processor. Once completed it waits for instructions allowing it to proceed from processor 0. This broadcast 'proceed' to the next stage does not exist in the original algorithm. Since the model employed is a synchronous approach, it forces LDSV to be synchronised.

The simulated LDSV does not differ radically from the real version (Figure 4.8). The section for testing 'light' load remains the same. The only section that differs is when 'heavy' processors repeatedly off-load tasks to 'light' processors. In the real execution, all 'heavy' processors execute the loop "**while** heavy_load (*pid*)" in Figure 4.7 in parallel; with each processor making a request to LIS. Each will then continue making requests if it is still in a 'heavy' state. Simulating this behaviour sequentially requires the visit to each 'heavy' processor be carried out in rounds until all processors know that either the LIS is empty or all the heavy processors becomes 'light'. In order to reduce the sequentialisation effects, the first processor to be 'executed' is chosen at random each time the load balancing is invoked.

Notice that the loop "**while not** *receive_proceed*" does not exist in the simulated version. This is simplified since there is no actual message passing library is used. The receiver is costed according to the relevant cost function. One of the main advantages of the sequential simulation is that the complexities introduced by message-passing programming are reduced.

---

[1] Note that a discussion on the original asynchronous version of the LDSV, including its implementation, is described in Section 2.5.

procedure Real_Loadserver (*pid, acknowledge_LIS*)

                                               { `case 'light' processor`   }
  if light_load (*pid*) **and not** *acknowledge_lis* **then**
     send_LIGHT_to_LIS (*pid*)
     acknowledge_lis := TRUE
  else   { `case 'heavy' processor`   }
     finished := FALSE
     while heavy_load (*pid*) **and not** *finished* **do**
        send_HEAVY_to_LIS (*pid*)
        tag_received := receive_reply_from_LIS ()
        if *tag_received* = LIGHT_NODE
           send_task (*pid*)
        else *finished* := TRUE
     endwhile
  endif
  if *pid* = 0 **then**
     send_PROCEED_to_all_processors ()
  endif
  receive_proceed := FALSE
  while not *receive_proceed* **do**   { `loop for an incoming message`   }
     wait_to_receive_message (*message_type*)
     case *message_type* **of**
        TASK:
           enqueue (*task*)
        PROCEED:
           receive_proceed := TRUE
     endcase
  endwhile
endprocedure

Figure 4.7: The real Loadserver algorithm.

In order to support the adaptivity experimentation, the structure definition shown in Figure 4.9 is employed. The variables t1 and t2 indicate whether the first and second transition for adaptivity is on, while c1 and c2 are their respective coefficients. The "no_lb" variable is a flag which indicates whether a load balancing algorithm is to be activated or not.

4.2.3 The Tree Computation Module

We need to model different kinds of workloads. For the class of computation considered, this means different kinds of task trees are required. The following variations are

```
procedure Simulated_Loadserver ()
    for i = 0 to P do                              { assumed light pid exist in LIS    }
        know_LIS_empty[i] := FALSE

    loop_again := TRUE
    while (loop_again) do                          { parallelise access to LIS         }
        loop_again := FALSE
        for all processors i in random order do    { parallelise the load balancing    }
                                                   { case 'light' processor            }
            if light (i) and not registered_to_LIS (i) then
                enqueue_LIS (i)

                                                   { case 'heavy' processor            }
            else if heavy (i) not know_LIS_empty[i] then
                destination := dequeue_LIS ()
                if destination <> NO_LIGHT_NODE then
                    migrate_task (i, destination)
                    loop_again := TRUE
                else know_LIS_empty[i] := TRUE
            endif
        endfor
    endwhile
    p0_send_proceedtoall_processors ()
endprocedure
```

Figure 4.8: The simulated Loadserver algorithm.

considered:

- Different degrees or *fan-out*, f, which may be fixed or varying. Trees of degree 2 or binary tree (e.g. parallel mergesort), 4 (e.g. quadtree algorithms) or 8 (e.g. octree algorithms) are common. Within any one computation, this degree is fixed.

- Different amounts of unbalance, m. Balanced trees are easy to simulate (and are included for completeness) but DLB is only useful when the workload is unpredictable, and the tree is therefore unbalanced.

  The simulator supports two kinds of unbalanced tree:

  (i) A *random* tree, in which the tree structure is guaranteed to be the same in each run of the same tree parameters, allowing easy comparison of results.

  (ii) An *imbalance* tree, in which the amount of *imbalance* is parameterised in the range of 0.0 to 1.0 (0.0 is fully balanced and 1.0 is completely unbalanced),

```
typedef struct {                              /* load balancing parameters    */

    int         i,                            /* load balancing interval      */
                t1,                           /* mark transition 1 and 2      */
                t2,
                curr_algo;                    /* the type of algorithm used   */
    float       c1,                           /* coefficients for transitions */
                c2;
    boolean     no_lb;                        /* load balancing is switch off */

} lb, *lb_ptr;
```

Figure 4.9: Data structure to support load balancing.

this allows a systematic study of the effects of varying imbalance to take place. However, the precise shape (and the number of nodes) of the tree may vary on each run.

- Different total size as reflected in the number of nodes. For a given degree and imbalance, this is primarily affected by depth. So, we have *depth*, *d*, as a parameter, beyond which no subdivision occurs.

The Unbalanced Tree

The approach to generating both types of unbalanced tree is motivated by the intuition that the likelihood of dividing decreases as the depth of the tree grows.

<u>Random tree</u>

Each node in the random tree will divide or split if the random number generated is lesser than the probability function used [58]. One choice of a probability function is a linear function which decreases with depth. It is possible to have the function to start decreasing the probability only after a certain minimum depth. In other words, the tree expands in full till the minimum depth, and only then starts splitting. Figure 4.10 shows the number of nodes created for different minimum splitting depths, with $f = 2$. The total nodes is the minimum when splitting is allowed from depth 0.

The repeatable characteristic is achieved by generating the same random number for a given node id. The repetitive random number is generated by multiplying it with constant coefficient throughout the run. The result is then hashed to get a value within a specified
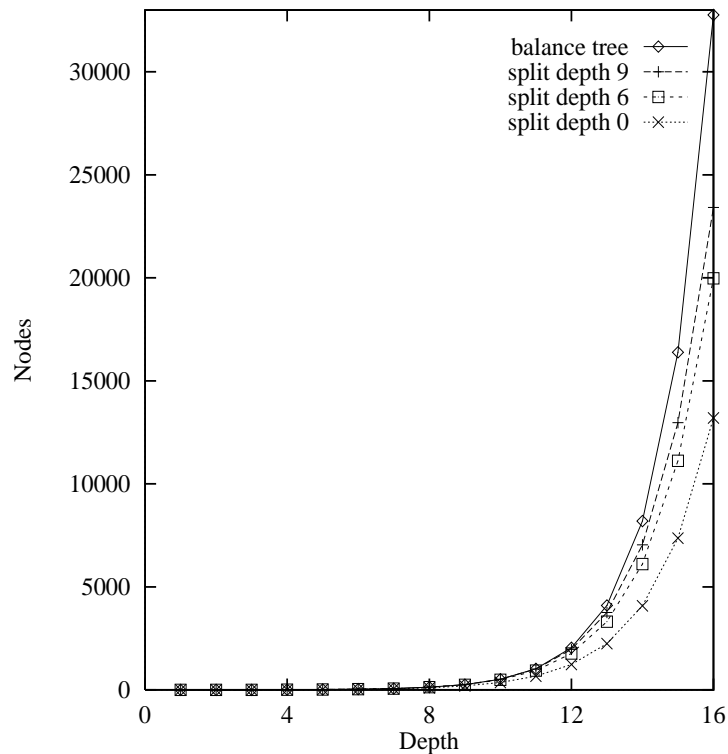
Figure 4.10: Total nodes created by the balance tree and the random tree in each depth (the latter with different minimum depth of splitting).

range. The random number is then compared with the value of the probability function. If it is smaller, then child node will be created. This repeatable characteristic ensure correctness and ease of experimentation.

Imbalance tree

The basic idea behind the imbalance tree is that the root node and the right-most child node always do the splitting. The left child will only split if the random number generated is lesser than or equal to the specified degree of imbalance [15]. The imbalance value of 0.0 means that all interior nodes will split resulting in a completely balanced tree. An imbalance value of 1.0, on the other hand, produces the worst case of imbalance where the tree is slanted towards the right side with only two children at each level. Figure 4.11 shows the total nodes created for varying imbalance with $f = 2$.

Since the tree produced may differ in size from one run to another, it is important that a measure of the program's performance should take into account the size of the tree. Clearly, the simple execution time is inadequate. An alternative is to use the execution rate
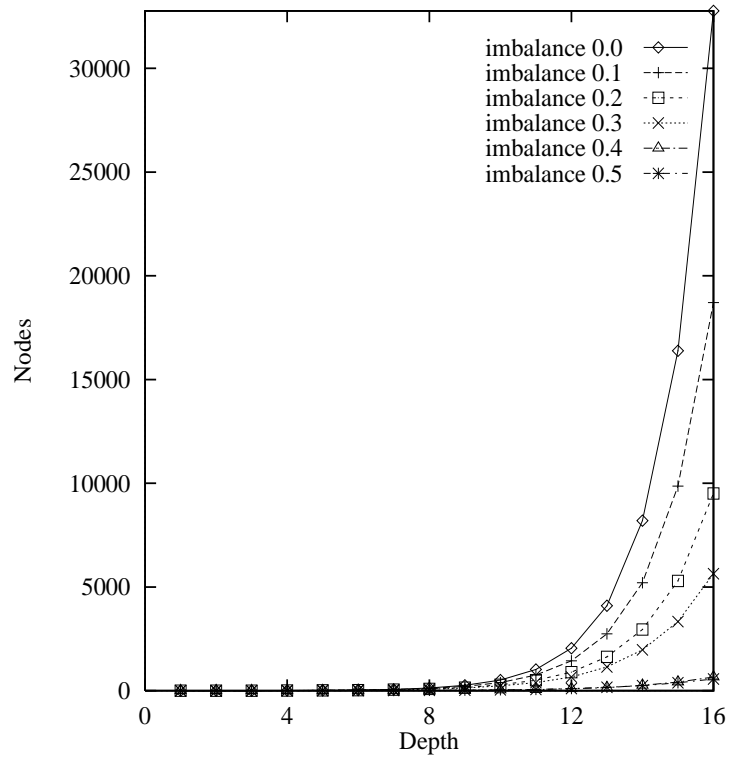
Figure 4.11: Total nodes created by the imbalance tree for varying degrees of imbalance in each depth.

Nodes	Time	Nodes/sec
36195	0.8306	43576.761
36695	0.8436	43497.746
37025	0.8499	43562.366
36695	0.8426	43548.770
36695	0.8426	43548.770
37025	0.8491	43603.493
37025	0.8495	43583.683
36075	0.8272	43609.150
37237	0.8545	43577.656
37025	0.8501	43553.843
1162	0.0273	111.404

Table 4.1: Variation of total nodes, time and speed (the last row being the difference between the largest and the smallest values).

(i.e. speed), that is the number of nodes executed per second which could normalise this difference. Table 4.1 shows a sample of 10 runs for the worst case of variation. Notice that the percentage difference for the total nodes is 3.12%, the total execution time variation is 3.2%, while the speed variation is only 0.26%. Our other experiments have shown that the speed variation is always less than 1%. Hence, the speed is a more appropriate measure to be adopted, since it has a normalising effect on the variation produced by the imbalance tree.

Most of the experiments in this thesis used the random tree because of its repeatability. The imbalance tree was used only when experimenting the effects of imbalance on the performance.

```
typedef struct {                        /* tree parameters       */

    int         tree_type,              /* balance or random     */
                max_depth,              /* maximum tree depth     */
                min_depth,              /* point to start splitting  */
                operation,              /* either stack or queue  */
                imbalance;              /* degree of tree imbalance  */
} tr, *tr_ptr;

typedef struct {                        /* individual subtree     */

    int         degree,                 /* degree of expansion    */
                num_tasks;              /* tasks created so far   */
    float       load;                   /* load created by the tree  */
    s_ptr       stack;                  /* stack of tree nodes    */
} tree, *tree_ptr;

tree        t[MAX_PROC];                /* array of subtrees      */
```

Figure 4.12: Data structures that support the tree computation.

As in the topology module, the tree computation is supported using two data structures, namely an array of sub-trees and tree parameters describing the characteristics of the tree concerned (Figure 4.12). The tree array corresponds to the array of processors, that is each element in the tree array represents the subtree created by the corresponding element in the processor array. A doubly linked-list was used to maintain the subtree. The linked-list can be used as a stack or queue using the standard stack or queue operations. This is to support the two basic tree traversal techniques, namely local depth-first and local breadth-first traversal. Local depth-first traversal processes nodes on any processor in

a last-in-first-out order of stack while local breadth-first uses the first-in-first-out order of queue. For simplicity we refer to 'local depth-first' as 'depth-first' and 'local breadth-first' as 'breadth-first' in the rest of this thesis.

In both techniques, the nodes nearer to the root are the candidates for migration; that is nodes at the bottom of the stack for depth-first and nodes at the the front of the queue in the case of breadth-first. Parallel depth-first execution tends to expand the right side of the tree (or higher siblings) first since these nodes are pop-ed out from the stack first. The reverse is true for parallel breadth-first.

Simulating the parallelisation of the tree computation is simpler than for load balancing. Local computation does not involve any other processors. Parallelisation is achieved by executing the node (or nodes of a sub-tree) of each processor in turn. Each node is retrieved from a local queue. If the node is a nonleaf node, it will be further divided. Otherwise the node will be discarded. In both cases, the node carries the same amount of floating point operations. If there is no node left, the simulator continues by 'executing' the subtree of the next processor. The same process is repeated for all processors.

4.3   Performance Modeling and Calibration

Modeling the performance of a BSP-like computation is simple. The time required to complete a superstep or an iteration is the time the last processor takes to complete its assigned tasks. The barrier synchronisation between iterations ensures that all processors start an iteration at the same time, and that processors that complete early sit idle waiting for the rest to finish.

If t_{ij} is the time processor j takes to complete its tasks during iteration i, the parallel runtime on p processors is

$$T_{par} = \sum_{i=1}^{I} \left[\max_{j=1}^{p}(t_{ij}) + T_{sy} \right]$$

where T_{sy} is the time to perform barrier synchronisation and I is the number of iterations.

The total cost of a processor in an iteration prior to executing the barrier is

$$t_{ij} = T_{cp}(ij) + T_{lb}(ij)$$

where T_{cp} is the time spent to execute tree computation, while T_{lb} is the time spent executing the load balancing operation. The total runtime cost model now becomes

$$T_{par} = \sum_{i=1}^{I} [\max_{j}^{p}(T_{cp}(ij) + T_{lb}(ij)) + T_{sy}]$$

In order to calculate the above execution time, an array of costs is used to keep track of the cost of each processor in each iteration. At the end of each iteration, all costs are set to the cost of the maximum processor. The total individual cost for compute, load balancing, synchronisation and idle time are also accumulated as the simulation proceeds. Idle time is the waiting time before executing the barrier. It does not include the time processors wait for work during load balancing.

The rest of this section describes the modeling of each of the components of the above formulation; compute, communication and the computation used in the load balancing and synchronisation.

4.3.1 Modeling the Cost of Computation

We assumed that the actual computation cost of executing a node, T_s, is equivalent to carrying out g floating point operations, for both the leaf and the non-leaf node;

$$T_s = g \times T_{fl}$$

This allows for experimentation on the effects of varying g. It is also possible to vary the node execution time to be proportional to data size, though, this has not been used in the experiments in Chapter 5.

The cost of executing a leaf node, T_l, is some house-keeping cost which is a constant value, T_c, and the cost of solving a single node;

$$T_l = T_c + T_s$$

The non-leaf node assumes the same cost as the leaf node, but with an additional cost of expanding f children. Therefore, the total cost of a non-leaf node, T_{nl}, can be written as:

$$T_{nl} = T_c + T_s + T_e \times f$$

The cost of a single node expansion, T_e, on the T3D, is 17.792 μsec, while that of the floating point operation, T_{fl}, is 0.172 μsec. Basic house-keeping for each node is 7.433 μsec.

Operation	Cost
Solve	$T_s = g \times T_{fl}$
Leaf	$T_l = T_c + T_s$
Non-leaf	$T_{nl} = T_c + T_s + T_e \times f$

Table 4.2: Computation cost models.

Table 4.2 summarises the cost models used for the tree computation.

4.3.2 Modeling the Cost of Communication

The simulator uses two point-to-point communication functions to support the load balancing operation. Both functions, *MPI_Send*() and *MPI_Recv*(), are blocking operations. Only one collective communication routine is used - the *MPI_Allreduce*(). The main function of the *Allreduce* is to collect the status of each processor and to distribute the total count, as well as acting as a barrier.

Howell in [35] proposed models to characterise the performance of MPI point-to-point and collective communications. The following two sections describe the fundamental idea behind the modeling of the two types of communications. Also discussed is the recalibration of the *MPI_Allreduce* model.

Modeling the Point-to-point Communication

In Howell's model, T_{send} is defined as the delay incurred on the sender when calling *MPI_Send*(), while T_{recv} is the delay on the receiver executing *MPI_Recv*(), when the sender and receiver start at the same time. If the message has already arrived before *MPI_Recv*() is called, the delay is $T_{recvmin}$. A delay on the receiver can be computed using the combination of these three models, that is, by taking the maximum of the relative timing when the sender posted the message, and the receiver initiated waiting for that message. In other words, the delay on the receiver is

$$receiver\_delay = max(R(t) + T\_recvmin, S(t) + T\_recv)$$

where $R(t)$ and $S(t)$ are the receiver and sender time respectively, prior to performing the communication. The simulator adopted this approach in costing the point-to-point communication. Note that the per-hop delay is not accounted for. Most current distributed-

memory machines, including the T3D, use 'worm-hole' routing techniques. Hence, the delay is negligible [5].

