PERFORMANCE OPTIMIZATION FOR DYNAMIC ADAPTIVE GRID HIERARCHIES

BY SIVAPRIYA RAMANATHAN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

New Brunswick, New Jersey

May, 2001

ABSTRACT OF THE THESIS

Performance Optimization for Dynamic Adaptive Grid Hierarchies

by Sivapriya Ramanathan

Thesis Director: Professor Manish Parashar

The accurate solution of many problems in science and engineering require the modeling of unpredictable physical phenomena. Examples include weather forecasting, relativistic modeling of black hole interactions, geophysical modeling of the whole earth, oil reservoir and porous media simulations etc. The storage and computing resource requirements of realistic simulation of these models exceed even the largest and most powerful machines that are available today. Consequently, only parallel and distributed implementations provide a viable solution. An interesting observation of these phenomena is that regions requiring high accuracy are localized, e.g. eye of a cyclone or the event horizon of a balck hole. As a result, additional resolution and correspondingly additional computing and storage resources can be selectively assigned only to these regions. The resulting solutions are termed "adaptive" and result in a more efficient use of scarce resources.

AMR *a*dative mesh refinement techniques provide a means of concentarting resources in adaptive simulations. AMR is especially more efficient than the use of uniform meshes when the solution is changing, much more rapidly in some areas than in others, that is, the nature of the change is dynamic. Parallel and distributed AMR methods have the potential for realistic modeling of tehse physical phenomena. However, they lead to interesting challenges in dynamic resource allocation, data distribution and load balancing, communications and

coordination, and resource management, because of the inherent dynamic nature of the problems.

GrACE is one such distibuted implementation of adaptive solutions to physical phenomena. GrACE provides an object oriented framework for the solution of partial differential equations that are used to model these physical phenomena. The overall goal of this thesis is to optimize the performance of GrACE, to be able to solve very large problems on thousand's of processors.

Specifically, this thesis makes the following contributions:

- A multithreaded communication engine for the GrACE library. The motivation for this was to improve the performance of the library by exploiting the inherent parallelism during the calculations and thus minimize the synchronization overheads by overlapping communication and computations.

- A hierarchical load balancing algorithm. The goal of this was to structure the communications so as to minimize the global synchronizations taking place among the processors during the recompose phase.

Acknowledgements

I would like to thank my research advisor Prof. Manish Parashar not only for his invaluable guidance and support throughout my study as a graduate student at Rutgers, but also for his patience. I would also like to thank Prof. Deborah Silver and Prof. Ivan Marsic for their participation on my committee and their valuable guidance and suggestions regarding my thesis. I would like to thank the CAIP computer facility staff for all their help provided during my stay at Rutgers. I would also like to thank all my colleagues in the lab for all their support, criticisms and help. Last but not the least, I would like to thank my husband Mr. Shankar Krishna , with whose unfailing love and support my graduate studies has become a dream come true.

Table of Contents

Abstract ii

Acknowledgements iv

List of Figures vii

1. Introduction 1

1.1. Overview of thesis 2

1.2. Contributions of the thesis 2

1.3. Organization of the thesis 3

2. Background and Related work 4

2.1. Structured Adaptive Mesh Refinement 4

2.1.1. Structure of the SAMR Grid Hierarchy 4

2.1.2. The AMR Algorithm 5

2.2. GrACE 6

2.3. Distributed AMR Infrastructures 7

2.3.1. BATSRUS 7

2.3.2. PARAMESH 8

2.3.3. SCOREC 8

2.3.4. SAMRAI 9

2.4. Related Multithreading Work 9

2.5. Related Load Balancing Work 9

2.5.1. PaLaBer(Parallel Load Balancer 10

3. Multithreaded Communication Engine 11

3.1. Introduction 11

3.2. Problem Description 12
3.3. A Multithreaded Communication Engine for Dynamic Adaptive Grid Hierarchies 13
3.3.1. *Architecture of the Multithreaded Engine* 15
3.4. Operation of the multithreaded communication engine 18
3.5. Implementation and Experimental Evaluation 19

4. Application level Hierarchical partitioning algorithm 25
4.1. Introduction 25
4.2. Load Balancing Techniques 25
4.3. The hierarchical Scheme 28
4.4. Operation 29
4.5. Operation of the hierarchical scheme 31
4.6. Implementation and Experimental Evaluation 32

5. Discussion and Future Work 36
5.1. Discussion 36
5.2. Future Work 37
References 38

List of Figures

2.1. Grid Hierarchy 5

2.2. HDDA in GrACE to store the grid data 7

2.3. Space filling curves - the Peano-Hilbert curve 8

3.1. Multithreaded communication engine 16

3.2. Sequence of events in the original communication engine 19

3.3. Sequence of events in the multithreaded engine on a single process 20

3.4. Plots of WaveAMR3D application 22

3.5. Plots of WaveAMR3D application with a 257x257x257 grid size 23

3.6. Plots of RM2D application 24

4.1. Centralized load balancing 26

4.2. Distributed load balancing scheme 27

4.3. Structure of the new compute groups - hierarchical scheme 30

4.4. Sequence of events in the original partitioning scheme in GrACE 33

4.5. Sequence of events in the HPA scheme in GrACE 34

4.6. WaveAMR3D application with a grid size=129x129x129 with original and HPA schemes 35

4.7. WaveAMR3D application with a grid size=257x257x257 35

Chapter 1

Introduction

The accurate solution of many problems in science and engineering require the modeling of unpredictable physical phenomena. Examples include weather forecasting, relativistic modeling of black hole interactions, geophysical modeling of the whole earth, oil reservoir and porous media simulations etc. The storage and computing resource requirements of a realistic simulation of these models exceed even the largest and most powerful machines that are available today. Consequently, only parallel and distributed implementations provide a viable solution. An interesting observation of these phenomena is that regions requiring high accuracy are localized, e.g. eye of a cyclone or the event horizon of a balck hole. As a result, additional resolution and correspondingly additional computing and storage resources can be selectively assigned only to these regions. The resulting solution mechanics are termed as "adaptive" and result in a more efficient use of resources.

AMR *A*dative Mesh Refinement[6] techniques provide a means of concentrating resources in adaptive simulations. Parallel and distributed AMR methods have the potential for realistic modeling of these physical phenomena. However, they lead to interesting challenges in dynamic resource allocation, data distribution and load balancing, communications and coordination, and resource management.

AMR is a class of methods for the solution of partial differential equations (PDE) that addresses this problem by performing high-resolution computation only in areas that require it. AMR methods may be structured or unstructured, depending on how they represent the numerical solution to the problem. Unstructured adaptive methods store the solution using graph or tree representation; these methods are called unstructured because connectivity information must be stored for each unknown. Structured AMR employ a hierarchy of nested mesh levels in which each level consists of many simple rectangular grids. Structured

AMR is a mesh based strategy that addresses the above mentioned problem of wasted computer resources by applying grids of a finer resolution only in the regions that require higher resolution, rather than use a uniform mesh with grid points evenly spaced on a domain. AMR strategies have been developed for elliptic, parabolic and hyperbolic systems. The different approaches differ in both philosophy and implementation. Some of the areas of research that AMR has been applied to are : computational fluid dynamics, computational astrophysics, structured dynamics, magnetics, thermal dynamics and many other areas of numerical research.

1.1 Overview of thesis

GrACE (Grid Adaptive Computational Engine)[12] is one framework that supports distibuted implementation of AMR solution technique.GrACE provides an object oriented framework that provides data management support to SAMR. The overall goal of this thesis is to optimize the performance of GrACE, to be able to solve very large problems on thousand's of processors. This thesis presents two optimization of the GrACE AMR library:

1. The design, implementation and evaluation of a multithreaded commuication engine.
2. The design, implementation and evaluation of a hierarchical load balancing algorithm.

1.2 Contributions of the thesis

The thesis makes the following contributions:

- A multithreaded communication engine for the GrACE library. The objective of the engine is to improve the performance of the library by exploiting the inherent parallelism in the calculations on component grids in the AMR hierarchy.This minimizes the synchronization overheads by overlapping communication and computations.
- A hierarchical load balancing algorithm. The objective of this algorithm is to minimize the global synchronizations taking place among the processors during the recompose phase. This is significant as the system increases to hundreds of processors.

1.3 Organization of the thesis

This thesis consists of five chapters organized as follows: Chapter 1 is the introduction. Chapter 2 presents the background and outlines the related work.Chapter 3 explains the design, implementation of the multithreaded communication engine. Chapter 4 explains the design and implementation of the hierarchical load balancing algorithm. Chapter 6 presents some conclusions and outlines future work.

Chapter 2

Background and Related work

This chapter highlights the details of GrACE(Grid Adaptive Computation Engine) on which this thesis is based. It then outlines related work and contrasts it to the presented research.

2.1 Structured Adaptive Mesh Refinement

Dr. Marsha Berger developed a formulation of the adaptive mesh refinement strategy for structured meshes [6] based on the notion of multiple, independently solvable grids, all of which were of identical type, but of different size and shape. The underlying premise of the strategy is that all grids of any resolution that cover a problem domain are equivalent in the sense that given proper boundary information, they can be solved independently by identical means. In this formulation, the multigrid concept is changed, reducing it from a set of computationally expensive set of grids of increasingly finer resolution covering the entire domain, to a set of levels, each of which employs a set of grids of finer resolution to cover only domains of interest.

2.1.1 Structure of the SAMR Grid Hierarchy

The numerical solution to a PDE is obtained by discretizing the problem domain and computing an approximate olution to the PDE at the discrete points. One approach to discretizing is to introduce a structured uniform Cartesian grid. The SAMR(Structured Adaptive Mesh Refinement) grid hierarchy is shown in figure 2.1. The unknown of the PDE are then approximated numerically at each discrete grid point. The resolution of the grid(or grid spacing) determines the local and global error of this approximation, and is typically dictated by the solution-features that need to be resolved. The resolution also determines computational costs and storage requirements.

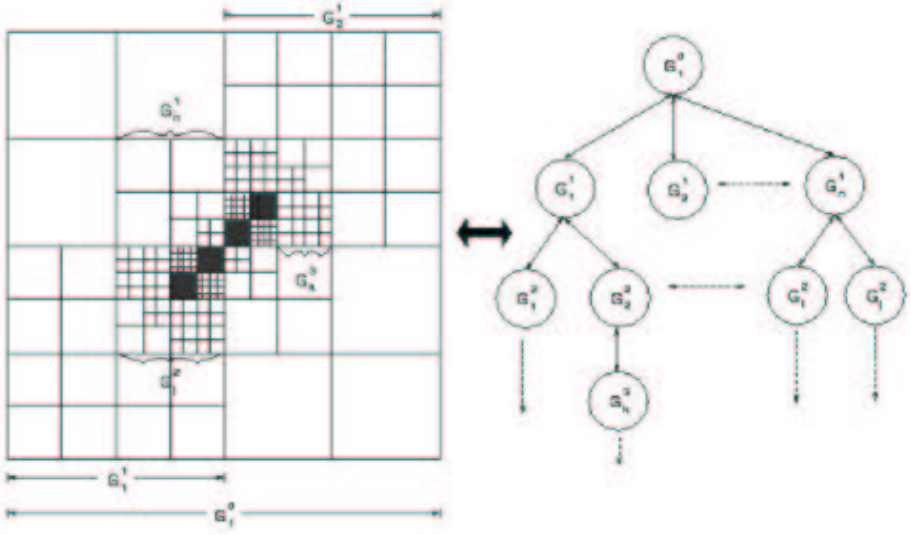


Figure 2.1: Grid Hierarchy

In the case of SAMR methods, dynamic adaptation is achieved by tracking regions in the domain that require higher resolution and dynamically overlaying finer grids on these regions. These techniques start with a base coarse grid with minimum aceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain with high solution error, requiring additional resolution are identified and refined. Refinement proceeds recursively so that the refined regions requiring more resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting structure is a dynamic adaptive grid hierarchy.

2.1.2 The AMR Algorithm

Berger's AMR scheme employs the nested hierarchy of grids to cover the appropriate sub-domain at each level. The integration algorithm recurses through the levels, advancing each level by the appropriate time step, then recursively advancing the next finer level by enough iterations at its (smaller) time step to reach the same physical time as that of the newest solution of the current level. That is, the integrations at each level are recursively interleaved between iterations at coarser levels. Thus, the Berger AMR approach refines in space and if the refinement factor between a finer level (l+1) and the next coarser level is r,

then grids on the finer level (l+1) will be advanced r time steps for every coarser time step. For a d dimensional domain, the grids at level (l+1) must cover the same portion of the computational domain as only $1/r^d$ coarser cells at level l. For example, using a refinement factor of 2 on a three dimensional domain, 2 iterations at level 1 will take more computation time that an iteration at the root level (which comprises the entire computational domain) unless the grids at level 1 cover no more than 1/8 of the domain.

Integration requires four operations:

- Boundary value collection, from parents, siblings and the exterior of the computational domain.
- Evolution, to advance the solution in time.
- Prolongation, to improve the solution values on coarse cells from the overlapping fine cells.
- Refinement, to place grids appropriately for the evolved condition of the solution.

Thus, a more precise expression of the integration algorithm is:

2.2 GrACE

The work presented in this thesis is based on the GrACE [12, 7] infrastructure, which is an approach to distributing AMR grid hierarchies, developed by Dr. Manish Parashar. GrACE is an object-oriented toolkit for the development of parallel and distributed applications based on a family of adaptive mesh-refinement and multigrid techniques. GrACE is built on a "semantically specialized" distributed shared memory substrate that implements a hierarchical distributed dynamic array (HDDA) [8] as shown in Figure 2.2.

HDDA provides uniform array access to heterogeneous dynamic objects spanning distributed address spaces and multiple storage types. The array is hierarchical in that each element of the array can be an array; it is dynamic in that the array can grow and shrink at run-time. Communication, synchronization and consistency of HDDA objects are transparently managed for the user. Distribution of the HDDA is achieved by partitioning its

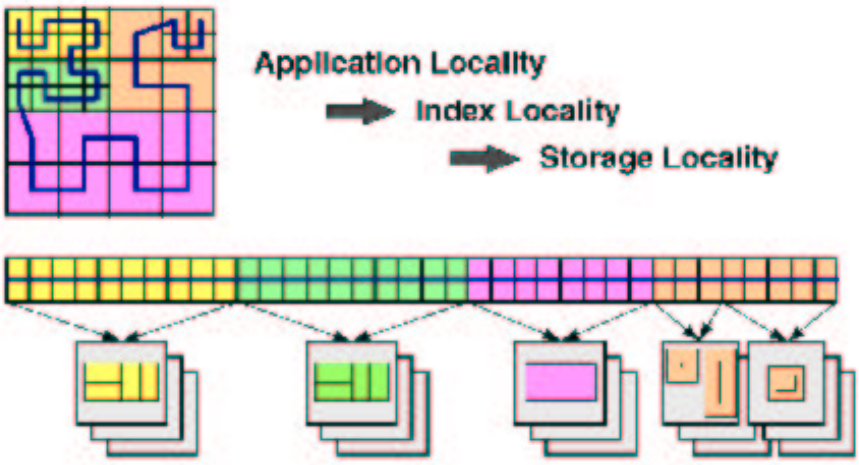


Figure 2.2: HDDA in GrACE to store the grid data

array index space across the processors. The index-space is directly derived from the application domain using locality preserving space-filling mapping [5, 4] which efficiently map N-dimensional space to 1-D dimensional space (see figure 2.3).

2.3 Distributed AMR Infrastructures

There already exists wide spectrum of software systems that support parallel and distributed implementations of AMR applications. Each system represents a unique combination of design decisions in terms of algorithms, data-structures, decomposition, mapping and distribution mechanism, and communication mechanism. This section explains a few of the existing AMR systems.

2.3.1 BATSRUS

BATSRUS [14] is implemented in FORTRAN90, using a block-based domain-decomposition approach. Blocks of cell (stored as 3D F90 arrays) are locally stored on each processor so as to achieve a reasonable balanced load. The application starts out with a pool of processors, some of which possibly unused. Every utilized processor has a block of equal memory size, but possibly at a different resolution and/or a different sized partition of physical space. As

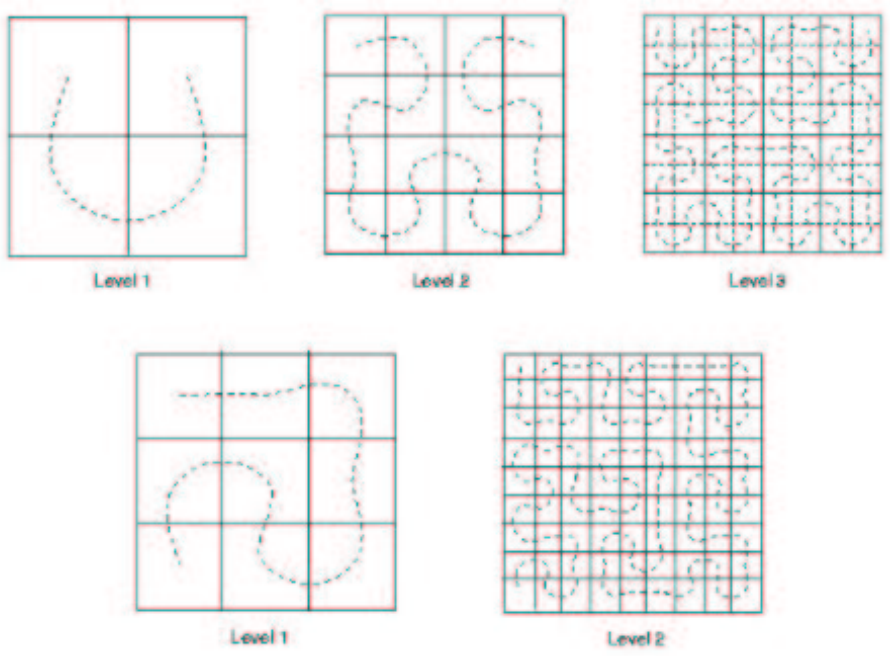


Figure 2.3: Space filling curves - the Peano-Hilbert curve

the application adapts and new (adapted) grids are created, these are allocated, in units of the same fixed block size to the unused processors. No more refinement can occur once all the virtual processors are used up.

2.3.2 PARAMESH

PARAMESH [10] is another FORTRAN 90 package designed to provide an application developer with an easy route to extend an existing serial code which uses a logically cartesian structured mesh into a parallel code with adaptive mesh refinement (AMR). The PARAMESH distribution strategy is based on partitioning a hierarchical tree representation of the adaptive grid structure.

2.3.3 SCOREC

SCOREC Parallel Mesh Databases [11] provides a generic mesh database for the topological, geometric and classification information that describes a finite element mesh. The database supports meshes of non-manifold models and multiple meshes on a single model or multiple

models. Operators are provided to retrieve, store and modify the information stored in the database. Parallel Mesh Database (PMDB) provides extensions to the SCOREC Mesh Database to create and manipulate meshes in a distributed memory environment. PMDB provides three static partitioning procedures for initial mesh distribution, three dynamic load-balancing schemes and mesh migration operators.

2.3.4 SAMRAI

SAMRAI [15] is an object-oriented framework that provides computational scientists with general and extensible software support for the prototyping and development of parallel structured adaptive mesh refinement applications. SAMRAI makes extensive use of object-oriented techniques and various design patterns, such as Abstract Factory, Strategy, and Chain of Responsibility.

2.4 Related Multithreading Work

These infrastructures do not employ mult-threaded runtime support. Multithreaded runtime for AMR infrastructures has been researched by other Nikos Chrisochoides at Cornell University [1], and Edward W. Felten and Dylan McNamee from University of Washington [2]. Chrisochoides' work on multithreading employs threads for load balancing. All processors start with a pool of threads. Threads can be interior threads or interface (boundary) threads. The thread scheduler schedules these threads in so as to minimize the overheads of communication. "New threads" is another thread library from university of Washington that can be used to improve the performance of general message passing applications. The library provides communication support between threads on different processors by using globally unique port numbers.

2.5 Related Load Balancing Work

Load balancing and partitioning is well addressed problem in the field of parallel and distributed computing. The systems described above do not use the hierarchical approach for load balancing. All the original work in hierarchical approach were at the operating system

or middleware level load balancing. Recent focus has turned into application sensitive load balancing, a system which adapts to the dynamics of the application in execution.