Another type of point-to-point communication used is the 'ping-pong' communication style. The delay on the sender is equivalent to the cost of sending a message and waiting for a reply from the receiver. This type of communication is common in a master-slave kind of communication. The minimum cost incurred on the slave for each access to the master is $T_{pingpong}$.

Table 4.3 shows Howell's cost models, which are used to support the two types of point-to-point communication. For a more accurate modeling, Howell differentiated the size of the message into two categories; small and large. A message which is 32 integer and below is regarded as small and uses the corresponding 'small' cost model; the reverse is the case for the larger messages.

Collective Communication

Initially the simulator was costed using Howell's *MPI_Allreduce* model below,

$$300 + 20 \times p + 0.9 \times log(p) \times data\_size$$

However, the results of the total execution time of the application showed a large overestimation compared to the actual measurement, but with a similar shape of curve. Since the total computational cost is proportional to the number of iterations, improving the *MPI_Allreduce* cost model was expected to pull the curve closer to the real measurement. Hence, the *MPI_Allreduce* model was recalibrated.

```
MPI_Barrier ()
start_t = MPI_Wtime ()

for (i = 0; i < MAX_ITER; i++)
    MPI_Allreduce ()

local_t = (MPI_Wtime () - start_t)/MAX_ITER
max_local_t = max_reduction ()
```

Figure 4.13: The code to measure MPI_Allreduce function.

The code used to time the *MPI_Allreduce* function is shown in Figure 4.13. The pro-

cesses are first synchronised using the *MPI_Barrier* (). The barrier only synchronises them logically, but does not time-synchronise. In other words, they may not start executing the **for** loop at the same time. The final time is obtained through a reduction as the maximum of all p average timing values, one from each processor. Statistical regression [1] was used to curve fit the actual measurements to Howell's general model of the *MPI_Allreduce* function;

$$a + b \times p + c \times log(p) \times dsize$$

where a, b and c are the coefficients that will give results close to the actual measurement and *dsize* is the length of the data in integer. The new coefficients for the improved model are shown in Table 4.4. The results of the new model are much closer to the real measurement when compared to Howell's model (see Figure 4.14).
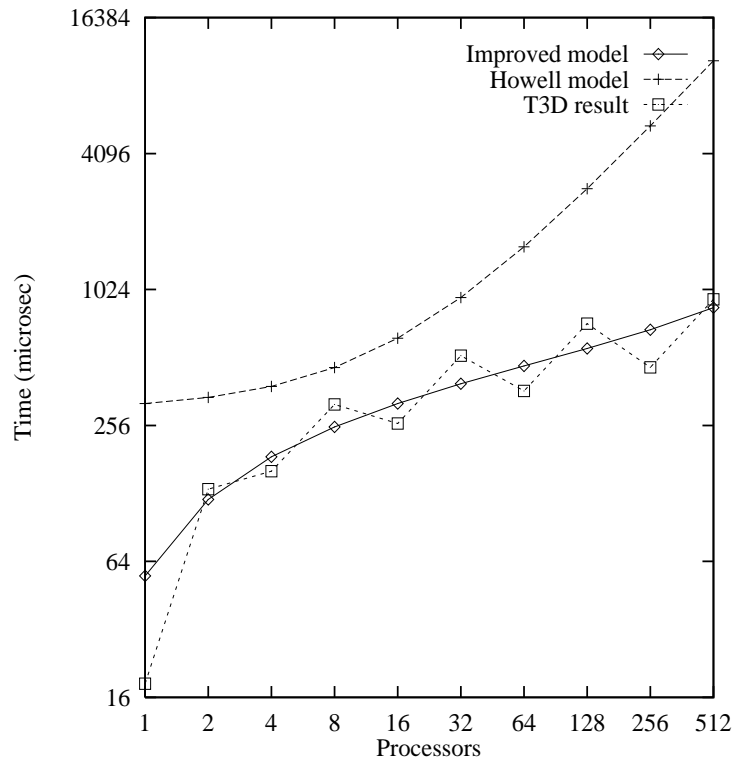

Figure 4.14: The original and the improved model with the T3D measurement.

Note that the timing of the *MPI_Allreduce* on the T3D in Figure 4.14 exhibits an anomaly, with a slower timing for the odd power of two processors (2, 8, 32, 128, 512). In these cases, a non-optimised algorithm is employed leading to strange timing results.

Operation	Small Message	Large Message
T_{send}	$70 + 3 \times dsize$	$100 + 0.09 \times dsize$
T_{recv}	$70 + 5 \times size$	$200 + 0.5 \times dsize$
$T_{recvmin}$	$50 + 3 \times dsize$	$100 + 0.4 \times dsize$
$T_{pingpong}$	$200 + 8 \times dsize$	

Table 4.3: Point-to-point cost models for small and large data.

Operation	Cost
$T_{allreduce}$	$54.8 + 0.42 \times p + 93.3 \times log(p) \times dsize$

Table 4.4: MPI_Allreduce cost model.

This calibration exercise has revealed the existence of a bug in the implementation of the function for MPI release 1.7a [31]. It is also interesting to note that the *MPI_Allreduce* performance on SP2 in [90] on page 18 demonstrates a similar pattern to the T3D's.

Tables 4.3 and 4.4 summarises the communication cost models used in the simulator, Howell's point-to-point communication models and our improved *MPI_Allreduce* model, respectively. $T_{allreduce}$ and $T_{pingpong}$ always use the 'small' cost models, since the reduction operation only involves a single integer flag (i.e. the status of each processor) and the message sent to (and received from) the centralised resource is also one integer in length.

4.3.3 Modeling the Cost of Load Balancing

In general, there are two aspects of modeling the cost of a load balancing algorithm, namely, computation and communication. Modeling computation includes modeling the enqueuing and dequeuing of data that involves in migration, while communication entails the load information acknowledgement and the actual migration of data. Some algorithms may require an additional "behaviour" to be modeled. In the case of the LDSV algorithm, the contention for service at the centralised Load Information Server was explicitly modeled. The modeling of this shared resource is discussed at the end of this section.

Modeling the Computation Cost of Load Balancing

The processing overheads can be modeled based on the number of enqueing or dequeing operations that occur in a single balancing operation since the processing time does depend on the number of occurrences of enqueueing/dequeueing. Each migration is a dequeue at

Operation	Sender (S)	Receiver (R)
Processing	$T_{pr} = T_{dq}$	$T_{pr} = T_{eq}$
Information Exchange	$T_{ie} = T_{send}$	$T_{ie} = max(R(t) + T\_recvmin, S(t) + T\_recv)$
Data Migration	$T_{dm} = T_{send}$	$T_{dm} = max(R(t) + T\_recvmin, S(t) + T\_recv)$

Table 4.5: Load balancing cost models.

the sender and an enqueue at the receiver. Hence, the processing time, T_{pr}, at the sender, is the total dequeue cost of the data to be migrated, while the cost at the receiver is the total enqueue cost of the received data (Table 4.5).

A single enqueue of a tree node, T_{eq}, is 4.015 μsec and a dequeue, T_{dq}, is 0.9 μsec. The enqueue is more expensive since it involves memory allocation and copying, while a dequeue only manipulates pointer variables.

This method does not account for execution of other load balancing codes such as migration decision or calculating the load level. The time taken for such operations is expected to be small. Furthermore, the computation cost of load balancing is assumed to be relatively small, anyway. However, this might not apply if more complex load balancing decisions are required.

Modeling the Communication Cost of Load Balancing

The source of communication overheads in load balancing usually originate from load information exchange and task migration. The message size for information exchange is usually small compared to tasks migration. The simulator used the same blocking point-to-point communication operations for both, that is *MPI_Send*() and *MPI_Recv*(). The method of costing these functions was discussed earlier in Section 4.3.2. Table 4.5 summarises the load balancing cost models used in the simulator. T_{ie} is the time for information exchange and T_{dm} represents data migration time.

Modeling the Cost of Contention in the Loadserver Algorithm

Loadserver algorithm uses a centralised server (LIS) to keep the information on the under-loaded nodes. This centralised server is a potential source of bottleneck. The contention at LIS occurs when 'heavy' processors compete for service (as discussed in Section 2.5). It can be modeled in three different ways:

(i) Optimistic case

No two (or more) processors access LIS at the same time. Let the round-trip time (i.e. the acknowledgement and receipt of reply), be T_{rt}, and the service time at LIS be T_{sv}. The total waiting time, T_w, of any heavy processor at any time is $T_w = T_{rt} + T_{sv}$.

(ii) Pessimistic case

This case assumes that each processor experiences contention during every access to LIS. Therefore, contention is proportional to p', which is the number of contending processors. Any reply is obtained after $T_{rt} + p'T_{sv}$ time.

(iii) Intermediate case

This case takes into account the number of contending processors and the possibility of delay in acknowledging 'heavy state' in the subsequent rounds.

Suppose all heavy processors, p', send messages during the first round and these messages are serviced in some sequential order;

– processor 1 receives its reply after $T_{rt} + T_{sv}$ since the time of its acknowledgement,

– processor 2 receives its reply after $T_{rt} + 2T_{sv}$, and

– processor p' receives its reply after $T_{rt} + p'T_{sv}$.

Assuming that some of them actually offload, they will continue sending 'heavy' messages in round 2. The messages will be send off staggered at intervals of T_{sv}, resulting in no message queuing at LIS. Hence, the heavy processors will be serviced in $T_{rt} + T_{sv}$ time (i.e. back to the best case for round 2 and the subsequent rounds).

In practise, the cost of assessing LIS is dominated by the communication cost, rather than the service at LIS. T_{rt} is the 'ping-pong' communication cost (see Table 4.3), which is 208 μsec for one integer message size while T_{sv} is only 0.775 μsec, the cost of dequeing one integer. Thus, we would not expect the different contention models to give significantly different results.

A series of experiments to evaluate the three cases of contention were carried out using a binary tree with depth 16 on a 2-d torus. The results suggest that contention at LIS is unlikely to have a major effect on performance (Table 4.6). This confirms our hypothesis.

p	Contention Model		
	Optimistic	Pessimistic	Intermediate
4	18.9396	18.9619	18.9460
8	11.9549	11.9923	11.9673
16	8.5543	8.5980	8.5726
32	6.8079	6.8553	6.8297
64	5.8911	5.9401	5.9146
128	5.4416	5.4914	5.4659
256	5.2429	5.2929	5.2678
512	5.2754	5.3252	5.3005

Table 4.6: The total execution time (in sec) for the three cases of contention at LIS.

4.3.4   Supporting Other Architectures

The main issue in simulating the performance of a parallel architecture is the performance model. With this in mind, the simulator was implemented to be flexible enough to support this purpose. Apart from simulating the T3D performance, it is able to simulate IBM-SP2 and a network of Sun Sparc workstations. The performance models for the last two can be found at [75] and [36], respectively. The general cost model of point-to-point communication for the three architectures remain the same. The only difference is the constant values which reflect the performance of the specific machine. For the *MPI_Allreduce*, the T3D and SP2 assume the same cost model (again with different constant values), since the architectures of both machines are similar; a distributed memory parallel machine. The cost model for the Sun workstations is rather different, since it involves a very different communication systems.

The IBM-SP2 machine is not at the disposal of the author for any validation or experimentation to be carried out on it. Hence, all the adaptivity experiments in this thesis employ the T3D performance model only.

4.4   Validation of the Simulator

If the simulator is to be used with confidence, it is essential that a validation process is conducted in order to confirm that the simulation results are sufficiently close to the performance of the real computations which are simulated.

The method used here is to validate the results obtained from the simulator with the actual performance results on the targeted machine in two stages:

p	Simulated	T3D
2	32768	32768
4	16385	16385
8	8194	8194
16	4099	4099
32	2053	2053
64	1031	1031
128	522	522
256	269	269
512	148	148

Table 4.7: Total iteration counts for GDEM.

- iteration count validation; and

- total cost validation.

For the purposes of validation, both algorithms, GDEM and LDSV, were implemented on the T3D. The implementation of the simulated and the real algorithms were both ensured to be very similar. Both algorithms used the same computational model, exactly the same tree module, and the same topology type. The load balancing module of the simulator used the same communication structure as the real implementation. The simulation results were validated against the T3D results in both stages. All validation was carried out on a 2-d torus, using binary tree of depth 16 ($2^{16} - 1$ nodes), and performing load balancing after every node execution.

4.4.1 Validation of the Iteration Counts with the Real Implementation

One of the advantages of the BSP computational style is that iteration counts are easily obtained. This assists in validation. The validation process started by comparing the total iteration counts for simulated and real algorithms. For the GDEM algorithm, the iteration results are exactly the same for all processor sizes (Table 4.7). This is due to the nature of the algorithm where balancing occurs between two isolated processor pairs. Furthermore, from our observation, the total communication and migration counts are also the same. We concluded that the simulator simulated the GDEM algorithm correctly.

Unlike the real GDEM, the real LDSV algorithm is less deterministic because the offloading of heavy processor(s) may occur in a different order. This different ordering may yield a slightly different iteration results in every run (as shown in Table 4.8). If the simulator went through the same execution route, for example balancing involved the same

p	Simulated	T3D		
		Run 1	Run 2	Run 3
2	65535	65535	65535	65535
4	21846	21846	21854	21852
8	9365	9366	9367	9365
16	4372	4372	4372	4372
32	2119	2185	2185	2185
64	1046	1046	1062	1062
128	523	527	527	527
256	264	266	266	266
512	137	140	140	145

Table 4.8: Total iterations counts for LDSV.

processor each time, then the simulator will produce the same results. For this reason, the simulated results are not always the same as the real one. However, there are occurrences where the two are identical (see Table 4.8). In the instances where the results differ, the difference is very small. On this basis, we concluded that the the LDSV algorithm was simulated correctly as well.

4.4.2   Validation of the Total Cost with the Real Implementation

Once the iteration counts were correct, the validation proceeded to the second stage – validating the cost model by comparing the real and the executed simulation times. Figures 4.15 and 4.16 show the performance of both algorithms for computation dominated and communication dominated application, with individual node grain size being 10000 and 100, respectively. There is a consistent overestimation of the simulation results for computation dominated problem. This is largely due to the non-linear cache performance [18] of the T3D. The simulator, on the other hand, simulates the floating point operation linearly by multiplying the number of operations with a single operation cost.

For computation dominated problem on 1 to 128 processors, the simulation results are within 15% accurate from the actual measurements (Table 4.9). Communication dominated problems, with the same processor range, achieved a lower accuracy, that is within 25% (Table 4.10). The projected times closely match the measured times for up to 128 processors.

For 256 and 512 processors, the projections deviate far from the measured results, within 40% and 60% respectively (with greater accuracy for the LDSV algorithm). This
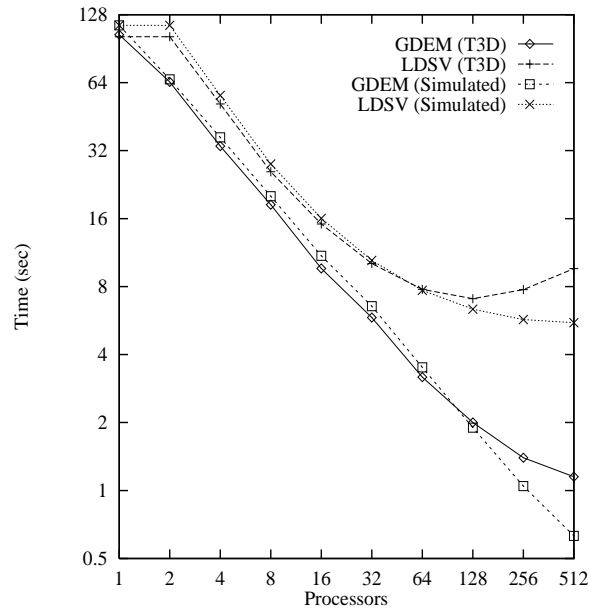
Figure 4.15: Computation dominated problem

p	GDEM (%)	LDSV (%)
1	89.9	87.7
2	97.3	87.7
4	90.6	90.9
8	91.1	91.7
16	86.3	94.1
32	87.6	96.7
64	89.6	98.9
128	94.8	89.9
256	74.9	73.7
512	54.7	71.3

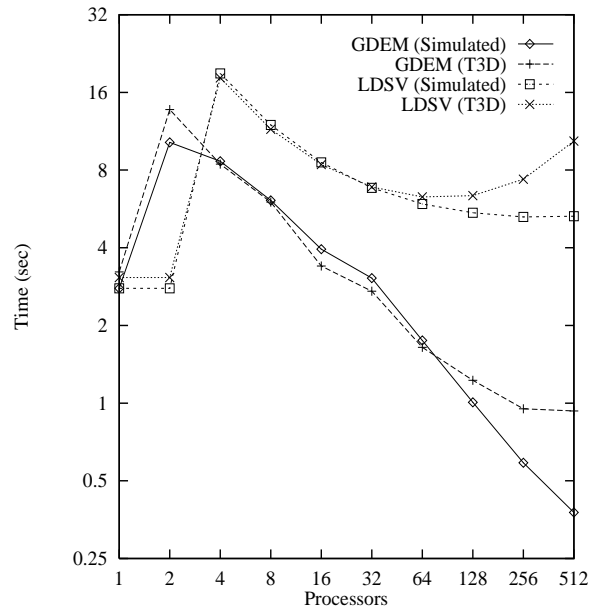Table 4.9: Percentage accuracy for computation dominated problem.

Figure 4.16: Communication dominated problem

p	GDEM (%)	LDSV (%)
1	86.4	90.8
2	74.6	90.8
4	97.5	95.9
8	98.5	96.1
16	83.8	98.0
32	87.7	99.3
64	93.7	93.9
128	82.2	85.8
256	61.8	71.5
512	40.4	51.1

Table 4.10: Percentage accuracy for communication dominated problem.

	Ring 16	16-ary 2-cube	Chain 8	Mesh 8 × 4
Xu & Lau	9.82	8.58	9.19	8.25
Simulator	10.66	8.6	9.6	8.6

Table 4.11: Average iterations for optimal λ ($\lambda_{opt} = 0.723$).

was anticipated because Howell's models were based on 32-processor experiments. Thus, the cost models are expected to be applicable for up to medium processor size. The results for large processors may perhaps be improved by using two different cost models, one for small and medium, and the other for large processor sizes. However, this requires remodeling of all point-to-point communication models, which is beyond the scope of this research. The results of 256 and 512 processors are treated with lesser confidence when compared to 1 to 128 processors.

To the best of the knowledge of the author, no reported work on MPI simulation extends up to 512 or even to 128 processors. Howell's simulation results achieved within a factor of two for communication dominated program and a factor of ten for computation dominated [35]. The maximum processor size used is 32. The MPISIM by Prakash [63] achieved within 20% of the target execution time. However, the processor range is up to 16 only. The two mentioned works (Howell's and Prakash's) are both full parallel simulation engine for MPI. The first is for T3D while the latter is for IBM-SP2.

4.4.3 Validation of GDEM with Published Results

Further validation is possible by comparing the simulation results with published results available in the literature.

Xu and Lau [88] carried out an experiment for a range of the workload exchange parameter, λ (i.e. $\lambda = 0.5$ to 0.95), for four different topologies and processor sizes; ring 16, 16-ary 2-cube, chain 8 and mesh 8 × 4. The results of each value of λ is an average total iterations of 100 runs. For a more detailed definition of λ, refer to Section 2.3.2.

We reproduced similar experiments using the same topologies and the same range of λ. We limited the number of runs to three since our purpose was simply to verify the correctness of the simulated GDEM. Similar patterns of average iteration counts, as tabulated on page 81 of [88] were produced for each lambda value. Using the optimal lambda, $\lambda_{opt} = 0.723$, for the four topologies, the simulator yielded very close results to Xu and Lau's (see Table 4.11). On the hypercube topology, the simulator produced a uniform load distribution in a single iteration from any initial workload distribution - a

distinctive characteristic of GDEM as discussed in [11].

The above two experiments act as a confirmation to the correctness and reliability of the simulator, as well as adding credence to the simulation results.

4.5   Limitations of the Simulator

The following are the limitations of the simulator:

- The simulator adopts a BSP-like model which is synchronous in nature. Any asynchronous load balancing algorithm (e.g. LDSV) has to be transformed to the corresponding synchronous version in order to suit the model. This has its toll on the performance, and surely does not reflect the actual potential of the asynchronous algorithm. In the synchronous version of the LDSV algorithm, facilities such as a broadcast to 'proceed' and a barrier have to be incorporated. These additional functions contribute to a slower execution time. A synchronous algorithm like GDEM, on the other hand, benefits from the BSP-like computational model.

- There is a limit in simulating parallelism using a sequential simulator. Algorithms which are more deterministic and whose balancing domain does not overlap with each other, such as LDSV and GDEM (especially the latter), have the advantage. The predicted results are reasonably close. The Diffusion algorithm is different. Balancing normally involves a subset of processors, and these subset of processors may overlap with one another. It may be difficult to avoid sequentialisation effect when simulating such behaviour using a sequential simulator. For this reason, GDEM is used instead of Diffusion in the subsequent experimental work.

4.6   Summary and Concluding Remarks

This chapter has described the design and implementation of the simulator for the purpose of evaluating phase-based adaptive dynamic load balancing algorithms for tree application. The simulator adopts the iterative BSP-style for simplicity. The main functional components of the simulator and the cost models were described. Extensive iteration and cost validation were performed on the simulator resulting in a close predicted results to the real execution times. The prediction time for both GDEM and LDSV algorithms for 1 to 128 processors is within 25% of the actual timing, and within 40% and 60%, for 256

and 512 processors, respectively. Although it is a sequential simulator, it is able to achieve high accuracy - as good as, or even better than, its parallel counterparts in the literature.

Thus, it is now possible to begin systematic performance experiments, with a high degree of confidence in the reliability of the simulation results. The next chapter takes up this task.

Chapter 5

Experimental Results

This chapter proceeds to describing the phase-based adaptivity experiments. It presents the results for parametric adaptivity techniques and establishes the relationship between the performance improvement and the individual node grain size, the network speed and the degree of tree imbalance, and discusses the potential algorithmic adaptivity results.

The chapter starts by presenting the results of an introductory experiment which aimed at verifying our earlier assumptions on the workload pattern of tree computation and determine the choice of the traversal method (in Section 5.1). This is then followed by Section 5.2 which explains the experimental methodology and the organisation of phase-based adaptivity experiments. Section 5.3 presents the results of the parametric approach and the relationships between the performance gained and the selected application and machine parameters. Section 5.4 discusses the results of the algorithmic approach. Finally, Section 5.5 concludes with a summary of results.

5.1 Preliminary Experiments

This section begins by discussing two preliminary experiments which help verify the assumptions made on the workload pattern, and raises the issue of the sensitivity of the DLB interval. One conclusion of the experiments is to consider only depth-first traversal for the later adaptivity experiments.

5.1.1 Verifying the Workload Pattern

Recall in Chapter 3 the workload pattern is the shape of the curve formed from the start until the end of a tree computation. The curve reflects the total workload with respect to

time (see Figure 3.1). It is an idealised workload pattern under the assumption that the workload grows smoothly in a 'bell-shape' curve. In practise, this may not be precisely true.

Therefore, we set out an experiment to verify our assumption by studying the effects of the traversal methods on the workload pattern using the two different DLB algorithms, namely GDEM and LDSV, on three different processor sizes.

The tree type and the tree parameter settings were set to be the same throughout; random tree with depth, $d = 16$, and fan-out, $f = 2$. The same applies to DLB parameters, where the interval, i, is 16. The grain size, g, of each node is 100, though g is unimportant in this context since we are only interested in the node count and not the execution time.

The results are as follows. Breadth-first (BF) traversal generates workload pattern close to a 'bell-shape' curve for both algorithms for most processor sizes (see Figures 5.1 and 5.2). The exception is with $p = 128$ for LDSV algorithm which exhibits a combination of two bell-curves. Depth-first (DF) follows a similar workload pattern, but the curve is 'flatter' and more 'jagged' (see Figures 5.3 to 5.6).

The smoother curve of BF is due to the early expansion of nodes near the top of the tree. Nodes higher up the tree have higher probability of expansion compared to those near the bottom. This leads to a steady increase in the total workload at the beginning of the execution, that is when the top part of the tree is being executed. This is then followed by a steady decline of the total load as the computation goes to the bottom, generating a smooth 'bell-shape' curve.

DF produces a different effects because the execution moves right to the bottom of the tree. Some of the leaf nodes are executed first before the nodes near the top of the tree. When the leaf nodes are executed, there is no new node generated. The total number of nodes drops creating a rather 'jagged' curves. This is more prominent for smaller p since the number of nodes expanded is relatively smaller in comparison to the number of nodes consumed. For the same reason the total maximum number of nodes generated by DF is always smaller in comparison with BF for the same DLB algorithm and processor size.

Note that the total processors that actively carries out the computation for LDSV algorithm is p - 1. One processor is reserved as the load information server. Hence, the total workload grows more slowly than GDEM.

The workload pattern produced by BF and DF traversal confirms the general validity of our assumption. However, the relatively more 'jagged' workload pattern of DF is likely to make detecting transition points more complex than the relatively monotonic workload of BF.
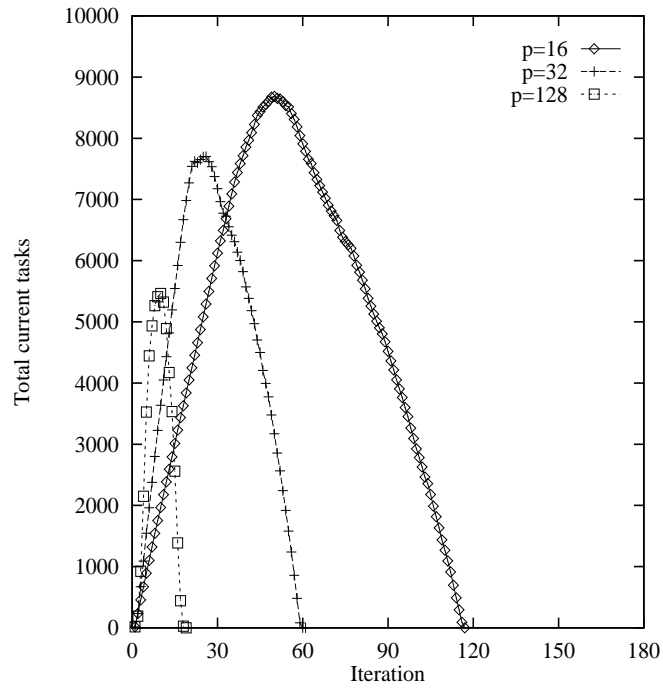
Figure 5.1: The workload pattern produced by GDEM using breadth-first traversal.
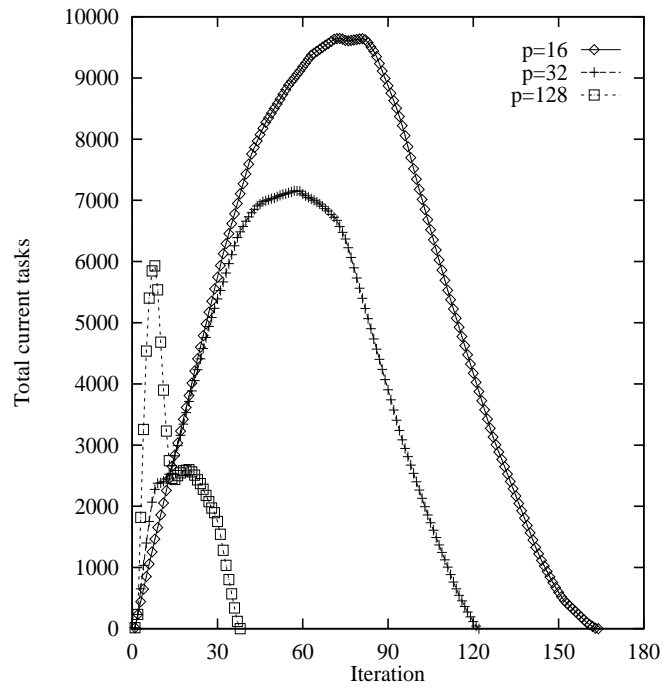

Figure 5.2: The workload pattern produced by LDSV using breadth-first traversal.
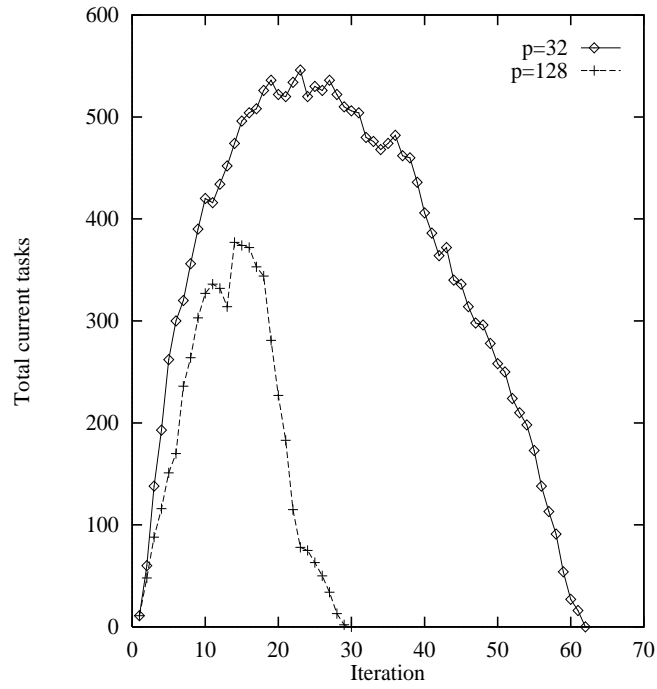
Figure 5.3: The workload pattern produced by GDEM using depth-first traversal for large and medium processor sizes.
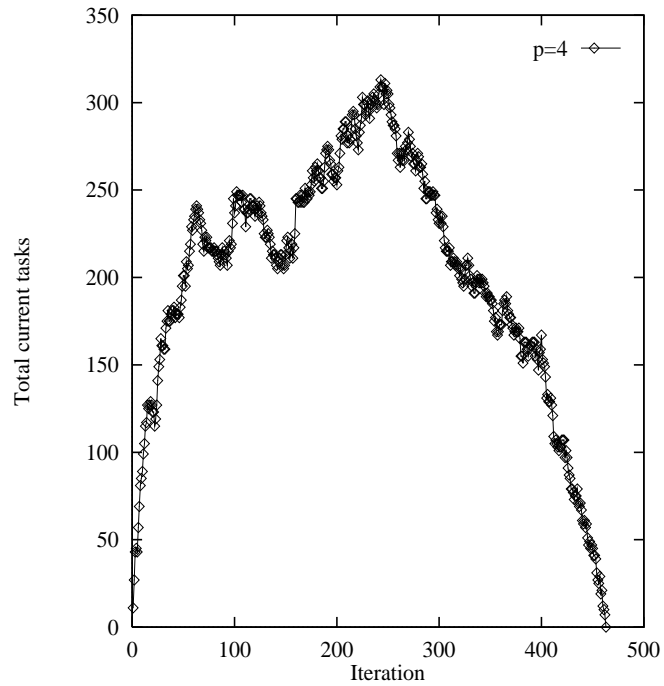

Figure 5.4: The workload pattern produced by GDEM using depth-first traversal for small processor size.
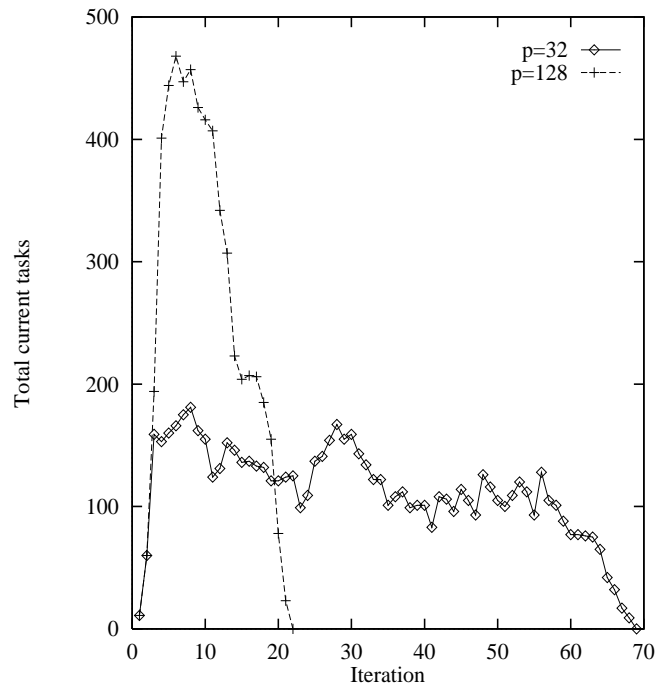
Figure 5.5: The workload pattern produced by LDSV using depth-first traversal for large processor sizes.
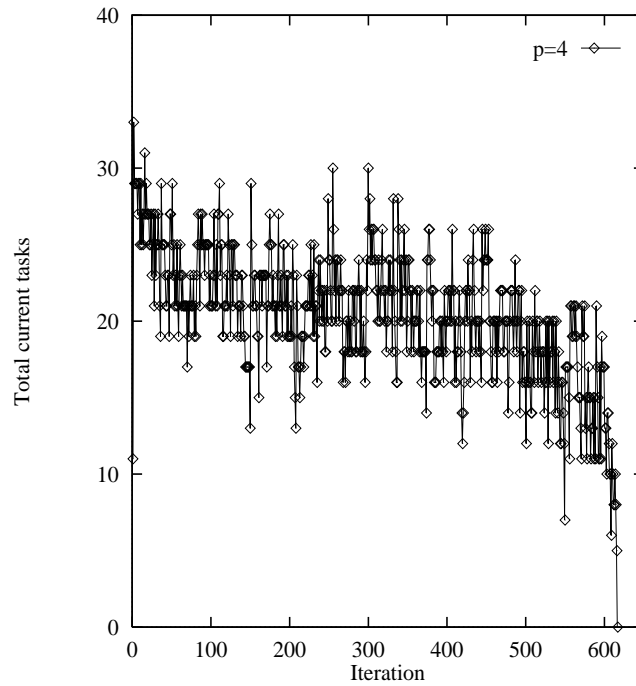
Figure 5.6: The workload pattern produced by LDSV using depth-first traversal for small processor size.

5.1.2   Sensitivity of the Execution Time to DLB Interval

Our preliminary experiments also revealed that the performance of a tree computation is very sensitive to DLB interval (see Figures 5.7 to 5.12). This is true for both DLB algorithms. For this reason we chose to first investigate the effect of adapting the interval (instead of other DLB parameters) for our parametric adaptivity technique. The sensitiveness also means that finding the best interval for each phase is important. Section 5.2.2 continues the discussion on the load balancing interval.

Figures 5.7 to 5.9 show the performance of GDEM and LDSV using BF traversal. Notice that the performance of LDSV is more sensitive to i. A slightly different interval (from the optimal) may lead to poor performance. We can give no adequate explanation for such behaviour. When DF is used the performance gets more stable; see Figures 5.10 to 5.12. A wider range of i yields results which are reasonably close to the best performance. This degree of sensitiveness has an influence on the choice of the traversal method. GDEM, on the other hand, seem not to be significantly affected by the different methods of traversal. Both algorithms perform better using DF traversal; there is a slight improvement for GDEM and a greater improvement for LDSV.

5.1.3   The Choice of the Traversal Method

We recognise that BF produces a workload pattern which is closer to our idealised conceptual diagram which makes phase detection easier. Nevertheless, we have chosen to use DF in future experiments. We outline the reasons for our choice:

- LDSV performance is very sensitive to the interval when BF is used and shows a stable performance with DF. GDEM does not show sensitivity problem with either method. In short, both algorithms seems to be stable with DF traversal. Furthermore, the total execution time for both algorithms are better for DF and this is more apparent for the LDSV algorithm.