2.5.1 PaLaBer(Parallel Load Balancer

This is a scalable hierarchical dynamic load balancing from IPVR(Institute of Parallel and Distributed High-Performance Systems). This system is implemented on Intel Paragon XP/S, and uses multilevel control for dynamic load balancing as well as for the communication manager. The control tree consists of three components:

- the root component monitors the systemand starts an application process if the system is underloaded
- inner component monitors the leaf components under its control and is the communication link between the root and the leaves.
- the leaf component work as autonomous units

The hierarchical load balancer uses non-premptive as well as premptive process migration to balance load between the nodes.

Chapter 3

Multithreaded Communication Engine

This chapter presents the design and implementation of a multithreaded communcation engine to enable scalable implementations of distributed adaptive mesh-refinement(AMR) applications. The primary motivation is to manage the computational heterogeneity inherent in this class of applications and to exploit the multiple granularities and levels of parallelism offered by the applications.

3.1 Introduction

Adaptive numerical methods dynamically focus computational resources, such as CPU cycles and memory, only to regions them; thus they can achieve better accuracy for the same computational resources as compared to non-adaptive methods. To be effective, the gains achieved through selective/adaptive refinement must outweigh the overheads associated with adaptivity, such as error estimation and data structure management. Adaptive mesh methods are difficult to implement because they rely on dynamic, complicated data structures with irregular communication patterns.

While distributed implementations of adaptive methods offer the potential for accurate solution of physically realistic models of important physical systems, these distributed implementations lead to many interesting challenges in dynamic resource allocation across processors, data-distribution and load balancing, interprocess communications and coordination, and resource management. The overall efficiency of the algorithms is limited by the ability to manage the underlying data-structures at run time so as to expose all inherent parallelism, minimize communication/synchronization overheads, and balance load. Because adaptive mesh applications change in response to the dynamics of the problem, little can be known about the structure of the computation at compile time.

AMR grids are inherently heterogeneous varying in both resolution and extent. Furthermore, these grids can be created, moved and deleted on the fly. AMR applications can offer multiple levels and granularities of parallelism. Grids at the same level of refinement can be operated on in parallel. Similarly composite slices across all refinement levels(i.e. a parent grid and all its children) can also be operated on in parallel. Finally, each grid can itself be operated on in a dataparallel fashion. The the AMR algorithm requires that each grid be periodically synchronized with its parents and its neighbouring siblings requiring communication at regular intervals. Clearly, there is a need for a runtime communication engine that can exploit these many levels and granularities of parallelism, and efficiently manage and overlapping the synchronizations and communications with computations.

In this chapter we present the design, implementation, and evaluation of such a multithreaded communication engine that addresses the issues listed above and support parallel/distributed AMR applications. The engine uses the MPI communication library (to ensure portability) and implements the capability for registering message handlers at the application level (similar in principal to Active Messages). These handlers enable each computational thread to provide a handler functions that defines how the particular class of messages have to be processed and how the thread is to be informed about message arrivals. Communications threads can now independently manage all communications and maximize their overlap with computations. The multithreaded communication engine has been built on the GrACE SAMR library.

3.2 Problem Description

Adaptive mesh algorithms communicate information about the numerical solution between levels of the hierarchy and also among grids at the same level of the hierarchy. Around the boundary of each grid patch is a *ghost cell* region which locally caches data from adjacent grids or, where no neighboring grids exist, from the next coarser level of the hierarchy. The size of the ghost regions is defined by the stencil spacing and is application dependent. The management of these bookkeeping details can be a daunting task because of the irregular and unpredictable placement of the refinement areas.

Distribution of adaptive methods based on hierarchical AMR consists of appropriately partitioning the adaptive grid hierarchy across available computing nodes, and concurrenlty operation on the local portions of this domain. Parallel AMR applications require two primary types of communications:

- **Inter-grid Communications** :Inter-grid communications are defined between component grids at different levels of the grid hierarchy and consist of prolongations(coarse to fine transfers) and restrictions(fine to coarse transfers). These communications typically require a gather/scatter type operations based on an interpolation or averaging stencil. Inter-grid communications can lead to serialization bottlenecks for naive decompositions of the grid hierarchy.

- **Intra-grid Communications** : Intra-grid communications are required to update the grid elements along the boundaries of local portions of a distributed grid. These communications consist of near-neighbor exchanges on the stencil defined by the difference operator. Intra-grid communications are regular and can be scheduled so as to overlap with computations on the interior region of the local portions of a distributed grid.

3.3 A Multithreaded Communication Engine for Dynamic Adaptive Grid Hierarchies

Parallel/distributed implementations of adaptive mesh refinement techniques for solving PDEs typically consist of three phases (a) computation phase, (b) load balancing phase and (c) data-migration phase. The computation phase is again sub-divided into a pure computation phase and the ghost synchronization phase. The ghost synchronization phase involves the exchange of ghost or boundary regions that are shared between processors participating in the computation. Global synchronization barriers ensure that all the processors reach the load balancing and data-migration phases at the same time. The underlying assumption here is that processes have a global view of the computation domain. Ghost synchronization is needed in order that all processors have the right data and to ensure that the solution converges. The ghost message exchanges are very expensive operations

since they involve message passing over the network. The requirement that these exchanges occur often enough for the solution to converge affects performance heavily. One solution is to use an overlap to alleviate the cost of communication. This can be achieved by using the multithreaded engine.

This section presents an architecture for a multithreaded communication engine for the GrACE SAMR library. GrACE is an object-oriented toolkit for the development of parallel and distributed applications based on a family of adaptive mesh-refinement and multigrid techniques. GrACE is built on a "semantically specialized" distributed shared memory substrate that implements a hierarchical distributed dynamic array (HDDA)[8, 7]. HDDA provides uniform array access to heterogeneous dynamic objects spanning distributed address spaces and multiple storage types. The array is hierarchical in that each element of the array can be an array; it is dynamic in that the array can grow and shrink at run-time. Communication, synchronization and consistency of HDDA objects are transparently managed for the user. Distribution of the HDDA is achieved by partitioning its array index space across the processors. The index-space is directly derived from the application domain using locality-preserving space filling mappings [5, 4]that efficiently map N-dimensional space to 1-D dimensional space.

In order to minimize the overheads of ghost synchronization, the GrACE library employs the following optimizations: At the beginning of the computation phase, the processes participating in the computation, calculate the boundary regions shared with other processes, anticipate the messages to be received from the neighboring processes and register a handler for the expected messages. During the synchronization phase, the processes exchange data. The messages are shipped in the form of HDDA objects or buckets consisting of a header and a payload consisting of the actual data much like an IP packet in the Internet. The header contains information of how to handle the message at the receiver. When the message arrives at the receiving end, the information from the header is extracted and the message processed accordingly. The received data is then copied into the appropriate message buffers using the information from the message header. The data present in the message buffers are then copied into the application space when the data is needed by the computation. This improves performance by eliminating the need to poll for messages to

arrive.

The computation on the initial domain leads to adaptive refinements in the regions requiring additional resolution. These adaptive refinements change over time and give rise to heterogeneous grid structures across different levels, which have heterogeneous computational requirements. The additional complexity arises when these heterogeneous grid blocks are partitioned and distributed across processors. The ghost synchronizations are performed cyclically, and are regular in nature. The single-threaded implementation suffered from excessive waiting times during these ghost synchronization phases. This motivated the need for developing a multithreaded communication model to reduce the latencies in communication by overlapping computation with communications.

3.3.1 *Architecture of the Multithreaded Engine*

In applications using GrACE, the processors performing the computations typically own a number of grid blocks per level. The inherent parallelism present in the heterogeniety of the underlying grid blocks, both in the number, size and shape, can exploited for multitasking. The computed block can be sent out and the messages arriving for that block can be copied into, as the computation is proceeding on the other blocks owned by the processor. The multithreaded engine developed for GrACE attempts to exploit this scenario to improve performance of the scientific applications using the library. The objective was to develop the threaded communication engine with minimum modifications to the existing code. The other goal was that this also should be portable to different architectures that GrACE had been ported to.

The threading model used in the multithreaded communication engine is the producer-consumer model, as opposed event-driven models which are used in applications where the occurence of events is asynchronous. We chose this model against an event-driven one because, the synchronizations and computations typically occur in lock-step. As the computations and these ghost synchronizations are tightly coupled one operation cannot proceed without the other being completed. The modified messaging infrastructure consists of three threads per process: the main computation thread, the send thread and the receive thread as shown in 3.1. The main thread shares a FIFO queue with each of the send and

receive threads.The main thread is the producer and the consumers are the send and receive threads.

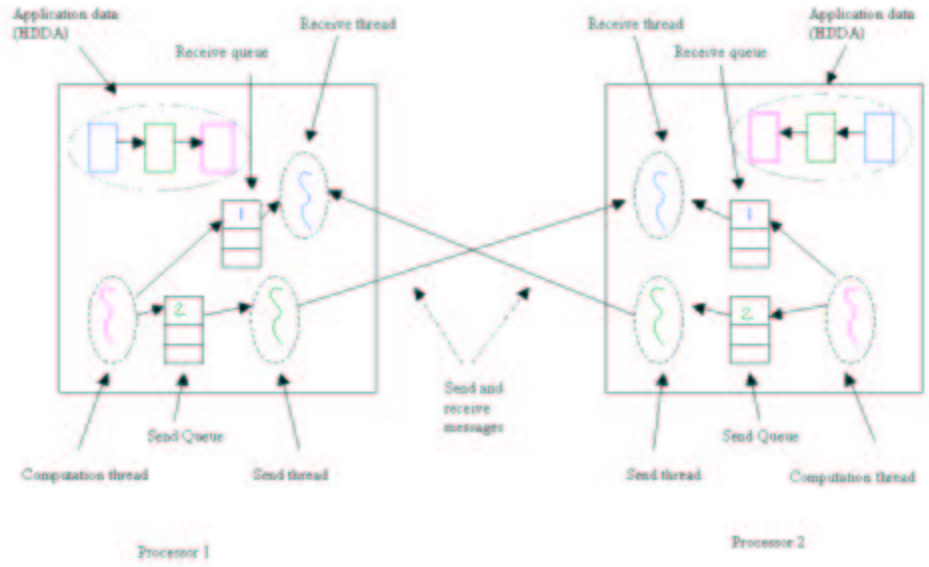


Figure 3.1: Multithreaded communication engine

Given below is a summary of the functions of the three threads that come to play in the modified messaging system.