- DF traversal is widely use in tree search algorithms [44, 45, 67]. Using it means that we are following the common practice.

- DF consumes less memory space [53]. Moreover, our experience have shown that obtaining simulation results using DF take a shorter time.

All the subsequent adaptivity experiments therefore use DF traversal method.
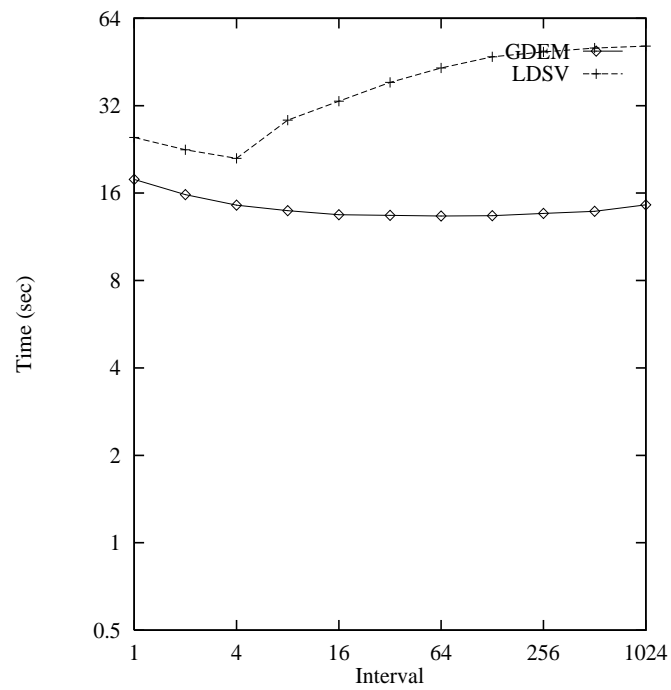
Figure 5.7: The sensitivity of i for breadth-first traversal ($p = 4$).
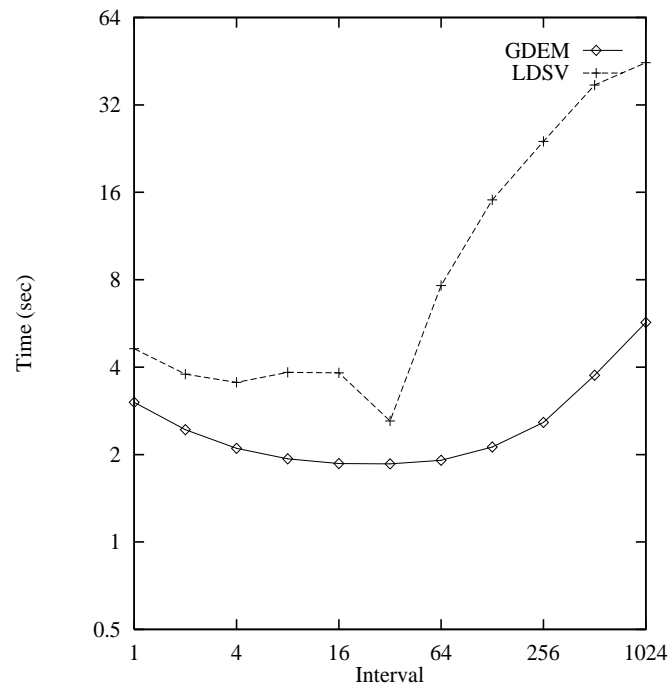
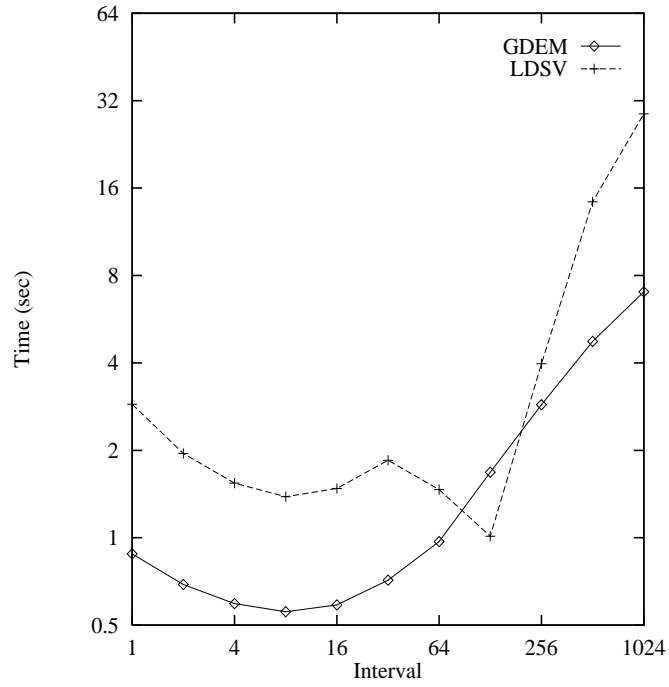Figure 5.8: The sensitivity of i for breadth-first traversal ($p = 32$).

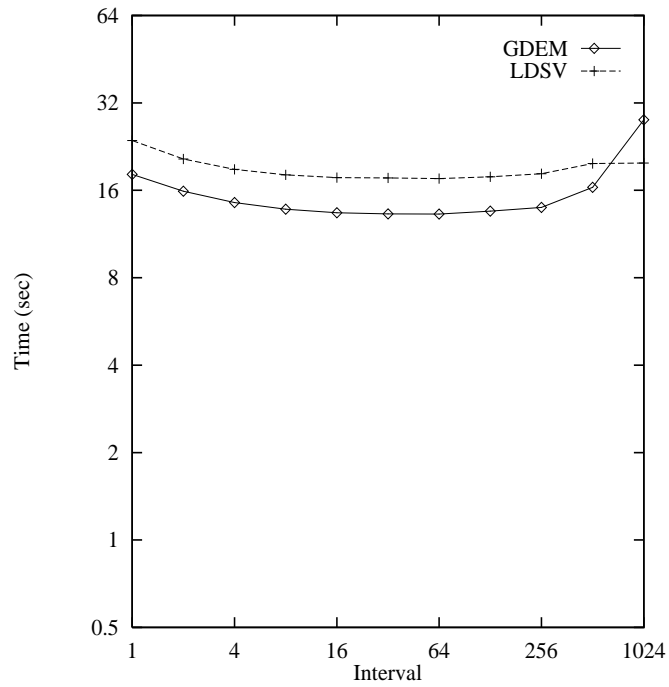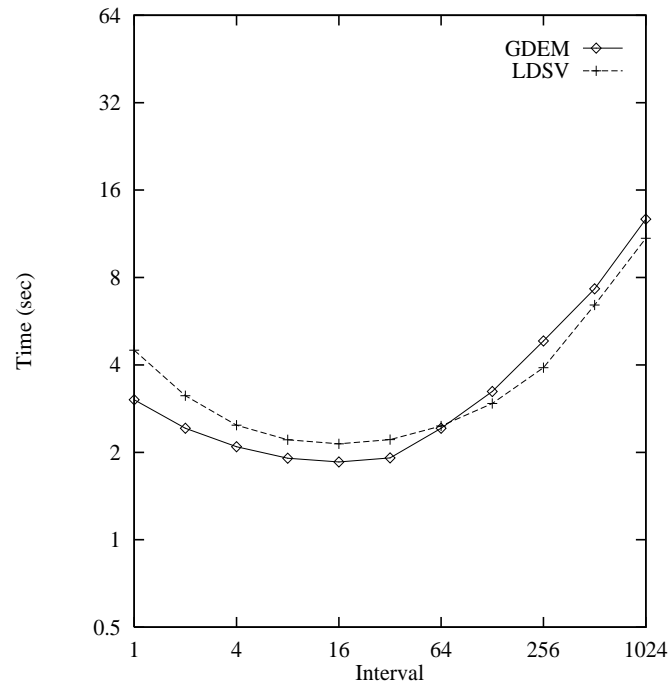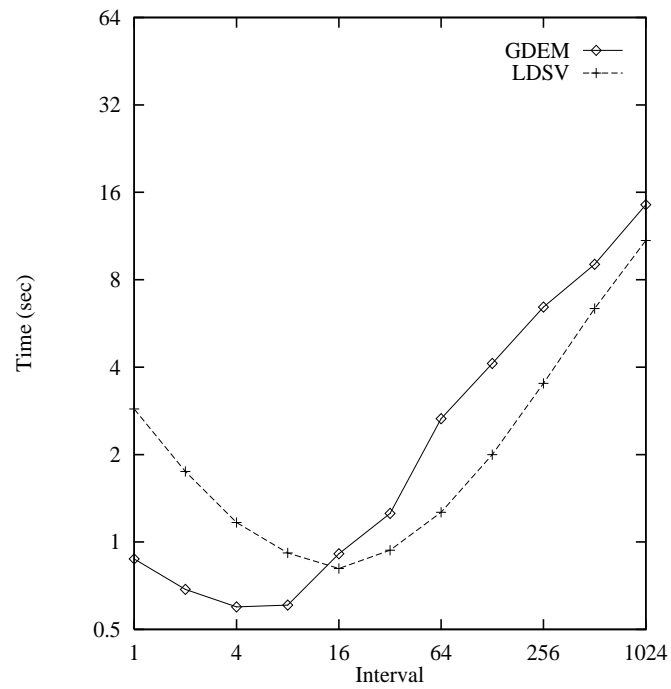Figure 5.9: The sensitivity of i for breadth-first traversal ($p = 128$).

Figure 5.10: The sensitivity of i for depth-first traversal ($p = 4$).

Figure 5.11: The sensitivity of i for depth-first traversal ($p = 32$).

Figure 5.12: The sensitivity of i for depth-first traversal ($p = 128$).

5.2 Experimental Methodology

This section starts by defining the metric used to assess the phase-based adaptive technique. This is then followed by a discussion on the load balancing interval and the description of parametric and algorithmic experiments. The values of the parameters used in the experiments are described at the end of the section.

The performance of a parallel tree application depends on many parameters such as machine, application and DLB parameters. The DLB parameters can be adapted according to the workload phases to improve performance (as suggested in Chapter 3). In order to gain insight on the effect of the individual parameter on the performance it is necessary to fix all other parameters and varies only the one which is under investigation.

Since our preliminary results have shown that the performance is very sensitive to i we decided to experiment the interval first instead of other DLB parameters (e.g. threshold and migration factor). This by no means implies other DLB parameters are not important neither indicates i as the most important. Only the experiments which are related to i were carried out and discussed in this thesis. All other experiments (e.g. threshold and migration factor) are identified as future work (detailed discussion can be found in Section 6.3).

5.2.1 Performance Metric

We introduce a metric called Improvement Through Adaptivity (ITA) to measure the improvement gained from our proposed technique. ITA measures the percentage improvement of adaptive over a non-adaptive approach;

$$ITA = \frac{T_{NA} - T_A}{T_{NA}} \times 100,$$

where T_{NA} and T_A were defined in Section 3.2. In the above case ITA refers to the total execution time.

In Chapter 4 we have described that a suitable metric for imbalance tree would be the rate of the execution (or speed) which normalised the different tree sizes. Speed is defined as the total number of nodes over time taken to execute those amount of nodes, either in adaptive or non-adaptive environment. For such cases ITA is defined as follows:

$$ITA = \frac{S_A - S_{NA}}{S_{NA}} \times 100,$$

where S_{NA} and S_A refer to the total speed in a non-adaptive and adaptive environment, respectively. For parametric adaptivity ITA refers to the total execution time or the number of nodes executed per second (i.e. speed). Detailed discussion is incorporated (in the relevant section) whenever speed is used.

For algorithmic adaptivity the improvement should be measured relative to the best execution time between the two individual algorithms.

In addition to ITA we also provide (when necessary) detailed measurements of performance, such as idle time, synchronisation time, load balancing time and computation time.

5.2.2 Load Balancing Intervals

Recall from Chapter 3 that there are three workload phases: filling, steady and emptying. No load balancing is required for the emptying phase. Hence, we need to determine only two load balancing intervals, i_1 and i_2, for the first and the second phase, respectively.

Since the objective of the load balancing during phase I is to quickly fill the machine, the most reasonable value for i_1 is one. This means that the load balancing is invoked after every node expansion. Hence, the likelihood of the machine being full is high. The total idle time processor waiting for work could be greatly reduced.

For i_2, we choose as a suitable value the best interval when carrying out the whole computation in a non-adaptive environment. The reason is that phase II usually dominates a computation. A single i_2 is used for the three processor sizes in both adaptive and non-adaptive runs. The effect of a non-optimal value of i_2 is belief to be similar in both cases, hence we do not anticipate any major effect on ITA. The general pattern of ITA is anticipated to be the same, whether a single i_2 or a different i_2 are used for all three processor sizes.

Detailed discussion on the method to determine suitable i_2 is given in Section 5.3.1.

5.2.3 Parametric Adaptivity Experiment

We have explained the motivation of choosing the interval in investigating the benefit of adaptivity in Section 5.1.2. This section continues by describing the motivation of adapting the interval in relation to one machine and two application parameters. We then outline the sub-experiments involved.

The computational granularity of a node may vary from one application to another. This motivates us to investigate the effect of varying g. Some of the tree applications

may have different degree of tree imbalance which produces irregular workload throughout the computation. Dynamic load balancing is more useful for applications with an unpredictable workload generation. Hence, we chose to study the effect of the tree imbalance, m. Another parameter of interest is the network speed, s, which is relative to the communication bandwidth of the T3D. Experimenting how the performance are affected by different network performance would be beneficial.

For parameters g, s and m we carried out experiments to investigate how ITA varies with the given parameter, keeping all other parameters fixed. These experiments were carried out at three different processor sizes representing small, medium and large networks, and for both GDEM and LDSV.

Thus for each of the parameters we carried out the following experiments:

(i) Determine the value of i_2.

(ii) Determine ITA for a range of values of the parameters in the following cases:

    (a) activating transition I only,

    (b) activating transition II only and

    (c) activating transition I and II.

From the results, the most beneficial forms of adaptivity are analysed.

5.2.4   Algorithmic Adaptivity

Studying algorithmic adaptivity requires the use of different DLB algorithms at different stages of computation. Global information and migration characteristics of LDSV facilitate work distribution so as to quickly fill the machine during phase I. GDEM's local information and migration characteristics help maintain the steady state of phase II. During the emptying phase, effort in load balancing may no longer brings any benefit. Hence, the load balancing is discontinued.

Because of the relatively poor performance of LDSV (due to artificial synchronisation), it is not sensible to carry out these experiments at this stage. Further discussion is given in Section 5.4.

5.2.5   Parameter Settings

As mentioned in Section 3.2 there are three categories of parameters that need to be considered when running the simulator; machine, application and load balancing parame-

ters. For each category we reason out the choice of the values made. We also include the discussion on the phase transition parameters.

The values of the parameters are described below:

- Machine parameters.

  We divided the processor sizes, p, into three main groups; small (2, 4, 8), medium (16, 32, 64) and large (128, 256, 512). The middle figure of each group is used except for large p where the simulation result for $p = 256$ is less accurate when compared to $p = 128$ (as discussed in Section 4.4.2), so we chose $p = 128$, instead. All experiments assumed T3D performance characteristics, with the exception of the experiments which aim at investigating the effect of network performance.

- Application parameters.

  All experiments (except those related to experimenting with the degree of tree imbalance) use a random tree with $f = 2$, $d = 16$ and zero minimum depth of splitting (which is explained in Section 4.2.3) producing a total of 29535 nodes, an amount of workload which is enough to benefit from parallelism. The repeatable characteristic of random tree enables easy comparison to be made (as opposed to the imbalance tree). For experiments which do not involve varying the grain size we wished to use a value of g which was neither excessively coarse-grain nor excessively fine-grain. An appropriate intermediate value was considered to be $g = 100$.

- Load balancing parameters.

  We recognise that there is scope for experimenting the combination of different values of DLB parameters (as discussed in Section 3.1.3). However, in order to assess the effects of adapting the interval we use the same values of h, l and r (as used in the original algorithms) throughout the experiments. For the LDSV, the values are: $h = 1$, $l = 0$ and $r = 1$. The high and low workload threshold is not applicable for GDEM while $r = \lambda_{opt}$ (recall that $\lambda_{opt} = 1/(1 + sin(2\pi/\mathrm{k}))$ for 2-d torus as described in Section 2.3.2).

- Transition parameters.

  We proceed by discussing two important parameters, c_1 and c_2, which determine the two transition points, t_1 and t_2, respectively. Recall that the condition for t_1 is met when the number of tasks reaches $c_1 \times p$ while t_2 is when the number of tasks drops below $c_2 \times p$.

p	c		
	0.5	1.0	1.5
8	6.6772	**6.6400**	6.6453
32	1.7261	**1.7082**	**1.7082**
128	0.5061	**0.5056**	0.5085

Table 5.1: Total execution time (sec) with varying c_1 for GDEM.

We carried out an experiment to determine the optimal value for c_1. Our results showed that the best results is achieved (most of the time) when $c_1 = 1$. Table 5.1 presents a sample of such results. The highlighted values are the best execution time and are the results of $c_1 = 1$. This means that the best transition point is when $t_1 = p$.

As for t_2, we pointed out earlier that the 'jaggedness' of DF traversal has made t_2 difficult to be detected. We acknowledge that there is a room for further research in this area. However, we have chosen to set t_2 to a fix value (again to reduce the number of experimentations) — a value which does guarantees a consistent decrease in total workload after the cut-off point. This value may or may not lead to the best switch off point.

From our observation, there is always a smooth drop in total workload at the end of the computation (as previously shown in Figure 5.3 to 5.6), making the transition detection more straight forward. Choosing $c_2 = 1$ ensures that the transition occurs within this 'safe' region, that is outside the 'jaggedness' area. Clearly there is a case for a larger value of c_2, as noted in Section 3.2, but the 'jagged' effects would make transition detection more complex.

Thus, the values of c_1 and c_2 were both set to 1 throughout the main experiments. Section 5.3.1 briefly investigate the effect of increasing c_2.

We started our experiments on a topology with 'intermediate' connectivity (i.e. torus 2-d). It remains to be discovered to what extent the results are affected by a lower or higher connectivity, such as ring and hypercube.

5.3 The Results of Parametric Adaptivity

This section presents the results of the improvement through adaptivity for each of the parameter under investigation, that is g, s and m, when the interval is adapted using t_1 alone, t_2 alone and $t_1 t_2$. The best technique and improvement are also identified.

5.3.1 Varying the Computational Grain Size

The purpose of this experiment is to investigate the performance improvement through adaptivity for a range of grain size, g. The values for g can either be 10, 100, 1000 or 10000 floating point operations.

Determining i_2

In order to determine i_2, we first varied i in a wide range, from 1 to 1024, for each value of p and g for each algorithm. Figures 5.13 to 5.15 illustrate the performance of GDEM with varying i. We then chose the interval which yields the best execution time for each case. Table 5.2 shows the complete results for the best interval, i_b, for each p and g.

The following observations can be made from Table 5.2:

- i_b decreases with g.

- i_b also decreases with p.

- GDEM has a smaller i_b if compared to LDSV.

GDEM incurs less load balancing overhead in each iteration when compared to LDSV. Therefore, the interval is expected to be smaller. These different values means that the best interval is algorithm dependent and there is no single best interval for all DLB algorithms.

The decrease of i_b with g is expected because when g increases the computational granularity between two successive iterations will increase. Hence, i_b, which is the number of nodes executed between iterations must decrease (due to the increase in grain size of the individual node). i_b decreases with p to quickly distribute the nodes throughout the machine – reducing the idle time waiting for work. The effect is significant for large p.

Although i_b decreases with p, for simplicity and practicality we decided to fix the value of i_2. This is also true when carrying out the experiments for s and m. We do not expect a fixed value of i_2 to give a major effect on the ITA since the same values of i_2 are used for the adaptive and non-adaptive version. This means that the same amount of discrepancy
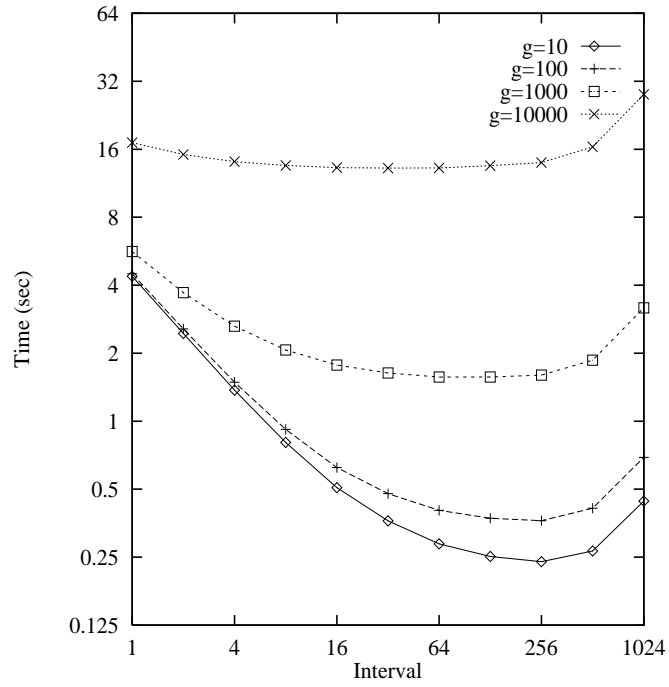
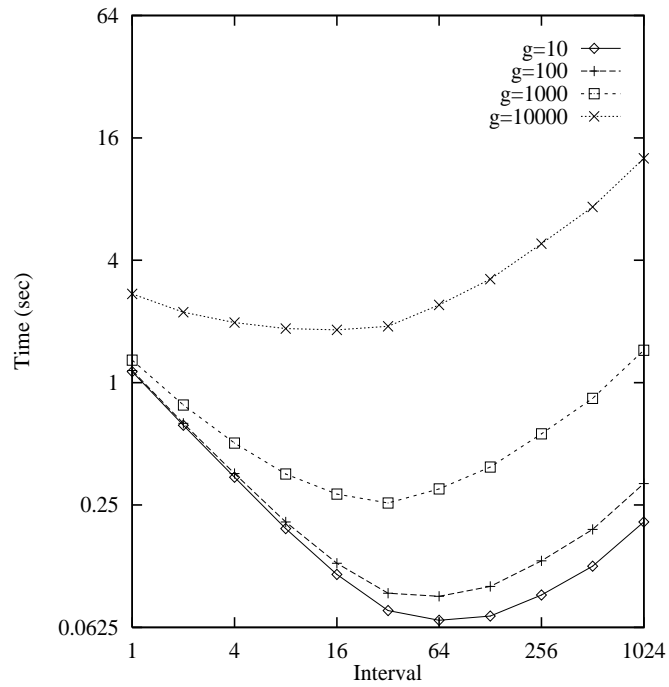Figure 5.13: Grain size: The effects of varying i for GDEM ($p = 4$).

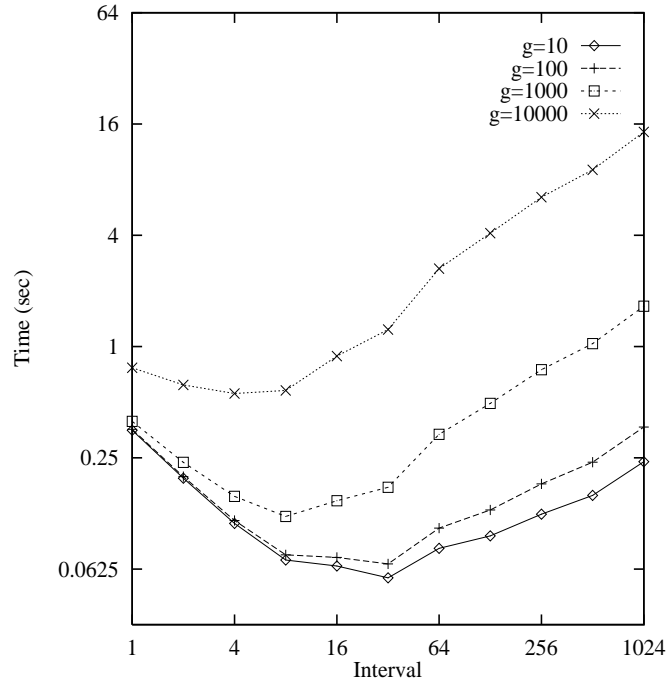Figure 5.14: Grain size: The effects of varying i for GDEM ($p = 32$).

Figure 5.15: Grain size: The effects of varying i for GDEM ($p = 128$).

p	g	Best interval (i_b)	
		GDEM	LDSV
4	10	256	256
	100	256	256
	1000	64	128
	10000	32	64
32	10	64	256
	100	64	128
	1000	32	64
	10000	16	16
128	10	32	256
	100	32	128
	1000	8	32
	10000	4	16

Table 5.2: Grain size: The best interval for a range of p and g.

may effect both cases, which leads to the same ITA value. We leave the task of establishing the relationship between i_b and p as future work.

We now discuss the method used to determine i_2 for GDEM. We calculated the discrepancy in performance between a selected i and the best i. We then chose the value of i which yield the smallest discrepancy over a range of p and g. The value of i_2 for GDEM is 16 with the worst discrepancies of 52.99% when $p = 4$ and $g = 10$ (see Figure 5.13). The same method is repeated for LDSV. The value of i_2 is 64 with the worst discrepancy of 36.66% when $p = 128$ and $g = 10000$.

Transition I Only

Table 5.3 shows the improvement when t_1 is used while Figure 5.16 depicts the actual graphs of the raw performance from which the results for $p = 128$ in Table 5.3 were derived.

The following observations can be made:

- In all cases ITA always increases with g.

- Both GDEM and LDSV can significantly benefit (in some circumstances) from using t_1.

- Overall, GDEM benefits from t_1 more than LDSV.

- Positive ITA for all g occurs for GDEM when $p = 128$, and for LDSV when $p = 32$.

- Negative values of ITA are mainly obtained for small g.

- Negative values of ITA for all g are obtained for LDSV when $p = 4$.

Discussion

The key idea in using t_1 is to invoke the load balancing frequently during phase I to facilitate work migration to the whole system in order to minimised the idle time waiting for work. This frequent invocation means that the number of iterations will increase and possibly the cost of load balancing increases as well. For this reason an algorithm which does not have a fast work distribution facility (e.g. GDEM on 2-d torus) is expected to benefit more from the technique. The technique is deemed useful for large networks. Another point to be anticipated is large improvement occurs for large g since the idle time waiting for work is greatly reduced.

p	g	ITA (%)	
		GDEM	LDSV
4	10	-0.13	-0.79
	100	-0.08	-0.62
	1000	0.06	-0.31
	10000	0.12	-0.21
32	10	-0.63	7.16
	100	-0.19	8.68
	1000	1.56	14.23
	10000	2.80	17.29
128	10	14.24	-13.03
	100	15.49	-9.29
	1000	21.02	15.22
	10000	25.52	40.96

Table 5.3: Grain size: Improvement through adaptivity using t_1 only.

Tables 5.4 and 5.5 show detailed measurements for GDEM and LDSV, respectively, when $p = 128$. Recall that T_{NA} and T_A refers to the total execution time for non-adaptive and adaptive runs. There is a consistent reduction of load balancing time and idle time for GDEM as g increases. Notice that for GDEM algorithm, the total number of iterations does not increase (as would be expected when t_1 is used). This contributes further to the larger benefit of t_1 for GDEM on 128 processors. In the case of LDSV, t_1 always increases idle time, while the cost of load balancing may increase or decrease. For cases where the load balancing time decreases, ITA is positive.

Both algorithms show little (or negative) improvement for small p. This is anticipated because the idea of rapidly distributing work is more important for larger networks.

As a summary, GDEM usually gives greater benefits of using t_1. LDSV already has a good load distribution capability, hence, it is expected not to benefit from the technique as much. Using t_1 decreases the idle time for GDEM but increases for the LDSV. For both algorithms, if the use of t_1 causes a decrease in the load balancing time (and a decrease in idle time too in the case of GDEM), and the amount exceeds the increase in synchronisation time, then there will be a benefit. The technique is expected to be profitable for large p which normally suffers from a long waiting time for work.

g	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
10	14.24	T_{NA}	0.0648	0.0062	0.0296	0.0122	0.0168	30
		T_A	0.0556	0.0062	0.0255	0.0071	0.0168	30
100	15.49	T_{NA}	0.0722	0.0098	0.0324	0.0133	0.0168	30
		T_A	0.0611	0.0098	0.0267	0.0078	0.0168	30
1000	21.02	T_{NA}	0.1463	0.0456	0.0598	0.0241	0.0168	30
		T_A	0.1155	0.0456	0.0384	0.0146	0.0168	30
10000	25.52	T_{NA}	0.8868	0.4038	0.3339	0.1323	0.0168	30
		T_A	0.6605	0.4038	0.1562	0.0835	0.0168	30

Table 5.4: Grain size: Detailed cost of GDEM using t_1 ($p = 128$).

g	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
10	-13.03	T_{NA}	0.1312	0.0063	0.0676	0.0512	0.0062	11
	-13.03	T_A	0.1483	0.0063	0.0688	0.0654	0.0078	14
100	-9.29	T_{NA}	0.1394	0.0099	0.0724	0.0509	0.0062	11
	-9.29	T_A	0.1524	0.0099	0.0693	0.0653	0.0078	14
1000	15.22	T_{NA}	0.2321	0.0460	0.1256	0.0544	0.0062	11
	15.22	T_A	0.1968	0.0460	0.0763	0.0666	0.0078	14
10000	40.96	T_{NA}	1.2642	0.4070	0.6774	0.1736	0.0062	11
	40.96	T_A	0.7464	0.4070	0.1667	0.1648	0.0078	14

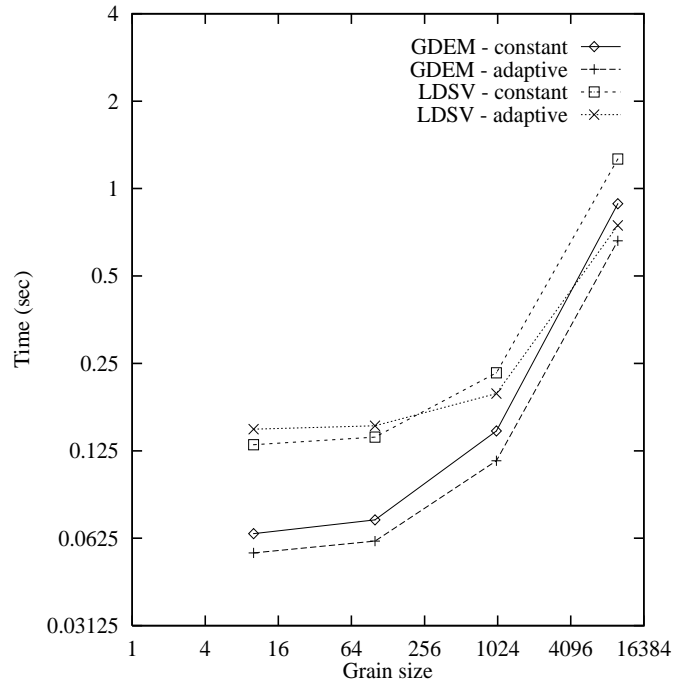Table 5.5: Grain size: Detailed cost of LDSV using t_1 ($p = 128$).

Figure 5.16: Grain size: Improvement through adaptivity using t_1 only for both algorithms for all g ($p = 128$).

Transition II Only

Table 5.6 shows the benefit of using t_2 only. The following observations can be made:

- In all cases ITA decreases with g.

- Both GDEM and LDSV can benefit (in some circumstances) from t_2, though the benefit is less than t_1.

- LDSV benefits more from t_2 than GDEM.

- There is zero ITA for $p = 4$ for both algorithms.

- Positive ITA for all g occurs for LDSV when $p = 128$, and for GDEM when $p = 32$.

- Negative ITA for all g occurs for GDEM when $p = 128$.

Discussion

The main idea in using t_2 is to reduce the cost of load balancing towards the end of a computation when there is lesser number of work than the number of processors. The benefit comes from a reduction of the load balancing and also synchronisation cost. The

p	g	ITA (%)	
		GDEM	LDSV
4	10	0.00	0.00
	100	0.00	0.00
	1000	0.00	0.00
	10000	0.00	0.00
32	10	1.55	3.18
	100	1.37	1.89
	1000	0.63	-0.11
	10000	0.10	-0.52
128	10	-26.19	14.43
	100	-46.20	13.15
	1000	-134.87	5.36
	10000	-207.11	-2.48

Table 5.6: Grain size: Improvement through adaptivity using t_2 only.

trade-off is an increase in idle time. Large grain size node has the potential of inducing a larger load imbalance, hence larger idle time.

LDSV exhibits a better improvement towards switching off the load balancing; positive improvement occurs for almost all g when $p = 32$ and $p = 128$ (Table 5.6). GDEM, on the other hand, shows an improvement for medium p only and with only a small percentage.

Tables 5.7 and 5.8 show the detailed measurement for both algorithms when t_2 is used. The large improvement benefited by LDSV comes from the reduction of the three sources of overhead; the synchronisation, load balancing and idle time. Notice that when $g = 10000$ the improvement is less. This is due to the increase in idle time instead of a decrease, as with other values of g. Large g cause greater imbalance, hence, induces large idle time.

For GDEM, the use of t_2 always increase the idle time (Table 5.8). If the increase exceed the total decrease of load balancing and synchronisation time, there will be positive results. Otherwise, there will be no benefit, as shown when $p = 128$.

Notice that there is zero improvement for both algorithms when $p = 4$ (Table 5.6). Zero improvement simply means that in practice the load balancing is never disabled yielding the same execution time. This behaviour can be best explained by means of a workload trace. Since $c_2 = 1$, the second transition should be detected when $n(t) \le p$. For cases where the total number of nodes is greater than p during the last iteration, as shown in Table 5.9, then t_2 is never met. Hence, there is no reduction in the total execution time in using t_2.

g	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
10	14.43	T_{NA}	0.1312	0.0063	0.0676	0.0512	0.0062	11
		T_A	0.1123	0.0063	0.0557	0.0447	0.0056	10
100	13.15	T_{NA}	0.1394	0.0099	0.0724	0.0509	0.0062	11
		T_A	0.1211	0.0099	0.0595	0.0461	0.0056	10
1000	5.36	T_{NA}	0.2321	0.0460	0.1256	0.0544	0.0062	11
		T_A	0.2197	0.0460	0.1020	0.0661	0.0056	10
10000	-2.48	T_{NA}	1.2642	0.4070	0.6774	0.1736	0.0062	11
		T_A	1.2955	0.4070	0.5472	0.3357	0.0056	10

Table 5.7: Grain size: Detailed cost of LDSV using t_2 only ($p = 128$).

We carried out a short experiment to see the effect of increasing c_2. Table 5.10 shows the improvement when $c_2 = 5$ and $c_2 = 10$. There are improvements in both cases but they are not significant.

Transition I and II

Table 5.11 shows the combined effects of both transitions. The following observations can be made:

- For both algorithms, ITA sometimes increases and sometimes decreases with g.

- Both algorithms can benefits from $t_1 t_2$ (in some circumstances).

- Overall, GDEM benefits more from $t_1 t_2$ than LDSV.

- Positive ITA for all g occurs for GDEM when $p = 128$, and for LDSV when $p = 32$.

Discussion

The net effect of both transitions are as follows. There is no improvement (or very little) for small processor size for both algorithms. This is anticipated because neither t_1 nor t_2 has any significant effect. The technique yields good improvement for very large g on medium and large processor size. No particular relationship can be established between $t_1 t_2$ and g for both algorithms.