Computation thread : The compute thread spawns the send and receive threads at thebeginning of the computation phase. As the computation proceeds on the grid blocks owned by the process, the threads are signaled to start. The block number on which computation is done and whose ghost regions are ready to be shipped out is entered in the threads' queues. The compute thread then waits for both the threads to finish, before beginning the next compuatation phase. This cycle continues until the end of computation. the threads then join when the computation comes to an end. The computation or main thread has the following functions:

- Setting up the grid hierarchy and the necessary grid functions as required by the application.
- Initialize the data structures, calculate and create message handlers for messages.

- Perform computation, load balancing, load distribution.
- Create the send and receive threads and initialize their data structures.

Send Thread : The send thread has a send queue where the pending requests of the blocks to be sent are stored. The queue is a FIFO(First in First out) queue and requests are processed in the order. The send thread has the following functions:

- Wait for a signal from the main thread signalling the start of the synchronization phase.
- Process the requests or blocks in the order entered in the send queue.
- Send out the blocks to processes.
- Signal the main thread when the sends are complete.

Receive Thread : The receive thread also has receive queue, where the main thread enters the block number/id of the block on which the computation is done. Thus the received message can be copied on to the application space. The receive queue is also a FIFO queue. The receive thread thus performs the following functions:

- Wait for signal from the main thread signalling the start of the synchronization phase.
- Receive messages that have arrived and copy them into the appropritae message buffers.
- Copy the data from message buffers to the grid blocks on which computation has completed.
- Signal the main thread when all the messages have been received and have been copied into the appropriate grid blocks.

The data item exchanged between the main and send and receive threads is a self contained structure that contains details of the level, timestep, block number and grid function id of the message that has to be sent out. It also contains a synchronization flag that serves to distiguish between two different synchronization phases occuring at different time steps. The code segment that follows makes these clearer.

```
struct threadItem {
   int time;     // timestep at which grid hierarchy is to be sync'd
   int level;    // level of the grid hierarchy to be sync'd
   int gfId;     // grid function id
   int blockId;  // grid number on which computation is done
   int syncFlag; // flag to indicate end of one sync phase
}
```

The data item exchanged between the threads is a structure instead of just the block number. This enables the threaded engine to overcome processor speed differences. For instance, a faster processor maybe in the process of sending messages for time t2, level 2 and gfid 2 whereas the receiving processor may still be in the process of computing and sending out blocks for time t1, level 1 and gfid 1.

3.4 Operation of the multithreaded communication engine

The original messaging model consisted of a single computation thread that did both the computation and communication. The sequence of events in the original messaging model is illustrated below:

As stated above, simulations occur in phases - computation phase and ghost synchronization phase. Ghost synchronizations involve exchanging the boundary regions of the grid contained with neighboring processors. Thus the sends involve the steps of copying the data from application buffers to the message buffers, packing them and sending them out. On the receiving end, the received messages are unpacked and then copied into the application data structure.

The modified messaging system takes advantage of the parallelism among the diiferent grid blocks owned by the processor. The new sequence of steps involved in this modified messaging are explained below and illustrated by means of the diagram.

At the start of the simulation, the main thread performs the initializations on the grid hierarchy and then spawns off the send and receive threads. The data structures are then set up for communication with the main thread. When the computation phase starts, the main

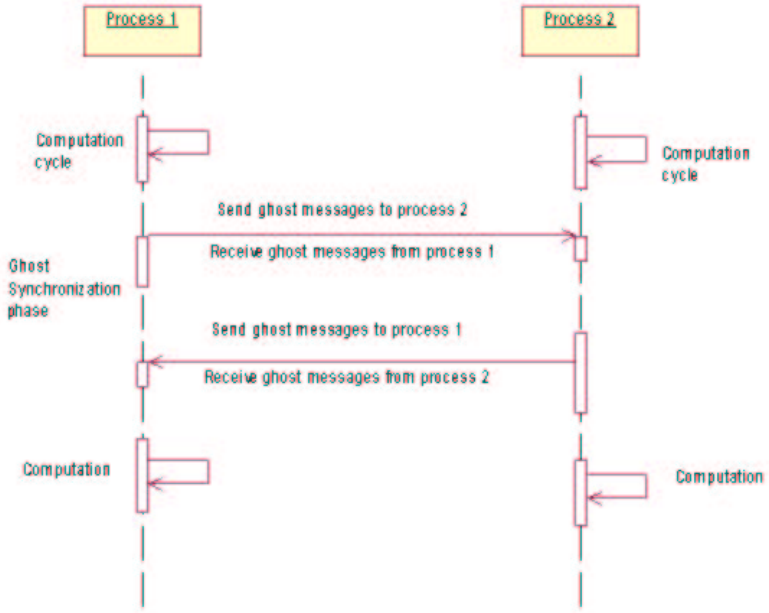


Figure 3.2: Sequence of events in the original communication engine

threads signals the start of the synchronization phase to both the send and receive threads. As soon as computation on a block is computed, the block number is put on the tail end of the send and receive queues. The threads remove the items from the head of the queue and process the blocks accordingly. As soon as a synchronization phase is complete, the two threads signal the main thread to start the next cycle of computation and synchronization. When the simulation is complete, the threads join the main computation thread and exit.

3.5 Implementation and Experimental Evaluation

The multithreaded engine was developed and tested on the Sun Enterprise1000 cluster. Each system is configured with sixty-four 400 MHz SPARC processors, 32GB of RAM, and approximately a terabyte of disk. Each system supports 16 processor boards with 4 processors per board. Each processor utilizes 4Mbytes of L2 cache. The communication infrastructure uses the POSIX [13, 3] library for creating and scheduling threads and was built using the thread safe MPI [16, 9] implementation available on the Sun E10K processors.

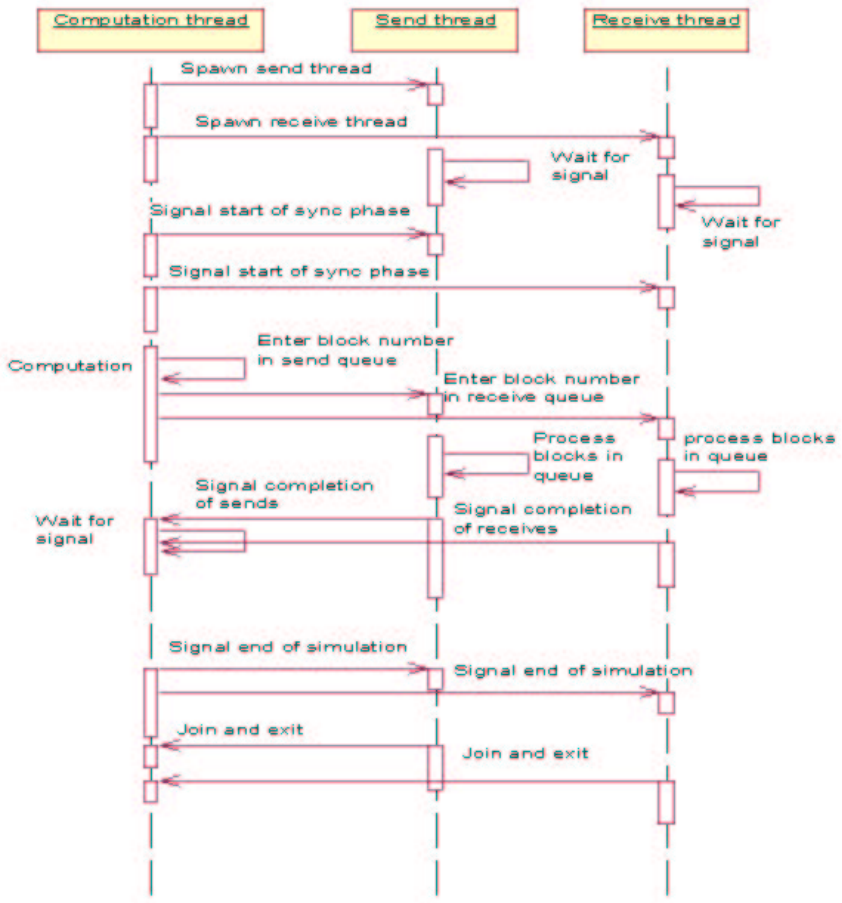


Figure 3.3: Sequence of events in the multithreaded engine on a single process

The applications used in these experiments belong to the general class of AMR applications.

1. RM2d - RM2D is a 2D compressible turbulence application solving the Richtmyer-Meshkov instability. This application is part of the virtual test facility developed at the ASCI/ASAP center at the California Institute of Technology. The Richtmyer-Meshkov instability is a fingering instability, which occurs at a material interface accelerated by a shock wave. This instability plays an important role in studies of supernova and inertial confinement fusion (ICF).

2. AMR3D - AMR3D is a 3D application in computational fluid dynamics that addresses

the forward facing step problem, describing what happens when a step is instantaneously risen in a supersonic flow. The application/simulation has several features including bow shock, Mach stem, contact discontinuity, and a numerical boundary. AMR3D is also part of virtual test facility developed at the ASCI/ASAP center at the California Institute of Technology.

The experiments conducted on the multithreaded communication engine measured the total execution time with and without the threaded communication engine. The plots shown in the graphs below measure the total execution time in seconds. The 3-dimensional wave application (Wave3d) seems to have gained in performance using the threaded engine. Specifically the runs on 4 and 8 processors seem to have gained a larger percentage of improvement over the original code. As the number of processors increased, the percentage gain seems to level off. This can be attributed to the fact that, given a fixed domain size, as the number of processors is increased, the number of grid blocks owned per processor is decreased. The threaded engine exploits the parallelism available during the computation on a number of grid blocks owned by a processor. As the number of blocks owned decreases, so does the potential parallelism available. As a result, the performance does not gain significantly.

The 2-dimensional application RM2d did not show much promise with the multithreaded engine. The results showed that performance did not improve with the use of the multithreaded communication engine. This can be attributed to the fact that a 2-dimensional grid offers lesser room for parallelism as the computation that has to be performed on a 2-d grid is also much lesser.

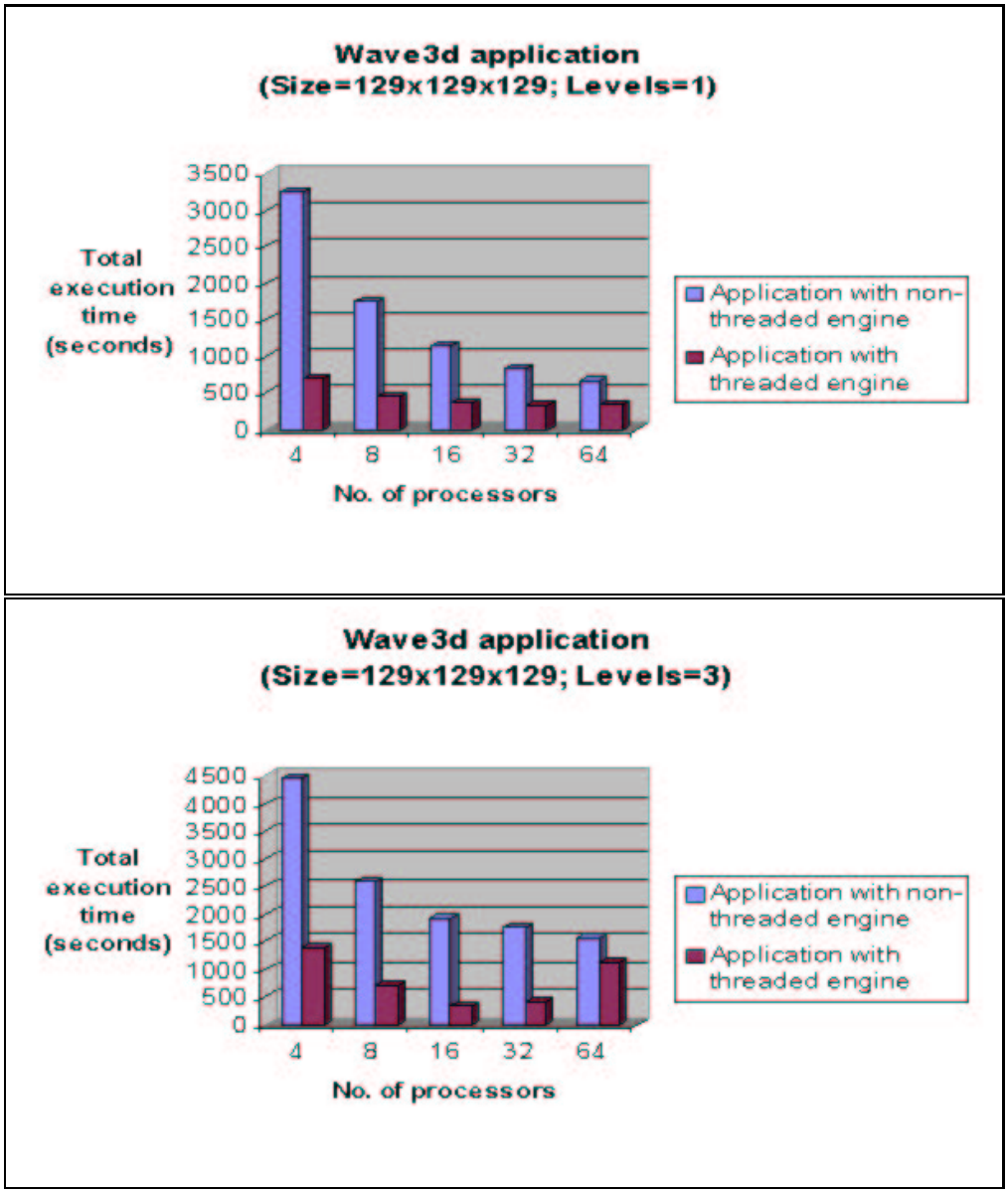


Figure 3.4: Plots of WaveAMR3D application

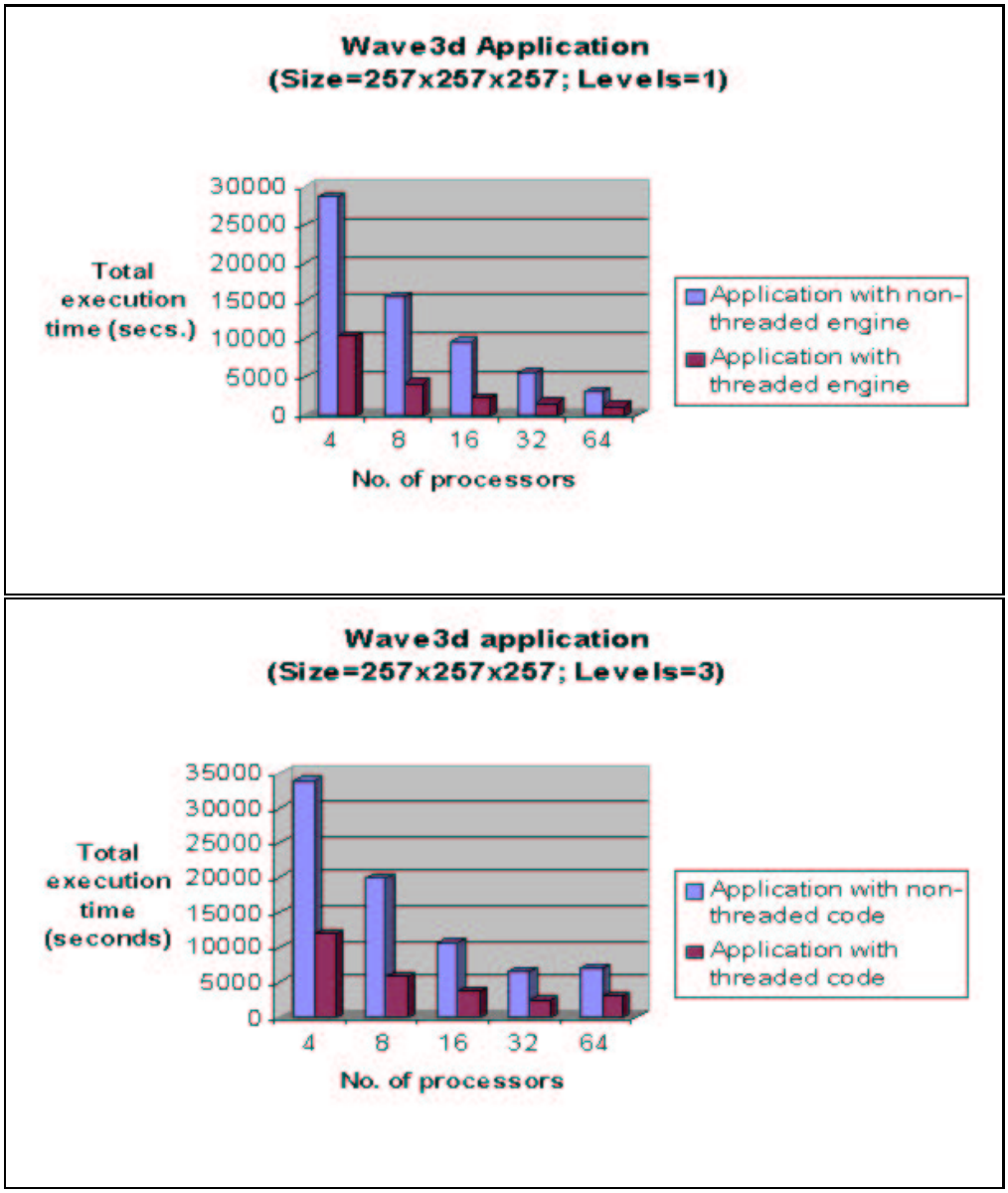


Figure 3.5: Plots of WaveAMR3D application with a 257x257x257 grid size

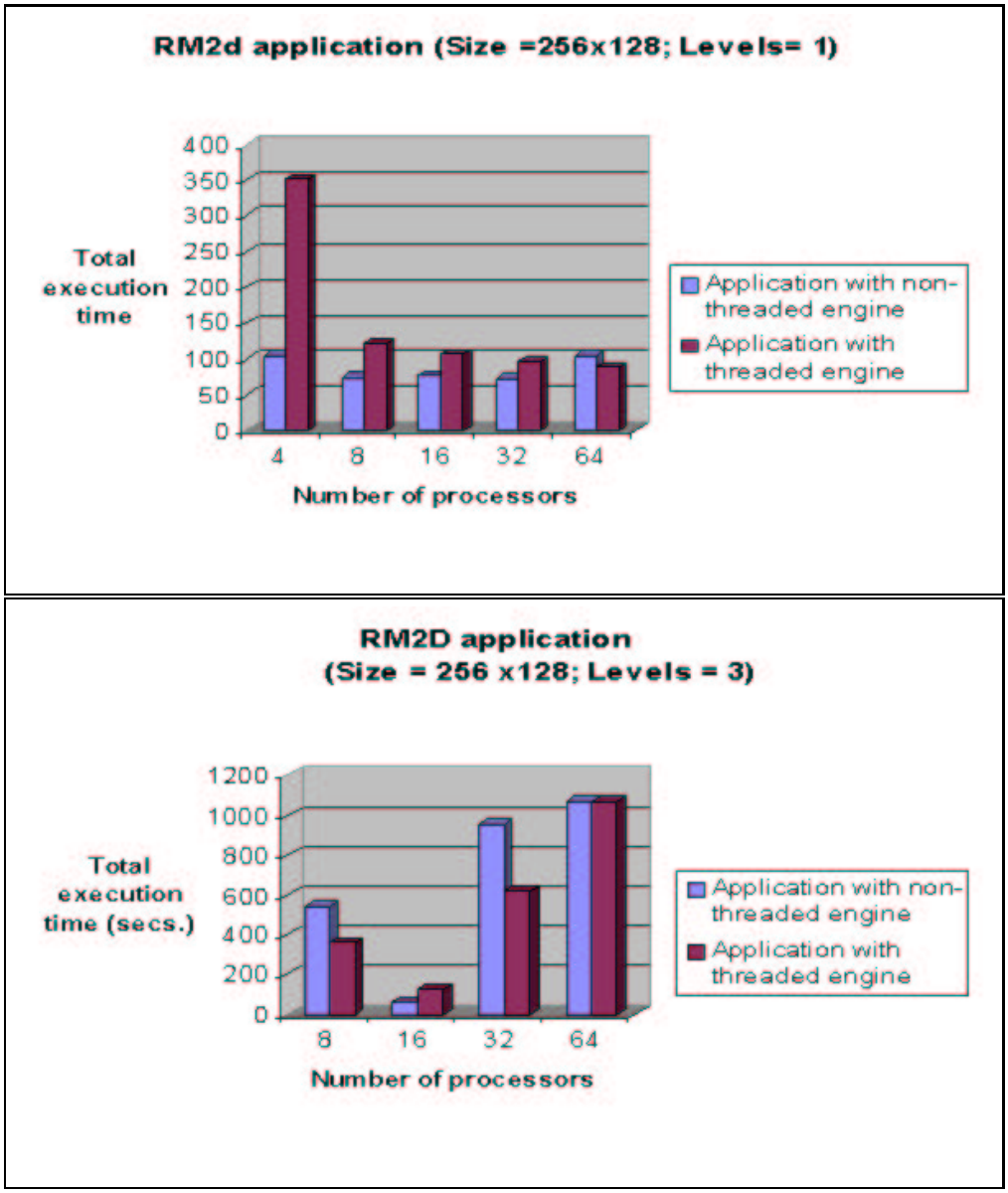


Figure 3.6: Plots of RM2D application

Chapter 4

Application level Hierarchical partitioning algorithm

4.1 Introduction

As a result of the evolutionary advancements in computing and communications technology infrastructure, parallel and distributed systems have become more and more popular as supercomputing environments during the last decade, but nevertheless they are still difficult to use. One of the reasons why these systems are so difficult to use is the problem of dynamic load balancing. In order to make optimum use of a distributed system, the workload should be distributed equally among all available nodes in the system. This can be done using static load balancing scheme which is based on a priori knowledge of the problem structure as well as the runtime behaviour of the application. However, for a large class of parallel applications, the runtime behaviour is not known in advance. For these problems, the workload is created dynamically during the execution and/or the runtime behavior of the processes dramatically uring the execution time of the application. Such problems need a dynamic load balancing mechanism to redistribute the work among the system nodes.

4.2 Load Balancing Techniques

In general, there are two approaches to dynamic load balancing-

- **Application-level load balancing**: Scheduling decisions are taken at the application level. These typically involve migrating data among processes. This approach can be optimised by taking advantage of the knowledge about the application and its runtime behavior.
- **Application-independent load balancing**: Scheduling decisions are taken by the operating system. This approach is more general and transparent to the application.

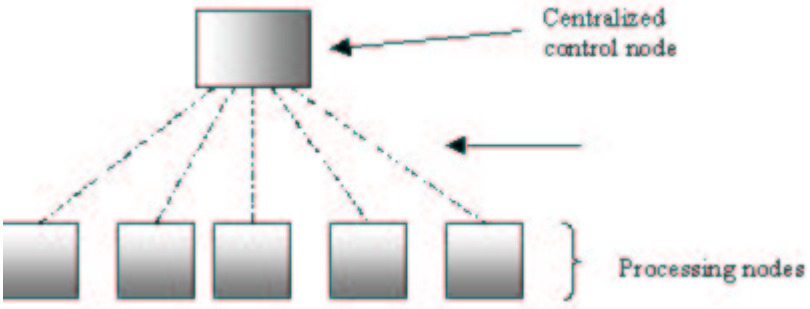


Figure 4.1: Centralized load balancing

Each of the above methods can again be subdivided into-

- **Centralized load balancing** - Responsibility for the task of dynamic scheduling physically resides in a single node(see fig. 4.1. This scheme has several advantages -
  - It yields optimal load balance and performance as the central load balancer has a global view.
  - It is easy to realize as there is a single flow of control.

  The disadvantages of this scheme are -
  - This scheme does not scale and hence it is suitable for large parallel and distributed systems.
  - The accumulation of the global load information becomes a formidable task.

- **Distributed load balancing** - Responsibility for the task scheduling is physically distributed among the nodes of the system (see fig. 4.2). This schemes has the following advantages over the centralized scheme:
  - This method scales well and hence can be used in large parallel and distributed systems.
  - This method operates on a local view of the domain and hence the step of global synchronization is avoided. This leads to faster operation times.

  The shortcomings of this method are -

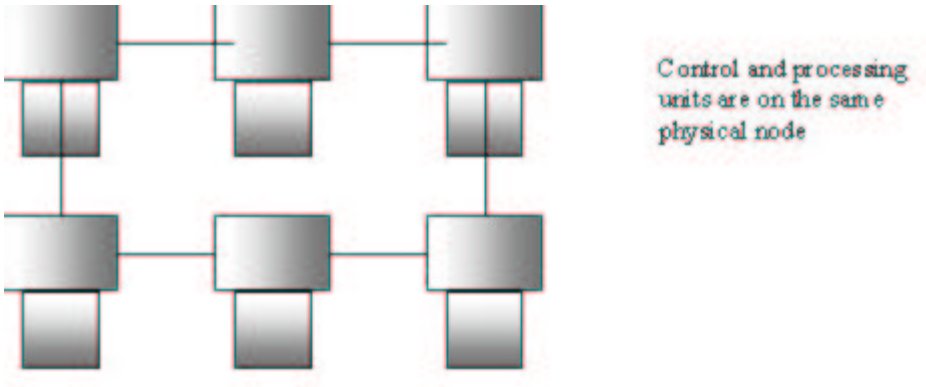


Figure 4.2: Distributed load balancing scheme

- Optimal scheduling decisions are difficult or impossible to make as each node in the system operates autonomously with a local view of the domain or system state.
- The decision making algorithms are complicated and this makes a fully distributed system dificult to implement and realize.

Application-level load balancing techniques can also be broadly divided into static and dynamic techniques.

- Static partitioning techniques are used when the grid is partitionied only once (or very few imes) and there is no dynamic redistribution involved. In this case, the initial partitioning is maintained through the execution of the application. Static techniques tend to focus on the partitioning quality rather than partitioning speed.
- Dynamic partitioning routines are used by adaptive applications (such as SAMR) to repartition and redistribute the dynamic grid structure at runtime. In addition, these techniques have to minimize data movement overheads and partitioning time, as grid adaptations occur at regular intervals. Consequently, partitioning quality is often sacrificed for speed and efficiency by these partitioners.

Static and dynamic partitioners can be further sub-divided into geometric and graph-based techniques. Geometric techniques take the geometry of the grids into account. This category