In terms of individual overhead, the techniques always increases the synchronisation and the idle time for LDSV, but the load balancing time only increases some of the time. With GDEM, there is no particular pattern in terms of the increase or decrease of each of the sources of overhead (see Table 5.12).

g	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
10	-26.19	T_{NA}	0.0648	0.0062	0.0296	0.0122	0.0168	30
		T_A	0.0818	0.0062	0.0223	0.0404	0.0129	23
100	-46.20	T_{NA}	0.0722	0.0098	0.0324	0.0133	0.0168	30
		T_A	0.1056	0.0098	0.0239	0.0590	0.0129	23
1000	-134.87	T_{NA}	0.1463	0.0456	0.0598	0.0241	0.0168	30
		T_A	0.3436	0.0456	0.0408	0.2443	0.0129	23
10000	-207.11	T_{NA}	0.8868	0.4038	0.3339	0.1323	0.0168	30
		T_A	2.7236	0.4038	0.2093	2.0975	0.0129	23

Table 5.8: Grain size: Detailed cost of GDEM using t_2 only ($p = 128$).

Iteration	Tasks
454	39
455	31
456	27
457	25
458	29
459	19
460	21
461	12
462	10
463	7

Table 5.9: Total tasks during the last 10 iterations: a case when t_2 condition is not met (GDEM, $p = 4, g = 10$).

c_2	ITA(%)	Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
1	0.00	0.5081	0.1990	0.2134	0.0096	0.0860	463
5	0.51	0.5055	0.1990	0.2112	0.0101	0.0853	459
10	1.08	0.5026	0.1990	0.2091	0.0102	0.0844	454

Table 5.10: Greater improvement using t_2 when c_2 is adjusted (GDEM, $p = 4, g = 10$).

p	g	ITA (%)	
		GDEM	LDSV
4	10	-0.13	-0.20
	100	-0.08	-0.16
	1000	0.06	-0.05
	10000	0.12	-0.01
32	10	-0.14	5.25
	100	0.24	5.10
	1000	1.76	4.38
	10000	2.83	4.38
128	10	21.43	-6.34
	100	19.64	-6.11
	1000	11.72	-4.21
	10000	5.26	7.56

Table 5.11: Grain size: Improvement through adaptivity using $t_1 t_2$.

g	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
10	21.43	T_{NA}	0.0648	0.0062	0.0296	0.0122	0.0168	30
		T_A	0.0509	0.0062	0.0206	0.0100	0.0140	25
100	19.64	T_{NA}	0.0722	0.0098	0.0324	0.0133	0.0168	30
		T_A	0.0581	0.0098	0.0212	0.0130	0.0140	25
1000	11.72	T_{NA}	0.1463	0.0456	0.0598	0.0241	0.0168	30
		T_A	0.1292	0.0456	0.0271	0.0424	0.0140	25
10000	5.26	T_{NA}	0.8868	0.4038	0.3339	0.1323	0.0168	30
		T_A	0.8402	0.4038	0.0857	0.3366	0.0140	25

Table 5.12: Grain size: Detailed cost for GDEM using $t_1 t_2$ ($p = 128$).

The Best ITA and Techniques

Tables 5.13 and 5.14 show which combination of transition gives the best ITA for all combination of g and p. The following observations can be made:

- Adaptivity techniques bring improvements (in most circumstances) and it is more significant for large networks.

- There is no single technique which is the best for a given algorithm or processor size.

- The techniques $t_1 t_2$ or t_1 alone yield the best results for large g.

- GDEM seems to favour $t_1 t_2$ while LDSV favours t_1 and t_2 individually.

p	ITA (%) and Technique			
	10	100	1000	10000
4	0.00 (t_2)	0.00 (t_2)	0.06 (t_1, t_1t_2)	0.12 (t_1, t_1t_2)
32	1.55 (t_2)	1.37 (t_2)	1.76 (t_1t_2)	2.83 (t_1t_2)
128	21.43 (t_1t_2)	19.64 (t_1t_2)	21.02 (t_1)	25.52 (t_1)

Table 5.13: Grain size: The best improvement and techniques for GDEM.

p	ITA (%) and Technique			
	10	100	1000	10000
4	0.00 (t_2)	0.00 (t_2)	0.00 (t_2)	0.00 (t_2)
32	9.03 (t_1t_2)	9.60 (t_1t_2)	14.23 (t_1)	17.29 (t_1)
128	14.43 (t_2)	13.15 (t_2)	16.61 (t_2)	40.97 (t_1)

Table 5.14: Grain size: The best improvement and techniques for LDSV.

5.3.2   Varying the Network Performance

This section presents the results of the ITA when the network performance, s, measured in terms of bandwidth, is varied relative to that of the T3D. To be precise, network speed, s, means that the bandwidth of an individual link is s times that of a single link on the T3D. By varying s while the cost of computational operation remains the same, it is possible to investigate how ITA varies with the communication-to-computation ratio of the underlying architecture. Since increasing g and increasing s have similar impact on the performance of the application, it would be expected that the ITA result would be similar.

Recall from Section 4.3.2 there are four cost models (i.e. $T_{send}, T_{recv}, T_{recvmin}, T_{pingpong}$) for the point-to-point communication and one cost model (i.e. $T_{allreduce}$) for collective operation used in the simulator. Experimenting the network speed means varying s with the term associated with the data size in all the models. Hence, the the general point-to-point cost model becomes;

$$a + b \times dsize \times \frac{1}{s}$$

while the collective operation is;

$$a + b \times p + c \times log(p) \times dsize \times \frac{1}{s}$$

p	s	Best interval (i_b)	
		GDEM	LDSV
4	0.25	256	128
	0.5	256	128
	1.0	256	128
	2.0	256	128
32	0.25	64	128
	0.5	64	128
	1.0	64	128
	2.0	64	128
128	0.25	32	256
	0.5	32	256
	1.0	32	256
	2.0	32	256

Table 5.15: Network performance: The best interval for a range of p and s.

As the value of s increases the network performance improves.

Determining i_2

The following observations can be made from Table 5.15.

- i_b does not change with s (for the range of s that were used).

- i_b decreases with p for GDEM and increases with p for LDSV.

For the range of s that were used i_b seems to be constant. In a separate experiment, which induces a greater communication overhead, as the value of s increases, i_b decreases. Therefore, similar pattern of results are anticipated if a wider range of s are used for this experiments. The decrease of i_b with s simply means that parallelism is preferred for fast machine. Load balancing should be invoked more frequently in fast machine to distribute the work. The need for work distribution is even more clear for large network. Thus, the interval gets smaller with p. However, this is not true for LDSV.

As with the grain size experiments we varied i from 1 to 1024 for each p and s to find the best interval, i_b. The complete results of i_b is shown in Table 5.15. Next, i_2 is determined (using the method established in Section 5.3.1). The value of i_2 for GDEM is 64 with 37.45% largest discrepancy when $p = 128$ and $s = 0.25$. For LDSV, $i_2 = 128$, with the worst discrepancy is 5.98% when $p = 4$ and $s = 0.25$.

p	s	ITA (%)	
		GDEM	LDSV
4	0.25	0.38	0.98
	0.5	0.53	1.06
	1.0	0.61	1.11
	2.0	0.65	1.13
32	0.25	7.27	11.45
	0.5	8.93	14.16
	1.0	10.20	15.66
	2.0	11.00	16.47
128	0.25	45.14	-1.15
	0.5	47.54	0.64
	1.0	49.33	1.71
	2.0	50.44	2.30

Table 5.16: Network performance: Improvement through adaptivity using t_1 only.

Transition I Only

Table 5.16 illustrates the benefit of adaptivity for varying s and p. The following observations can be made:

- ITA always increases with s.

- Both algorithms benefit from t_1 (in most circumstances).

- GDEM benefits more than the LDSV.

- The best ITA for all s for GDEM occurs when $p = 128$ and for LDSV when $p = 32$.

Discussion

As with g, both algorithms benefit from t_1 because the total time waiting for work during filling phase is reduced. GDEM seems to benefit more because frequent work distribution has reduced the imbalance too, hence the idle time (see Table 5.17). For LDSV the technique increases the idle time in most cases. Again in here, we see the benefit of parallelism (by means of a small interval value) during the filling phase as the communication gets cheaper (i.e. faster network). The reason why GDEM seems to perform best when $p = 128$ whereas LDSV performs best when $p = 32$ (as in the grain size experiments) is not currently clear.

s	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.5	13.23	T_{NA}	0.0744	0.0392	0.0265	0.0044	0.0043	22
		T_A	0.0646	0.0392	0.0179	0.0029	0.0047	24
1.0	10.20	T_{NA}	0.0888	0.0392	0.0356	0.0054	0.0086	22
		T_A	0.0798	0.0392	0.0272	0.0039	0.0094	24
2.0	6.49	T_{NA}	0.1178	0.0392	0.0538	0.0077	0.0172	22
		T_A	0.1102	0.0392	0.0461	0.0061	0.0188	24
4.0	2.78	T_{NA}	0.1761	0.0392	0.0903	0.0122	0.0345	22
		T_A	0.1712	0.0392	0.0841	0.0103	0.0376	24

Table 5.17: Network performance: Detailed cost for GDEM using t_1 only ($p = 32$).

Transition II Only

Table 5.18 shows the improvement of adaptivity using t_2. The following observations can be made:

- ITA always decreases with s.

- Both algorithms can benefit from t_2 (in some circumstances), though the benefit is less than t_1.

- LDSV benefits more from the approach than GDEM (similar with the results of g experiment).

- Negative ITA for all s occurs for GDEM when $p = 128$ (exhibiting the same behaviour as g). Almost all negative ITA for LDSV when $p = 32$.

Discussion

For both algorithms the benefit of t_2 comes from the reduction of load balancing and synchronisation cost. In the case of LDSV these reduction always exceeds the idle time resulting a positive ITA. This is not true for GDEM. Positive ITA only obtained for small p. The large negative ITA for GDEM when $p = 128$ is due to an early switch-off point, which leads to a huge load imbalance, hence, idle time (see Table 5.19). The improvement decreases with s shows that the technique is more beneficial for slower networks.

p	s	ITA (%)	
		GDEM	LDSV
4	0.25	0.08	0.03
	0.5	0.07	0.03
	1.0	0.07	0.03
	2.0	0.07	0.03
32	0.25	0.58	2.46
	0.5	-1.79	1.83
	1.0	-3.39	1.47
	2.0	-4.33	1.28
128	0.25	-190.78	6.66
	0.5	-264.67	5.62
	1.0	-319.73	5.00
	2.0	-354.53	4.66

Table 5.18: Network performance: Improvement through adaptivity using t_2 only.

s	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.2	-190.78	T_{NA}	0.1553	0.0098	0.0541	0.0454	0.0461	24
		T_A	0.4517	0.0098	0.0131	0.4153	0.0134	7
0.5	-264.67	T_{NA}	0.1211	0.0098	0.0500	0.0370	0.0243	24
		T_A	0.4414	0.0098	0.0120	0.4126	0.0071	7
1.0	-319.73	T_{NA}	0.1040	0.0098	0.0479	0.0328	0.0135	24
		T_A	0.4363	0.0098	0.0114	0.4112	0.0039	7
2.0	-354.53	T_{NA}	0.0954	0.0098	0.0469	0.0307	0.0080	24
		T_A	0.4338	0.0098	0.0112	0.4105	0.0023	7

Table 5.19: Network performance: Detailed cost for GDEM using t_2 only ($p = 128$).

p	s	ITA (%)	
		GDEM	LDSV
4	0.25	0.38	1.06
	0.5	0.53	1.14
	1.0	0.61	1.19
	2.0	0.65	1.21
32	0.25	9.04	11.69
	0.5	10.22	14.28
	1.0	11.15	15.77
	2.0	11.75	16.58
128	0.25	49.72	-1.15
	0.5	50.52	0.64
	1.0	51.11	1.71
	2.0	51.46	2.30

Table 5.20: Network performance: Improvement through adaptivity using both $t_1 t_2$.

Transition I and II

Table 5.20 shows the results of the net effect of activating t_1 and t_2. The following observations can be made:

- ITA always increases with s.

- Both algorithms can benefit from $t_1 t_2$, in most circumstances.

- The benefit is more significant for medium and large processor sizes for GDEM and only medium processor size for the LDSV.

Discussion

Both algorithms benefit from the technique because of the huge reduction of load balancing time. For LDSV the combined technique always increases the idle time (see Table 5.21) while it sometimes increases idle time for GDEM. For the cases where the idle time decreases, the improvement is the largest (e.g. for GDEM when $p = 128$).

s	ITA(%)		Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.5	18.33	T_{NA}	0.0821	0.0099	0.0456	0.0241	0.0025	9
		T_A	0.0671	0.0099	0.0287	0.0254	0.0031	11
1.0	1.71	T_{NA}	0.1245	0.0099	0.0687	0.0409	0.0050	9
		T_A	0.1224	0.0099	0.0552	0.0512	0.0062	11
2.0	-5.41	T_{NA}	0.2210	0.0099	0.1173	0.0837	0.0101	9
		T_A	0.2329	0.0099	0.1081	0.1026	0.0123	11
4.0	-9.69	T_{NA}	0.4140	0.0099	0.2158	0.1681	0.0202	9
		T_A	0.4541	0.0099	0.2141	0.2055	0.0246	11

Table 5.21: Network performance: Detailed cost of LDSV using $t_1 t_2$ ($p = 128$).

The Best ITA and Techniques

Tables 5.22 and 5.23 show which combination of transition gives the best ITA for all combination of s and p. The following observations can be made:

- Adaptivity always brings improvements and is more significant for large p for GDEM and medium p for LDSV.

- $t_1 t_2$ yields the best results for GDEM.

- $t_1 t_2$ seems to be beneficial for LDSV too, but for very large p, t_2 is preferable.

p	ITA (%) and Technique			
	0.5	1.0	2.0	4.0
4	0.38 $(t_1, t_1 t_2)$	0.53 $(t_1, t_1 t_2)$	0.61 $(t_1, t_1 t_2)$	0.65 $(t_1, t_1 t_2)$
32	9.04 $(t_1 t_2)$	10.22 $(t_1 t_2)$	11.15 $(t_1 t_2)$	11.75 $(t_1 t_2)$
128	49.72 $(t_1 t_2)$	50.52 $(t_1 t_2)$	51.11 $(t_1 t_2)$	51.46 $(t_1 t_2)$

Table 5.22: Network performance: The best improvement and techniques for GDEM.

p	ITA (%) and Technique			
	0.5	1.0	2.0	4.0
4	1.06 $(t_1 t_2)$	1.14 $(t_1 t_2)$	1.19 $(t_1 t_2)$	1.21 $(t_1 t_2)$
32	11.69 $(t_1 t_2)$	14.28 $(t_1 t_2)$	15.77 $(t_1 t_2)$	16.58 (t_2)
128	6.66 $(t_1 t_2)$	5.62 (t_2)	5.00 (t_2)	4.66 (t_2)

Table 5.23: Network performance: The best improvement and techniques for LDSV.

5.3.3 Varying the Tree Imbalance

This section presents the results of varying the degree of tree imbalance on ITA when the same three adaptive techniques, t_1 alone, t_2 alone and the combination of t_1 and t_2, were applied.

As noted earlier, this set of experiments use imbalance tree instead of random tree because the degree of imbalance can be parameterised in the former. However, as highlighted in Section 4.2.3, the imbalance tree is not repeatable – each run may produce different tree size and shape. This has two implications; the first is on the pattern of ITA and the second is how ITA is measured. In some cases, the pattern of ITA is not as consistent as those of g and s, to enable a clear relationship being made. Despite that some general pattern can still be established.

Recall from Section 5.2.1 the ITA for experimenting the imbalance refer to speed instead of the execution time. Detail comparison in terms of the amount of improvement made between g and s, and m may not be appropriate but the comparison between the general relationships established is still valid.

Note that an additional column for speed is incorporated in the detailed measurements.

Determining i_2

Table 5.24 shows the best interval for the whole parameter set. The following observations can be made:

- i_b decreases with m.

- i_b decreases with p.

- GDEM has a smaller i_b if compared to LDSV.

The value of i_b decreases with m because as the degree of imbalance increases the workload gets more irregular resulting in a larger number of idle (or lightly loaded) processors.

p	m	Best interval (i_b)	
		GDEM	LDSV
	0.0	512	256
4	0.1	256	256
	0.2	128	1024
	0.3	64	128
	0.0	128	256
32	0.1	64	128
	0.2	32	128
	0.3	32	64
	0.0	32	128
128	0.1	32	64
	0.2	16	64
	0.3	16	64

Table 5.24: Tree imbalance: The best interval for a range of p and m.

Load balancing is needed to distribute the workload to these processors. Therefore, the invocation has to be more frequent for a tree with higher degree of imbalance. This is especially true when larger processor sizes are used.

As with g and s, the value of i is varied from 1 to 1024 to find i_b for each p and m. We then selected the best i_b as i_2. Since the size and the shape of the tree changes in each run, finding i_b with the least discrepancies is not sensible. For each algorithm we chose the value of i_b which occurs most frequently (please refer to Table 5.24). The value of i_2 for GDEM is 32 and LDSV is 128.

Transition I Only

Table 5.25 shows the benefit gained from t_1 for varying m. The following observations can be made:

- ITA usually increases with m.

- Both algorithms can benefit from the techniques most of the time.

- GDEM benefits more than LDSV.

- Positive ITA occurs for all m for GDEM when $p = 128$ and for LDSV when $p = 32$. For both cases ITA increases consistently with m.

p	m	ITA (%)	
		GDEM	LDSV
4	0.0	8.77	1.20
	0.1	-0.01	0.41
	0.2	0.84	9.30
	0.3	6.45	-8.84
32	0.0	0.40	1.45
	0.1	3.28	6.31
	0.2	13.60	36.64
	0.3	-0.80	60.35
128	0.0	24.48	-6.33
	0.1	25.07	31.21
	0.2	62.03	45.29
	0.3	145.32	-8.71

Table 5.25: Tree imbalance: Improvement through adaptivity using t_1 only.

Discussion

The increase of ITA with m is anticipated because when the degree of imbalance increases the workload gets more irregular and some of the processors have no work. Therefore, frequent invocation of load balancing will reduce idle time and improve the performance.