includes binary dissection, ISP, and geometric mesh partitioning. Graph-based techniques use a graph representation of the problem domain and partition this graph. This category includes recursive spectral bsection and the multilevel algorithms in the software partitioning library Metis and ParMETIS. Graph-based techniques can also be used for partitioning domains where geometry plays an important role. In such cases, geometrical information is coded into the graph used as the imput to the partitioner.

Dynamic partitioning/load-balancing techniques may be global and local. Global techniques maintain a global view of the problem domain and use this global information to partition the domain. Global techniques include space-filling curve (SFC) partitioners and diffusion schemes based on global work load. While global techniques lead to a better and more balanced distribution, global synchronization required can make them expensive.

4.3 The hierarchical Scheme

This section gives a little background on the partitioning schemes currently used in GrACE. It then explains the shortcomings of the present scheme and hence the motivation for a new scheme.

The overall efficiency of parallel/distributed SAMR applications is limited by the ability to partition the underlying grid hierarchies at runtime to expose all inherent parallelism, minimize communication and sychronization overheads, and balance load. Given the different classifications described above, the partitioning system of GrACE can be classified as a dynamic, global, domain-based partitioning scheme. The system is also a hybrid of the centralized and the distributed methods of partitioning. Here the task of scheduling/balancing the workloads is collectively done by all the participating nodes of the system, but with all the nodes having a global knowledge of the total workload.

The partitioning scheme thus has all the advantages of a global domain-based partitioner, namely better and more balanced distribution, and better scalability. The disadvantage is that the collection of the global information requires global synchronization which make this method expensive and hence lower performance. The load partitioning phase consists of the following steps:

1. Global synchronization of all nodes participating in the computation.

2. Load information exchange. After this step, all the nodes have a global view of the grid hierarchy.

3. Load partitioning phase - All nodes calculate the average load and partition the grid hierarchy. This is a collective operation as the progarmming model used is SPMD.

Simulations (using GrACE) on large number of processors showed that the global synchronization and information exchange phase became a performance bottleneck. To improve this communication bottleneck, a better scheme or communication pattern had to be designed that would minimize the communication overhead. The method proposed in this thesis does not propose a partitioning scheme *per se*, but a way of structuring the steps during the load partitioning/balancing phase that minimizes the communication overheads incurred by the global synchronization phase. The hierarchical scheme proposed here tries to minimize the barrier synchronizations at the beginning of every load balancing phase. The communications are done in stages among a few processors at a time rather than an "all-process" barrier. This is done by means of dividing the processes/processors into "compute groups" with one "master" node per group. The master node is the gateway of communication with other groups. The group size is a programmable parameter and is given at the start of the simulation.

4.4 Operation

In the light of the newly defined compute groups,the new communication scheme now divides the partitioning phase in GrACE into sub-phases. This is done to minimize the number of processors synchronizing at the same time, and hence the global synchronization overhead.

The proposed scheme has two partitioning phases:

- Local partitioning phase: The processors belonging to a *compute group* do a partitioning within the compute group based on a local load threshold. This step is the same as the original load balancing in GrACE.

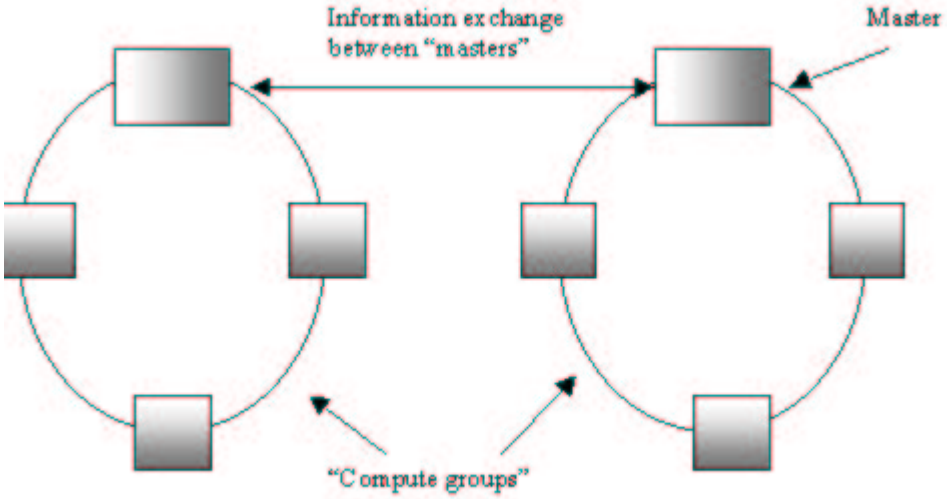


Figure 4.3: Structure of the new compute groups - hierarchical scheme

- Global partitioning phase: The *master* nodes then decide if a global partitioning has to be done based on a global threshold.