Again, we see the same pattern of results of positive improvement on $p = 128$ for GDEM and $p = 32$ for LDSV. We can provide no adequate explanation for such preferences. From the detail measurements of the two results (in Tables 5.26 and 5.27) we could derived that the large improvement for GDEM is due to a large decrease in load balancing and idle time (though the synchronisation increases in most cases). LDSV tends to have a decrease in load balancing time in most of the cases too. However, idle time and synchronisation time increases.

m	ITA(%)		Speed (node/sec)	Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.0	24.48	S_{NA}	673040.4376	0.0974	0.0217	0.0383	0.0194	0.0180	32
		S_A	837819.9228	0.0782	0.0217	0.0295	0.0095	0.0174	31
0.1	25.07	S_{NA}	558487.6674	0.0708	0.0131	0.0326	0.0122	0.0129	23
		S_A	698495.5277	0.0620	0.0144	0.0261	0.0075	0.0140	25
0.2	62.03	S_{NA}	294810.7026	0.0579	0.0057	0.0262	0.0154	0.0107	19
		S_A	477673.6721	0.0433	0.0069	0.0202	0.0051	0.0112	20
0.3	145.32	S_{NA}	148215.4330	0.0511	0.0025	0.0216	0.0175	0.0095	17
		S_A	363596.5354	0.0384	0.0046	0.0186	0.0051	0.0101	18

Table 5.26: Tree imbalance: Detailed cost for GDEM using t_1 only ($p = 128$).

m	ITA(%)		Speed (node/sec)	Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.0	1.45	S_{NA}	362186.0614	0.1809	0.0898	0.0586	0.0244	0.0082	21
		S_A	367431.9807	0.1784	0.0898	0.0444	0.0345	0.0097	25
0.1	6.31	S_{NA}	263658.2041	0.0993	0.0359	0.0424	0.0164	0.0047	12
		S_A	280289.5869	0.1419	0.0545	0.0469	0.0323	0.0082	21
0.2	36.64	S_{NA}	196081.4980	0.0825	0.0222	0.0418	0.0146	0.0039	10
		S_A	267915.9652	0.0828	0.0304	0.0298	0.0171	0.0054	14
0.3	60.35	S_{NA}	130441.2734	0.0735	0.0131	0.0437	0.0133	0.0035	9
		S_A	209165.3837	0.0649	0.0186	0.0274	0.0142	0.0047	12

Table 5.27: Tree imbalance: Detailed cost for LDSV using t_1 only ($p = 32$).

p	m	ITA (%)	
		GDEM	LDSV
	0.0	0.07	-27.66
4	0.1	0.13	-0.38
	0.2	0.00	1.45
	0.3	0.30	-0.31
	0.0	0.86	2.69
32	0.1	0.72	-2.11
	0.2	0.71	-16.10
	0.3	0.94	25.42
	0.0	-38.95	10.19
128	0.1	2.94	-1.23
	0.2	-4.31	23.29
	0.3	-78.83	11.62

Table 5.28: Tree imbalance: Improvement through adaptivity using t_2 only.

Transition II Only

Table 5.28 presents the results for t_2 when m is varied. The following observations can be made:

- No clear relationship can be made between ITA and m since the improvement oscillates.

Discussion

No clear relationship can be noted between ITA and m for this case, but we analysed the pattern of each of the sources of overhead. The results for g and s have shown a consistent decrease in synchronisation and load balancing time due to the discontinuing of the load balancing. GDEM produces this same pattern (see Table 5.29). For cases where the increase in idle time exceed the benefit of reduced load balancing time and synchronisation, ITA is negative. For LDSV, the load balancing time decreases but the idle time increases as well (as shown in Table 5.30).

m	ITA(%)		Speed (node/sec)	Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.0	0.86	S_{NA}	352631.4405	0.1858	0.0870	0.0631	0.0092	0.0266	68
		S_A	355677.4845	0.1843	0.0870	0.0604	0.0106	0.0262	67
0.1	0.72	S_{NA}	322581.7516	0.1261	0.0540	0.0477	0.0064	0.0180	46
		S_A	324898.2208	0.1253	0.0540	0.0436	0.0104	0.0172	44
0.2	0.71	S_{NA}	267746.0253	0.0686	0.0244	0.0296	0.0048	0.0098	25
		S_A	269653.4479	0.0682	0.0244	0.0241	0.0110	0.0086	22
0.3	0.94	S_{NA}	264760.4301	0.0529	0.0186	0.0237	0.0032	0.0074	19
		S_A	267259.6906	0.0524	0.0186	0.0206	0.0062	0.0070	18

Table 5.29: Tree imbalance: Detailed cost for GDEM using t_2 only ($p = 32$).

m	ITA(%)		Speed (node/sec)	Total Time (sec)	Compute (sec)	Load Balance (sec)	Idle (sec)	Synch (sec)	Iter
0.0	2.69	S_{NA}	362186.0614	0.1809	0.0898	0.0586	0.0244	0.0082	21
		S_A	371930.2329	0.1762	0.0898	0.0489	0.0298	0.0078	20
0.1	-2.11	S_{NA}	263658.2041	0.0993	0.0359	0.0424	0.0164	0.0047	12
		S_A	258098.6493	0.1136	0.0402	0.0502	0.0181	0.0050	13
0.2	-16.10	S_{NA}	196081.4980	0.0825	0.0222	0.0418	0.0146	0.0039	10
		S_A	164517.0066	0.0822	0.0185	0.0407	0.0195	0.0035	9
0.3	25.42	S_{NA}	130441.2734	0.0735	0.0131	0.0437	0.0133	0.0035	9
		S_A	163594.2164	0.0615	0.0138	0.0332	0.0119	0.0027	7

Table 5.30: Tree imbalance: Detailed cost for LDSV using t_2 only ($p = 32$).

p	m	ITA (%)	
		GDEM	LDSV
4	0.0	25.21	-25.30
	0.1	-0.01	0.44
	0.2	0.84	5.79
	0.3	6.88	3.82
32	0.0	1.14	-15.08
	0.1	4.16	24.60
	0.2	14.51	40.70
	0.3	5.45	64.08
128	0.0	11.92	5.61
	0.1	25.92	39.77
	0.2	67.77	47.76
	0.3	166.97	-4.66

Table 5.31: Tree imbalance: Improvement through adaptivity using both $t_1 t_2$.

Transition I and II

Table 5.31 illustrates the net-benefit of both transitions when m is varied. The following observations can be made:

- ITA usually increases with m (and is more significant for large p).

- Both algorithms can benefit from the technique (most of the time).

- Overall, GDEM benefits more from $t_1 t_2$ than LDSV.

- Positive ITA for all m for GDEM occurs when $p = 32$ and $p = 128$ and almost all positive ITA occurs for LDSV for the same p.

Discussion

The pattern of results are similar to g and s, where the benefits comes from the reduction of the load balancing time. GDEM tends to benefit more because there is a reduction in the syncronisation cost as well. As with g and s, the benefit from using t_1 is more prominent, hence the observations made for t_1 alone applies to $t_1 t_2$. Note that we see the recurrences of preference of the algorithms on a particular p in this experiment as well.

p	ITA (%) and Technique			
	0.0	0.1	0.2	0.3
4	25.21 ($t_1 t_2$)	0.13 (t_2)	0.84 (t_1,$t_1 t_2$)	6.88 ($t_1 t_2$)
32	1.14 ($t_1 t_2$)	4.16 ($t_1 t_2$)	14.51 ($t_1 t_2$)	5.45 ($t_1 t_2$)
128	24.48 (t_1)	25.92 ($t_1 t_2$)	67.77 ($t_1 t_2$)	166.97 ($t_1 t_2$)

Table 5.32: Tree imbalance: The best improvement and techniques for GDEM.

p	ITA (%) and Technique			
	0.0	0.1	0.2	0.3
4	1.20 (t_1)	0.44 ($t_1 t_2$)	9.30 (t_1)	3.82 ($t_1 t_2$)
32	2.69 (t_2)	24.60 ($t_1 t_2$)	40.70 ($t_1 t_2$)	64.08 ($t_1 t_2$)
128	10.19 (t_2)	39.77 ($t_1 t_2$)	47.76 ($t_1 t_2$)	11.62 (t_2)

Table 5.33: Tree imbalance: The best improvement and techniques for LDSV.

The Best ITA and Techniques

Tables 5.32 and 5.33 show which combination of transition gives the best ITA for all combination of m and p. The following observations can be made:

- Adaptivity always bring improvement, and is more significant for large p for GDEM and medium p for LDSV.

- $t_1 t_2$ usually yields the best results for GDEM.

- $t_1 t_2$ sometimes yields the best results for LDSV.

5.4 Experimenting with Algorithmic Adaptivity

The non-adaptive results with GDEM and LDSV (see Figures 4.15 and 4.16) show that LDSV execution time are much worse than GDEM's. It was proposed in Chapter 3 that algorithmic adaptivity using t_1 could employ LDSV in the filling phase and Diffusion (or GDEM) in the steady phase. It is clearly impossible to carry out a sensible experiment at this stage, since the poor absolute performance of LDSV means that t_1 would substantially degrade performance compared with GDEM alone.

This does not mean that the hypothesis of using the algorithmic adaptivity at t_1 is invalidated. The poor performance of LDSV relative to GDEM is accounted for by two

factors. First, LDSV is essentially an asynchronous algorithm which has been artificially synchronised by the simulator, whereas GDEM is inherently synchronous. Moreover, LDSV is an experimental algorithm which has substantial scope for further optimisation (see Section 6.3), whereas GDEM is a well-established and optimised algorithm. Proper experimental work on t_1 must therefore await for a more optimal version of LDSV. Recall that LDSV will always complete the filling stage in optimal number of iterations (see Section 3.1.3), there is clearly scope for this further experimentation to yield positive results for t_1 transition.

Transition t_2 in algorithmic adaptivity is exactly the same as in parametric adaptivity, the benefits of which have already been demonstrated in previous section.

5.5   Summary Results

The following are the general observations made:

- Parametric phase-based adaptivity can bring substantial performance improvement for both GDEM and LDSV in many situations.

- Both algorithms show significant improvement for medium and large processor sizes.

- t_1 is suitable for large grain node, fast network or high degree of tree imbalance.

- t_2 is the reverse; it is suitable for small grain node, slow network and a more balance tree.

- For GDEM $t_1 t_2$ is suitable for large processor sizes, and large grain node or slow network or high degree of imbalance. For LDSV $t_1 t_2$ is more suitable for high degree of imbalance, and small and medium processor sizes (regardless of network speed).

- Overall, $t_1 t_2$ usually yields the best results if compared to t_1 alone or t_2 alone.

- When t_1 is used, the best performance for GDEM always occur when $p = 128$ and for the LDSV when $p = 32$.

- There is a symmetry in the performance of both algorithms.

  - GDEM benefits more from t_1 where as LDSV from t_2.
  - There are preferences in techniques that yields the best results. GDEM always favours $t_1 t_2$ followed by t_1 then t_2. LDSV usually favours $t_1 t_2$ followed by t_2 then t_1.

- The best interval decreases with the grain size, network speed and tree imbalance.

- Experimental investigation of algorithmic adaptivity can not sensibly be done until equivalently optimised asynchronous version of the algorithm concerned is available.

Chapter 6

Conclusion

The final chapter presents a summary of the work described in this thesis, highlights the main contributions and identifies possible future direction of the work.

6.1 Summary and Evaluation

This section briefly describes the framework for adaptivity, and the tool used to evaluate the suggested techniques, followed by a summary of the results.

6.1.1 Conceptual Framework

The thesis defines a new conceptual framework for adaptive dynamic load balancing (DLB) for parallel tree computation. It introduces the notion of *phase-based* adaptivity and its variants - the *parametric* and *algorithmic* approach - and positions the new approach in relation to the existing adaptivity and phase-based techniques. We refer to the existing work on adaptivity as *periodic* approach.

The periodic adaptivity does not assume any prior knowledge of the workload of the system. The DLB parameters (or policies or algorithms) are adjusted from time to time, according to the current local (or global) state of the system. Such an approach is more applicable to system level load balancing where the nature of workload are usually processes of varying sizes and arrival rate, recognised by the operating system.

On the contrary, phase-based adaptivity requires some knowledge of the "evolution" of the workload. The DLB algorithms makes an adjustment according to this (prior) knowledge of its "surroundings" to optimise the overall performance. Hence, the approach is more applicable to parallel applications, whose general workload pattern could be antici-

pated beforehand.

Phase-based adaptivity carries three basic concepts;

- the workload *phases*;

- *transitions* between these phases; and

- and the *mechanisms* to adapt at these transitions.

We chose tree computation to illustrate the above concepts. Tree computation starts with a single node. At this stage, only one processor has work, while the rest are idle. The aim of the DLB is to quickly distribute the work. As more nodes are created more processors will have work and finally the whole machine is filled. We call this stage the *filling phase* or phase I. Once all processors are busy, the aim of the DLB is to maintain this state as much as possible. This we called the *steady phase* or phase II. Eventually the workload becomes so low that it is not possible to utilise all the processors. At this stage, there is little benefit in using DLB. We name this stage the *emptying phase* or phase III.

From the above, there are three distinct phases, thus we can define two transitions to switch between these phases. The transition can be defined by the number of tasks currently in the whole system in relation to the number of processors used to execute the application. The first transition, t_1, occurs when the number of tasks generated start exceeding (or at least equal to) the number of processors, while the second, i.e. t_2, is when the total tasks drop sufficiently low relative to the total processors.

There are two mechanisms that could be used to adapt at these two transitions:

- Parametric approach.

  This technique adjusts the values of any of DLB parameters (e.g. the load balancing interval, workload threshold or migration factor) according to the workload phases. For example the interval value of one is used during phase I (or filling phase) to facilitate work distribution and the best interval is used in phase II (or steady phase) to obtain the best possible execution time and an infinity during phase III (or emptying phase) to avoid invoking the load balancing. The effect of a very large interval is equivalent to disabling the dynamic load balancing algorithm.

- Algorithmic approach.

  Although the idea of algorithmic adaptivity already exists in the context of the distributed systems, it has never been experimented nor discussed in relation to the DLB algorithms used in this thesis. Algorithmic adaptivity entails the use of completely different DLB algorithms at different workload phases. A global method, for example the LDSV, is suggested during phase I to facilitate work distribution so as to quickly fill the machine. A local method, such as Diffusion (or GDEM), is preferred in phase II in order to maintain the steady state. This is followed by a complete disabling of the algorithm in phase III.

The basic principles behind both techniques are to speed up the initial workload distribution, maintain a steady state when the machine is full and remove the overhead of the load balancing at the end of the computation. An example of a full-scale application that may adopt the phase-based adaptivity technique is VLSI floorplan optimisation, which is used in microprocessor and memory chip designs [22].

As it stands now, the conceptual framework presented in this thesis applies only to the tree computation. The generality and the applicability of the idea has yet to be tested. We leave this for future work.

6.1.2 The Simulator

In order to investigate the idea of phase-based adaptivity quantitatively, a simulator was developed which allows different load balancing algorithms to be tested, with varying network speeds (relative to the T3D), and simulated workloads representing trees of varying depth, grain size and imbalance.

The use of a sequential simulator speeds up the experimental process when compared with developing a full parallel testbed. More importantly, it allows the potential of both types of adaptivities, algorithmic and parametric, to be cheaply and easily assessed without the need for a global collection of processor workloads and distributed detection of phase changes.

Extensive validations on the simulator were carried out in two stages, namely, iteration count and cost validations. The predicted and the real iteration count results showed at least a close, if not an exact, number of iterations between the two. The cost validation showed a close prediction to the real measurement, with the predicted time within 25%, for the processor range used in the experiments (i.e. 1 to 128). These results are better

than (or at least as good as) the reported parallel simulation results in the literature. Therefore, we conclude that the validity and reliability of the simulator are justified.

6.1.3 Experimental Results

The purpose of the experiments is to investigate the performance gained from parametric phase-based adaptivity. There are three main experiments, each corresponds to experimenting with varying grain size, network speed and tree imbalance. Two load balancing algorithms, GDEM and LDSV, are evaluated for all experiments.

Before investigating the effects of the adaptive approach, we first study the performance impact of a sensitive parameter, the load balancing interval, in a non-adaptive situation. The conclusions of the non-adaptive results are:

- The performance of the tree application is very sensitive to the value of the load balancing interval. Therefore, it is important to obtain the interval value which would yield the best result.

- The results also reveal that the best interval is algorithm dependent. Hence, there is no single best value for all DLB algorithms.

- The best interval increases as the node grain size, the network speed and the tree imbalance decreases.

After gaining the above insights, we proceeded to adaptivity experiments to investigate the relationship between the performance improvement with the above mentioned application and machine parameters by activating individual transition or the combination of both.

There are three sub-experiments which correspond to the three adaptive techniques for each application or machine parameter (i.e. grain size, network speed and tree imbalance);

(a) t_1 only - an interval of one is used during phase I followed by the best interval,

(b) t_2 only - the best interval is used during phases I and II followed by no load balancing during phase III and

(c) $t_1 t_2$ - interval one is used during phase I, followed by the best interval and a disabling of the balancer, during phases II and III, respectively.

The following are the insights obtained from the parametric phase-based adaptivity experiments:

- The experiments show that the parametric adaptivity approach always bring benefits. The benefits or improvements can either come from t_1 alone, t_2 alone or the combination of the two. This confirms the importance of the two transitions defined in the conceptual framework.

  The relative merits of the techniques are as follows:

  - t_1 is suitable for an application with high computation to communication ratio. In other words, large grain tree applications and high speed networks will benefit most from this approach. The tree with high degree of imbalance is expected to benefit as well, though the benefit may not always be consistent.

  - t_2 on the other hand, is beneficial if the application has a low computation to communication ratio. Therefore, the opposite applies - small grain node and slower networks benefit. As to the tree imbalance, there is no clear relationship that could be established. The benefit gained from t_2 is lesser than t_1 for all the parameters.