Given below is a pseudocode of the new load balacing phases:

```
load_balancing phase:
if(my_load > threshold) {
   do a local partition;
}
if(master) {
   if(group_load > global threshold) {
do a partition among masters;
   }
}
broadcast new global list;
if(global_partition) {
   do local partition with new information;
}
begin computation;
```

.

.

The hierarchical partitioning scheme attempts to exploit the fact that given a group with adequate number of proceesors, and a carefully defined group size, the number of global partitioning phases will be minimized. This will effectively minimize the barrier synchronization phases. The processors in the "compute groups" are automatically synchronized with the other groups when the master nodes synchronize to exchanged global information.

4.5 Operation of the hierarchical scheme

The sequence of steps taking place in the original GrACE library for partitioning and scheduling ghost communications is described below and illustrated by means of the sequence diagram (see figure 4.4). At the beginning of the simulation, all the processors have the initial domain and that is partitioned by them. This enables the processors to schedule and post receives for the ghost communications that is done after every iteration. During the load balance phase, all the processors synchronize and exchange their local domain information. At the end of this phase, every processor has a consistent global view of the domain. The partitioning algorithm then partitions the domain among the processors. The processors then migrate data that no longer belongs to local domain that is now owned by them. The processors then schedule ghost communications based on the new local domain and post the receives.

The hierarchical scheme on the other hand creates processor groups. After the groups are created and the initial grid hierarchy is setup, the "master" nodes partition the initial domain, which is the global partitioning phase. At the end of this phase the masters have a portion of the domain that is then partitioned among the processors in the group. This is similar to the partitioning sequence in GrACE where all the processors syncronize and then partition. This is the local partitioning phase. After this phase, the processors calculate and schedule ghost communications which might be even across different processor groups. This is illustrated by means of the sequence diagram in figure 4.5. This scheme exploits the fact that with the right number of processor groups and the right partitioning scheme

to get the initial load, the partitions can be just restricted local partitioning in majority of the cases, without having to do the global partition. Even when the global partition does occur, the costs of partitioning are amortised by the partitioning itself occurring in phases.

4.6 Implementation and Experimental Evaluation

The hiearchical partitioning scheme was implemented as part of the GrACE library. The groups were created using the MPI library which provides primitives to create groups. The communcation within groups was done with the help of intercommunciators and the communication between processors belonging to different groups is done with the help of intercommunicators provided by MPI. The scheme was evaluated on an IBM cluster.

The application used in these experiments belongs to the general class of AMR applications.AMR3D is a 3D application in computational fluid dynamics that addresses the forward facing step problem, describing what happens when a step is instantaneously risen in a supersonic flow. The application/simulation has several features including bow shock, Mach stem, contact discontinuity, and a numerical boundary. AMR3D is also part of virtual test facility developed at the ASCI/ASAP center at the California Institute of Technology.

The experiments measured the total execution time taken by the simulations to complete with and without the hierarchical scheme. The measurements were taken to evaluate if the scheme had any performance impact on a simulation with smaller number of processors. The plots given below show the total execution time plotted against the number of processors.The experiments validated that the HPA scheme has minimal impact on performance in case of simulations on smaller number of processors. ON the contrary, it even improved performance.In the larger runs, namely the 128 and 256 processor runs, we see that the number of groups plays an important role. As the number of groups is increased from four to eight, the performane increases as expected.

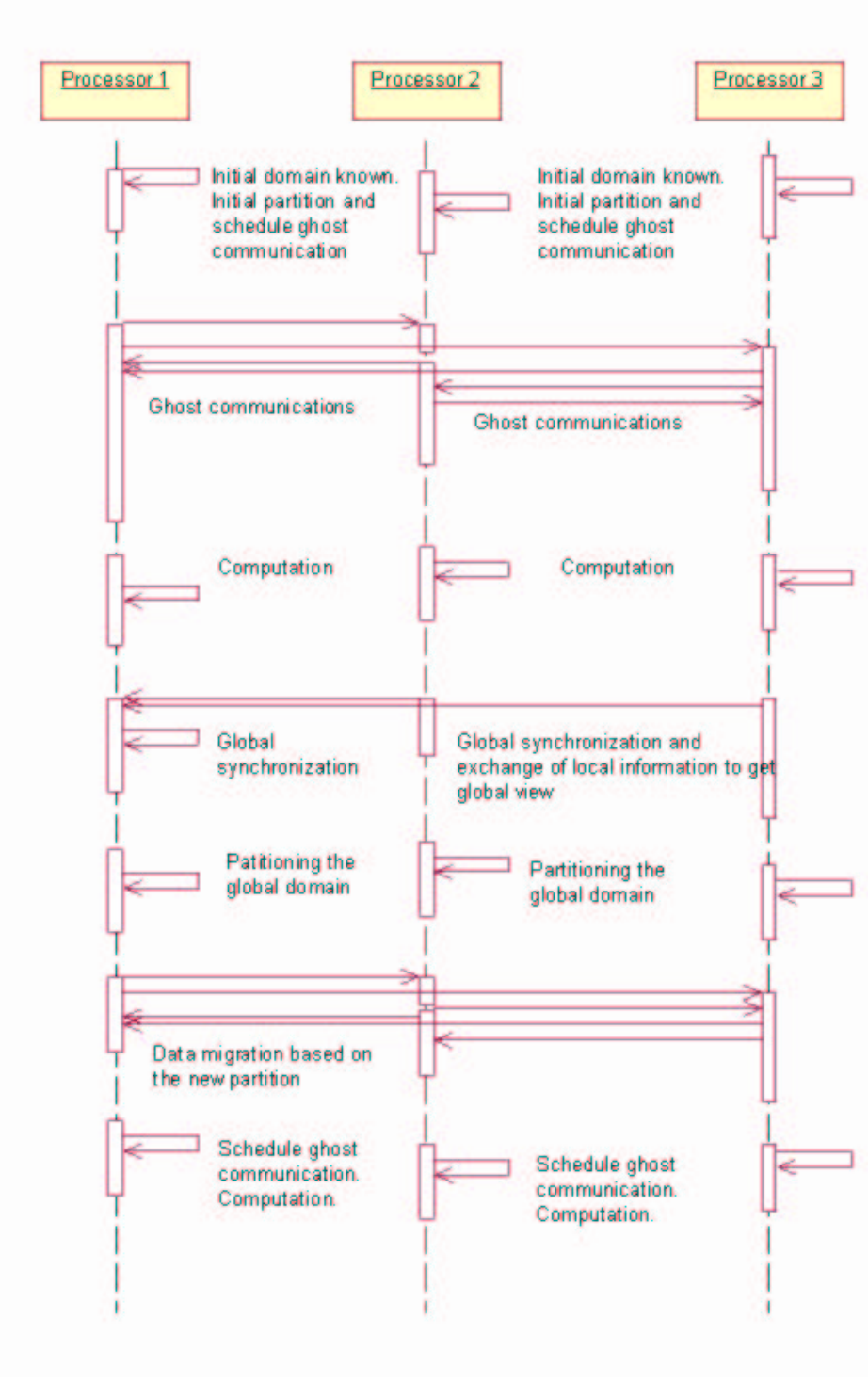
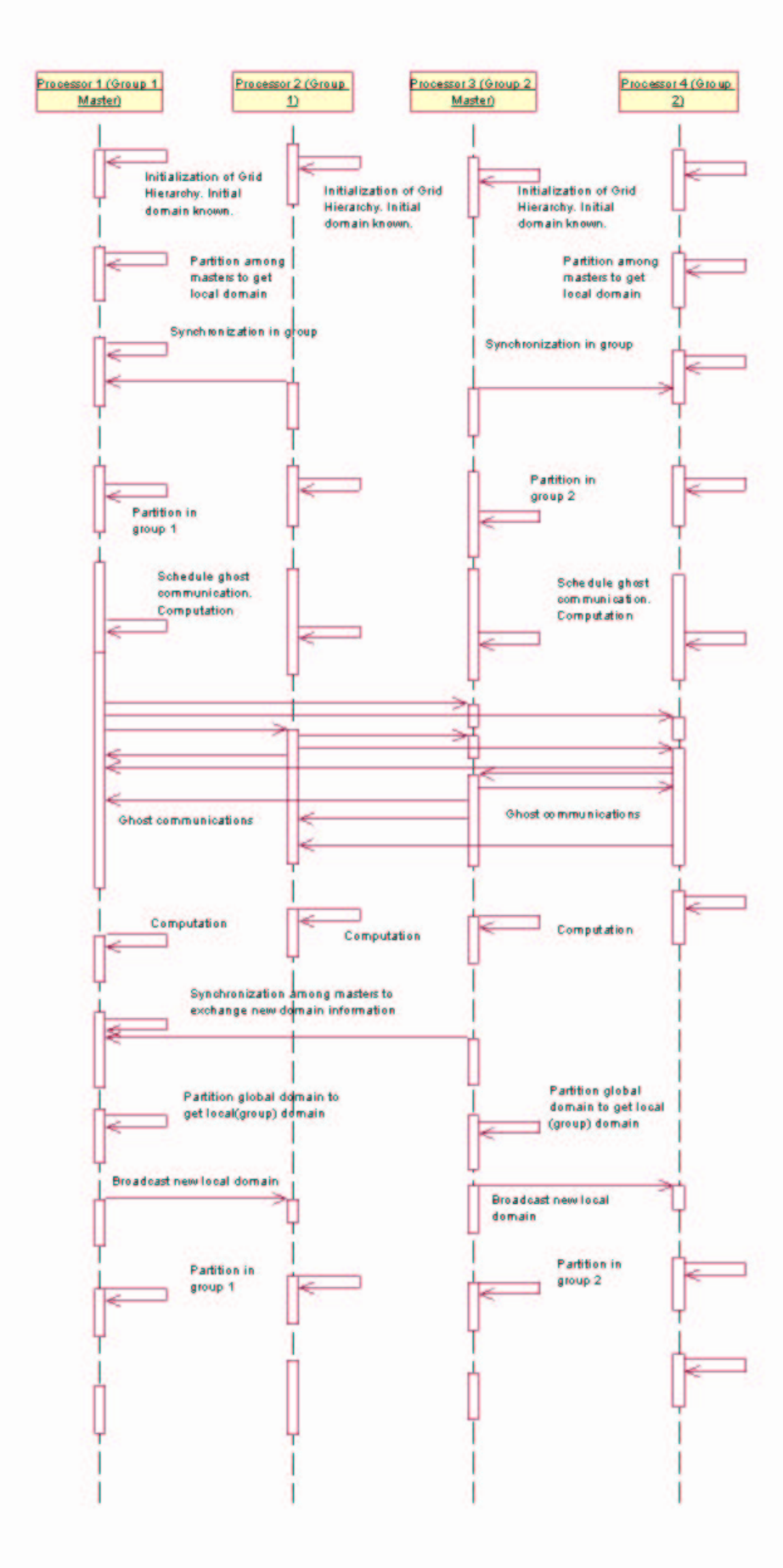