  - $t_1 t_2$ exhibits a similar pattern of improvement to t_1 for all three parameters, indicating a greater advantage gained from t_1 compared to t_2 when both techniques are combined. Hence, as the grain size and the network speed increases, the benefit increases consistently. High tree imbalance, however, does not show consistent advantage, but the pattern is similar to t_1.

- From the results it can be derived that the best overall technique (most of the time) for the three parameters is $t_1 t_2$. Our observations also show that GDEM favours $t_1 t_2$ more than LDSV. As to the single transition technique, t_1 gives more benefit to GDEM and t_2 to the LDSV. We can conclude that the performance of each technique is algorithm dependent.

- The improvement gained through adaptivity increases with the grain size, network speed, and processor size. It usually increases with the degree of imbalance.

- The benefit of parametric phase-based adaptivity is more significant for medium and large processor sizes.

For algorithmic adaptivity experiments, LDSV should be used during phase I, followed by GDEM during phase II and a disable of the load balancing in phase III. Both algorithms are expected to use the best interval values during the first two phases.

However, due to the relatively poor performance of the unoptimised synchronous version of LDSV algorithm as compared to GDEM, it is not sensible to proceed with algorithmic adaptivity experiments.

The poor performance of LDSV shows that the synchronous simulator is not suitable to be used to study algorithmic adaptivity. Since many DLB algorithms are asynchronous in nature, implementing an asynchronous simulator (e.g. an event-driven simulation) can be identified as a future work. Only then, a fuller experiments which encompass both algorithmic adaptivity and parametric adaptivity can be carried out.

6.2 Contributions

The following are the main contributions of this thesis:

- A new conceptual framework for adaptivity in dynamic load balancing has been proposed (explained in Chapter 3 and summarised in Section 6.1.1).

- A sequential simulator to explore this framework has been developed and closely validated (described in Chapter 4 and a short summary is provided in Section 6.1.2).

- The results of the investigation are presented in Chapter 5 and Section 6.1.3. They can be summarised as follows:

  - The parametric phase-based adaptivity techniques have demonstrated good performance improvement.

  - The conditions under which each transition (or the combination of both) provide the most benefit have been identified.

  - The best overall technique for all application and machine parameters considered (i.e. grain size, network speed and tree imbalance) has also been identified.

The following is subsidiary contribution (not directly related to adaptivity):

- A dynamic load balancing algorithm, Loadserver, initially proposed by Davy [14] has been implemented on the T3D (Chapter 2).

6.3 Future Work

The above research can be extended in two main directions:

6.3.1 Further Simulation Studies

The work reported in this thesis covered only one type of parametric phase-based adaptivity, that is the load balancing interval. There are other parameters that can be experimented with, such as the workload threshold and load migration factor. These experiments can easily be carried out using the existing simulator without any major changes.

Within the same scope, the effects of other application and machine parameters, such as the fan-out and topology, can be investigated. Apart from 2-d torus, the simulator currently supports ring, chain, mesh and hypercube. Experimenting with the effects of varying fan-out or processor connectivity are straight forward since these facilities are already incorporated within the simulator.

Another potential further work is experiments to optimise the performance of the LDSV algorithm. An improvement to the LDSV can be undertaken in two ways. First, using a larger migration factor instead of one. By doing so, more than one tasks may be packed and sent to a single destination. There is also the possibility of having varying number of tasks in a single pack. Second, more than one light node id can be returned to the heavy node upon making a request for light node ids. Hence, the extra work could be distributed to these light nodes, as in the case for a single request to the information server. This may facilitate work distribution further and help minimise the cost during filling phase. These two optimisations are expected to reduce the total cost of the LDSV algorithm, and perhaps amplify the benefits gained from the algorithmic adaptivity approach.

6.3.2 Real Parallel Implementation

The simulation results have shown the potential performance benefit of phase-based adaptivity. Therefore, the natural step forward is to consider practical implementations with a real parallel dynamic load balancing algorithm and a real application (e.g. divide-and-conquer). The main challenges in real parallel implementation revolves around the issue of acquiring the global system state (i.e. the number of tasks currently in the system) and to detect phase transition in a distributed manner with minimum additional overhead. Otherwise the benefit of adaptivity can not be justified.

In the synchronous model (as in the simulator), one can take the advantage of the termination detection algorithm. For example, the *MPI_Allreduce*() function can be used to gather the status, as well as to collect the workload of the processors through the same global reduction. No additional facility is needed. Transitions can be checked after retrieving the workload state every time if the computation has not reached to an end.

Many DLB algorithms are asynchronous [9, 32, 50, 76, 82, 87]. Therefore, it is important to test the idea in this context. With the asynchronous parallel implementation, the issue is more challenging. Acquiring the knowledge of the system state using message passing may be implemented in two ways. First, through a global or centralised facility if the algorithm uses one (e.g. LDSV algorithm). Second, again, through the termination detection algorithm. The system workload could be coupled with the status of the processor passed throughout the system as the case in token termination detection algorithm described in [45]. However, both techniques may suffer from information aging. In the centralised approach, additional communication is required for information collection and distribution.

The high-level sharing facilities provided by the TallShip project [28] may help ease the problem. The *shared accumulator*, for example, may be used to accumulate the current workload status of the system without tedious implementation of message-passing. The problem of information aging vanishes since information is collected only when it is needed. However, the effect on overall performance has yet to be assessed.

The above mentioned issues pose various practical challenges which we will leave for future investigation.

Bibliography

[1] *Minitab Reference Manual: Release 7*, April 1989.

[2] I. Ahmad and A. Ghafoor. Semi-Distributed Load Balancing For Massively Parallel Multicomputer Systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, October 1991.

[3] K. Ben-Mohammed. *An Evaluation of Load Balancing Algorithms for Distributed Systems*. PhD thesis, School of Computer Studies, University of Leeds, September 1991.

[4] J. E. Boillat. Load Balancing and Poisson Equation in a Graph. *Concurrency: Practise and Experience*, 2(4):289–313, December 1990.

[5] C. Calvin and L. Colombet. Performance Evaluation and Modeling of Collective Communications on Cray T3D. *Parallel Computing*, 22:1413–1427, 1996.

[6] K. Cameron, L. J. Clarke, and A. G. Smith. CRI/EPCC MPI for Cray T3D. http://www.epcc.ed.ac.uk/t3dmpi/Product/, 1997.

[7] T L Casavant and J G Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[8] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized Load Balancing for Tree-structured Computation. In *Proceeding of the IEEE Scalable High Performance Computing*, pages 666–673, 1994.

[9] L. Chengjiang and Li Sanli. Strategy and Simulation of Adaptive RID for Distributed Dynamic Load Balancing in Parallel Systems. In *International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, pages 406–412. IEEE Computer Society Press, December 1994.

[10] L. A. Crowl. How to Measure, Present, and Compare Parallel Performance. *IEEE Parallel & Distributed Technology*, pages 9–25, Spring 1994.

[11] G. Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[12] W. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[13] P. Dasgupta, A. K. Majumder, and P. Bhattacharya. V_THR: An Adaptive Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, 42:101–108, 1997.

[14] J. R. Davy. *Using Divide-and-Conquer for Parallel Geometric Evaluation*. PhD thesis, School of Computer Studies, University of Leeds, September 1992.

[15] J. R. Davy. Generating Random Tree. Personal Communication, 1997.

[16] J. R. Davy and P. M. Dew. Parallel Divide and Conquer Using a Task Pool Machine. Technical Report 91.32, School of Computer Studies, University of Leeds, October 1991.

[17] J.R. Davy and W. Essah. Generating Parallel Applications of Spatial Interaction Models. Accepted for Euro-Par '98, 1998.

[18] P. Dickens, P. Heidelberger, and D. Nicol. A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs. In *In Workshop on Parallel and Distributed Simulation*, pages 32–38, July 1994.

[19] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6:53–68, 1986.

[20] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Environments. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[21] J. Fisher. *Introduction to the Cray T3D at EPCC*, 1997.

[22] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1994.

[23] G. C. Fox. Achievements and Prospects for Parallel Computing. *Concurrency: Practise and Experience*, 3(6):725–739, December 1991.

[24] J. Gehring. Dynamic Program Description as a Basis for Runtime Optimisation. In *Third International Euro-Par Conference*, pages 958–965, August 1997.

[25] J. Gehring and A. Reinefeld. MARS - A Framework for Minimising the Job Execution Time in a Metacomputing Environment. *Future Generation Computer Systems*, 12:87–99, 1996.

[26] B. Gendron and T. G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, Nov-Dec 1994.

[27] D. Goodeve. *A Simple Portable Coroutines Library*. University of Leeds, April 1995.

[28] D. M. Goodeve, S. A. Dobson, J. M. Nash, J. R. Davy, P. M. Dew, M. Kara, and Wadsworth. Towards a Model for Shared Data Abstraction with Performance. *To appear in the Journal of Parallel and Distributed Computing*, 1998.

[29] M. G. Gouda and T. Herman. Adaptive Programming. *IEEE Transaction on Software Engineering*, 17(9):911–921, September 1991.

[30] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[31] T3D MPI Group. MPI for Cray T3D: Known Problems and Support. http://www.epcc.ed.ac.uk/t3dmpi/Product/Support/index.html, 1997.

[32] F. Haron and J. R. Davy. Loadserver - A Hybrid Dynamic Load Balancing Algorithm. In *Proceedings of the National Conference on Research and Development in Computer Science and its Applications (REDEC '96)*, pages 100–105, June 1996.

[33] F. Haron and J. R. Davy. Algorithmic Adaptivity in Dynamic Load Balancing. In *Proceedings of the Parallel and Distributed Computing and Networks (PDCN '97)*, pages 151–154, August 1997.

[34] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, 10:160–166, 1990.

[35] F. Howell. *Approaches to Parallel Performance Prediction*. Phd thesis, Dept of Computer Science, University of Edinburgh, 1996.

[36] F. Howell. Datasheets for MPI on Sun Sparc Workstations. http://www.dcs.ed.ac.uk/home/fwh/timing/index.html, 1996.

[37] M. Kara. A Global Plan Policy for Coherent Cooperation in Distributed Dynamic Load Balancing. *Distributed Systems Engineering Journal*, 2(4):212–223, December 1995.

[38] G. Karypis and V. Kumar. Unstructured Tree Search on SIMD Parallel Computers. *IEEE Transaction on Parallel and Distributed Systems*, 5(10):1057–1072, October 1994.

[39] K. Kennedy, C. F. Bender, J. W. D. Connolly, J. L. Hennessy, M. K. Vernon, and L. Smarr. A Nationwide Parallel Computing Environment. *Communications of the ACM*, 40(11):63–72, November 1997.

[40] P. Krueger and M. Livny. The Diverse Objectives of Distributed Scheduling Policies. In *The 7th International Conference on Distributed Computing Systems*, pages 242–249. IEEE Computer Society Press, September 1987.

[41] P. E. Krueger. *Distributed Scheduling for a Changing Environment*. PhD thesis, University of Wisconsin - Madison, 1988.

[42] N. Kuck, M. Middendorf, and H. Schmeck. Generic Branch-and-Bound on a Network of Transputers. In R. et. al. Grebe, editor, *Transputer Applications and Systems '93*, pages 521–535. IOS Press, 1993.

[43] J. Mohan Kumar, L.M. Patnaik, and A. Das. Load Balancing Algorithms for an Extended Hypercube. *Computer Digital Technology*, 141(5):298–306, September 1994.

[44] V. Kumar, P. S. Gopalakrishnan, and L. N. Kanal. *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, 1990.

[45] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[46] S. Kumaran and M. J. Quinn. Divide-and-Conquer Programming on MIMD Computers. In *Proceedings of 9th International Symposium on Parallel Processing*, pages 734–740, April 1995.

[47] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[48] F. C. H. Lin and R. M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[49] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An Algorithm for the Travelling Salesman Problem. *Operations Research*, 11(6):972–989, 1963.

[50] R. Luling, B. Monien, and F. Ramme. Load Balancing in Large Networks: A Comparative Study. In *Proceeding of the 3rd Symposium on Parallel and Distributed Processing*, pages 686–689, 1991.

[51] S. Martello and P. Toth. Algorithms for Knapsack Problems. In *Annals of Discrete Mathematics*, pages 213–258. Elsevier Science, 1987.

[52] Oliver A. McBryan. An Overview of Message Passing Environments. *Parallel Computing*, 20:417–444, Apr 1994.

[53] D. McBurney and M. R. Sleep. Transputers + Virtual Tree Kernel = Real Speedups. In *Proceedings 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 128–137, January 1988.

[54] W. F. McColl. Bulk Synchronous Parallel Computing. In *Abstract Machine Models for Parallel and Distributed Computing*, pages 41–63. Oxford University Press, April 1995.

[55] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. Adaptive Load Balancing of Irregular Applications. In *3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR '96)*, August 1996.

[56] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. An Adaptive Load Information Collection Policy. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, August 1996.

[57] J. M. Nash, P. M. Dew, J. R. Davy, and M. E. Dyer. Scalable Dynamic Load Balancing Using a Highly Concurrent Shared Data Type. In *The 2nd European School of Computer Science: Parallel Programming Environments for High Performance Computing*, pages 123–128, April 1996.

[58] J.M. Nash, P.M. Dew, and J.R. Davy. A Parallelisation Approach for Supporting Scalable and Portable Computing. In *Euro-Par '97*, pages 678–682, 1997. Springer Lecture Notes in Computer Science 1300.

[59] L. M. Ni and K. Hwang. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. *IEEE Transactions on Software Engineering*, SE-11(5):491–496, May 1985.

[60] M.G. Norman and P. Thanisch. Models of Machines and Computation for Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):263–302, September 1993.

[61] H. Peitgen, H. Jurgens, and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, 1992.

[62] C. Powley, C. Ferguson, and R. E. Korf. Parallel Tree Search on a SIMD Machine. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 249–256, 1991.

[63] S. Prakash. *Performance Prediction of Parallel Programs*. PhD thesis, University of California - Los Angeles, November 1996.

[64] K. Ramamritham, J. Stankovic, and W. Zhao. Meta-level Control in Distributed Real-time Systems. In *The 7th International Conference on Distributed Computing Systems*, pages 10–17. IEEE Computer Society Press, September 1987.

[65] V.J. Rayward-Smith, S.A. Rush, and G.P. McKeown. Efficiency Considerations in the Implementation of Parallel Branch-and-Bound. *Annals of Operations Research*, 43:123–145, 1993.

[66] A. Reinefeld and V. Schnecke. AIDA* - Asynchronous Parallel IDA*. In *Proceeding of the 10th Canadian Conference on Artificial Intelligence (AI '94)*, May 1994.

[67] A. Reinefeld and V. Schnecke. Work-load Balancing in Highly Parallel Depth-First Search. In *Proceeding of the IEEE Scalable High Performance Computing*, pages 773–780, 1994.

[68] T. L. Sergent and B. Berthomieu. Balancing Load Under Large and Fast Load Changes in Distributed Computing Systems - A Case Study. In *The 3rd Joint International Conference on Vector and Parallel Processing*, pages 854–865, September 1994.

[69] B.A. Shirazi, A.R. Hurson, and K.M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[70] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, December 1992.

[71] W. Shu and M. Wu. Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers. *IEEE Transactions on Parallel and Distributed System*, 7(6):637–649, June 1996.

[72] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27:118–141, 1995.

[73] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec 1990.

[74] E. Tarnvik. Dynamo - a portable tool for dynamic load balancing on distributed memory multicomputers. *Concurrency: Practise and Experience*, 6(8):613–639, December 1994.

[75] S. Toledo. Datasheets for MPI on IBM SP2. http://www.dcs.ed.ac.uk/home/fwh/timing/index.html, 1996.

[76] N. Touheed and P.K. Jimack. Dynamic Load-Balancing for Adaptive PDE Solvers with Hierarchical Refinement. In *Proceedings of Eighth SIAM on Parallel Processing for Scientific Computing*, 1997.

[77] A. Trew and G. Wilson. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. Springer-Verlag, 1991.

[78] D. W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20:657–673, April 1994.

[79] J. Watt, M. Rieffel, and S. Taylor. A Load Balancing Technique for Multiphase Computations. In *Proceedings of the High Performance Computing*, pages 15–20, 1997.

[80] M. C. Wikstrom, J. L. Gustafson, and G. M. Prabhu. A Threshold Test for Dynamic Load Balancers. In *International Conference on Parallel Processing*, pages II268–II269, 1991.

[81] M. Willebeek-LeMair and A.P. Reeves. Local vs Global Strategies for Dynamic Load Balancing. In *Proceedings of International Conference on Parallel Processing*, volume 1, pages 569–570, 1990.

[82] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed System*, 4(9):979–993, September 1993.

[83] I-C Wu and H. T. Kung. Communication Complexity of Parallel Divide and Conquer. In *32nd Annual IEEE Conference on Foundations of Computer Science*, October 1991.

[84] M. Y. Wu and W. Shu. The Direct Dimension Exchange Method for Load Balancing in *k*-ary *n*-cubes. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 366–369, October 1996.

[85] C. Xu, F. C. Lau, B. Monien, and R. Luling. Nearest-neighbour Algorithms for Load-balancing in Parallel Computers. *Concurrency: Practice and Experience*, 7(7):707–736, October 1995.

[86] C.Z. Xu and F.C.M. Lau. Analysis of the Generalised Dimension Exchange Method for Dynamic Load Balancing. *Journal of Parallel and Distributed Computing*, 16(4):385–393, December 1992.

[87] C.Z. Xu and F.C.M. Lau. Iterative Dynamic Load Balancing in Multicomputers. *Journal of Operation Research Society*, 45(7):786–796, July 1994.

[88] C.Z. Xu and F.C.M. Lau. The Generalised Dimension Exchange Method for Load Balancing in *k*-ary *n*-Cubes and Variants. *Journal of Parallel and Distributed Computing*, 24:72–85, January 1995.

[89] J. Xu and K. Hwang. Heuristics Methods for Dynamic Load Balancing in a Message-Passing Multicomputer. *Journal of Parallel and Distributed Computing*, 18:1–13, 1993.

[90] Z. Xu and K. Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel & Distributed Technology*, pages 9–23, Spring 1996.