Figure 4.4: Sequence of events in the original partitioning scheme in GrACE


Processor 1 (Group 1 Master)
Processor 2 (Group 1)
Processor 3 (Group 2 Master)
Processor 4 (Group 2)
Initialization of Grid Hierarchy. Initial domain known.
Initialization of Grid Hierarchy. Initial domain known.
Initialization of Grid Hierarchy. Initial domain known.
Partition among masters to get local domain
Partition among masters to get local domain
Synchronization in group
Synchronization in group
Partition in group 1
Partition in group 2
Schedule ghost communication. Computation
Schedule ghost communication. Computation
Ghost communications
Ghost communications
Computation
Computation
Computation
Synchronization among masters to exchange new domain information
Partition global domain to get local(group) domain
Partition global domain to get local (group) domain
Broadcast new local domain
Broadcast new local domain
Partition in group 1
Partition in group 2

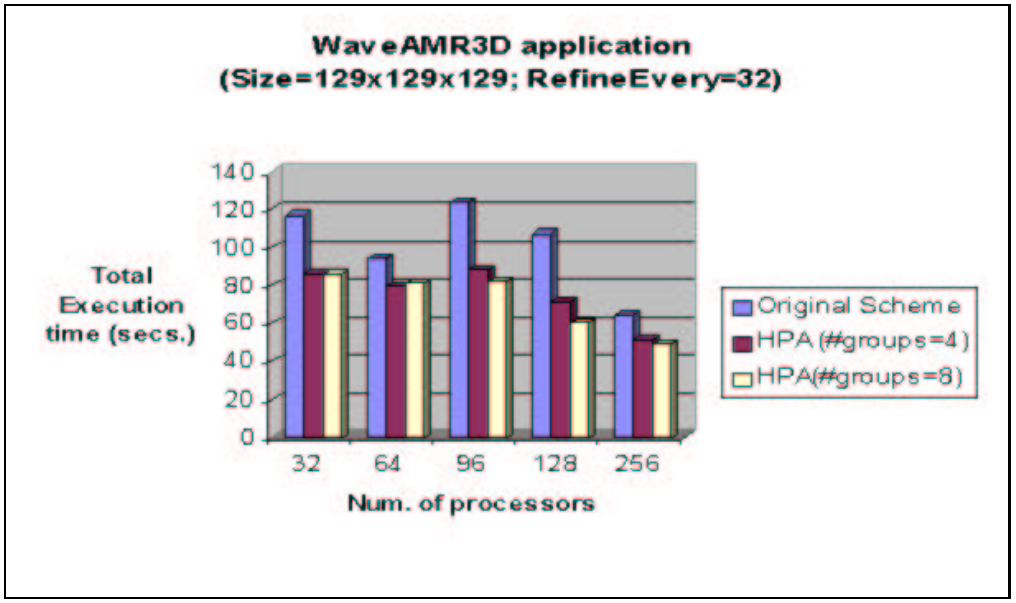


Figure 4.6: WaveAMR3D application with a grid size=129x129x129 with original and HPA schemes

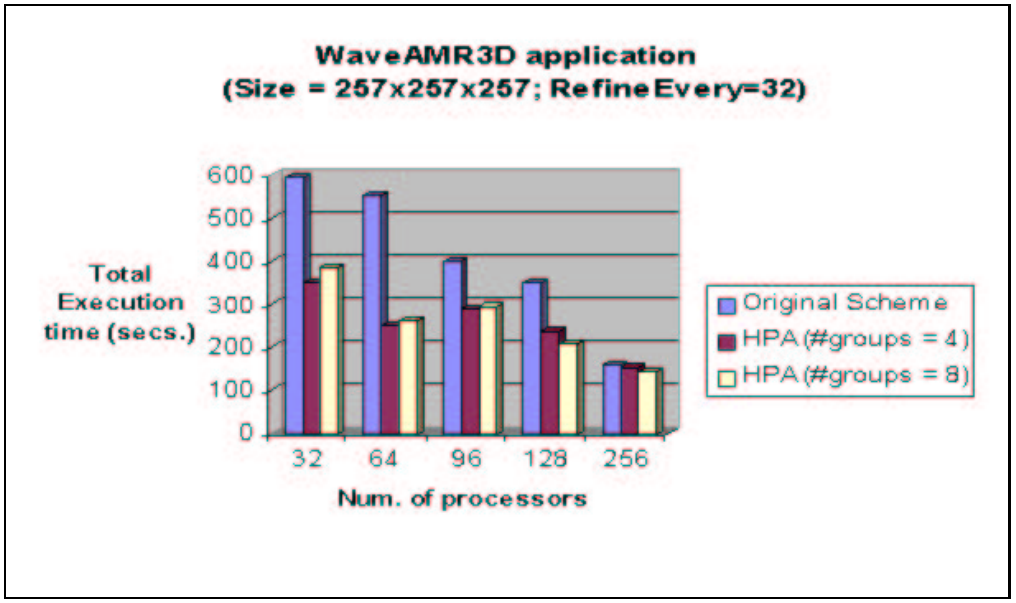


Figure 4.7: WaveAMR3D application with a grid size=257x257x257

Chapter 5

Discussion and Future Work

5.1 Discussion

Scientific applications typically model physical phenomena like weather, cyclones, black hoe interactions etc. These physical models have storage and computational requirements that cannot be fulfilled by one single large machine that is available today. Distributed implementations of these physical models give us a viable solution, but these give rise to challenges in data distribution, communication and sychronization, load balancing and distribution etc.

Performance is critical in such scientific applications that already have large resource requirements. This thesis attempted to optimize the performance of one such AMR libray, GrACE by

- Building a multithreaded communication engine that attempted to improve performance by overlapping communication with computation.
- Proposing a new hierarchical load balancing algorithm to decrease global synchronization overheads of the repartition phase. This will enable better scaling of the library on thousand's of processors.

The experiments using the multithreaded communication engine showed that performance of the library improved by as much as 50The experiments using the hierarchical partitioning scheme showed that the new scheme has no overheads on smaller number of processors. These results show that the scheme has the potential to perform and scale very well on larger number of processors.

5.2 Future Work

This thesis is but a small step in the field of performance optimization. The experiments show that multithreading does improve performance to a great extent. The future direction would be to integrate both the multithreaded engine and the hierarchical scheme and experiment the performance. The hierarchical partitioning scheme groups the processors into smaller groups to decrease global synchronizations. This scheme can be programmed to change the processor affiliations to groups and also change the group sizes during the course of computation. This can improve performance by decreasing data movement during the repartitioning phase.

The grouping scheme can also be extended to heterogeneous partitioning. The heterogeneous processors can be grouped to form groups of different sizes, which participate in the computation. The loads assigned to these processor groups can be evaluated by some partitioning algorithms that assign loads based on the heuristic group capacities.

References

[1] Nikos Chrisochoides. Multithreaded model for dynamic load balancing parallel adaptive PDE computations. Technical Report CTC95TR221, Cornell University, 1995.

[2] Edward W. Felten and Dylan McNamee. Improving the performance of Message-Passing Application by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference*, April 1992.

[3] Bradford Nichols et. al. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, 1996.

[4] H.Sagan. *Space Filling Curves.* Springer Verlag, 1994.

[5] J.R.Pilkington and Scott B. Baden. Partitioning with Space filling Curves. Technical Report CS94-349, Department of Computer Science, University of California, San Diego, CA, 1994.

[6] M.J.Berger and J.Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

[7] M.Parashar and J.C.Browne. A Data Management Infrastructure for Parallel Adaptive Mesh Refinement Techniques. Technical Report 2.400, TICAM, University of Texas at Austin, Texas, 1995.

[8] M.Parashar and J.C.Browne. Distributed Dynamic Data Structures for Parallel Adaptive Mesh Refinement. In *Proceedings of the International Conference for High Performance Computing*, pages 22–27, December 1995.

[9] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers Inc., San Fransisco, CA, 1997.

[10] PARAMESH Software Package. http://esdcd.gsfc.nasa.gov/ess/macneice/paramesh/paramesh.html.

[11] SCOREC Parallel Scientific Computation Software Package. http://www.scorec.rpi.edu/programs/parallel/parallelscientific.html.

[12] Manish Parashar. Grace - Grid Adaptive Computational Engine. http://www.caip.rutgers.edu/ parashar/TASSL/Projects/GrACE.

[13] David R.Butenhof. *Programming with POSIX threads.* Addison-Wesley, 1997.

[14] BATSRUS Software Package. http://hpcc.engin.umich.edu/HPCC/codes/2/BATSRUSv2.html.

[15] SAMRAI Structured Adaptive Mesh Refinement Applications Infrastructure. http://www.llnl.gov/CASC/SAMRAI, SAMRAI homepage, 1999.

[16] *MPI Forum hompage.* http://www.mpi-forum.org.