ABSTRACT OF THE THESIS

Evaluation and Optimization of Load Balancing/Distribution Techniques for Dynamic Adaptive Grid Hierarchies.

By MAUSUMI SHEE

Thesis Director: Professor Manish Parashar

Dynamically adaptive techniques for the solution of partial differential equations that employ locally optimal approximations can yield highly advantageous ratios for cost/accuracy. *Adaptive Mesh Refinement* (AMR) techniques seek to improve the accuracy of the solution by dynamically refining the computational grid in regions of high local solution error. Distributed implementations of these methods offer the potential for accurate solutions of physically realistic models of important physical systems. These distributed implementations however, lead to interesting challenges in dynamic data-distribution, re-distribution and load balancing. This thesis makes the following contributions:

- An application-centric performance characterization and evaluation of dynamic partitioning and load-balancing techniques for distributed adaptive grid hierarchies that underlie adaptive mesh-refinement algorithms (AMR). The goal of the characterization is to enable the selection of the most appropriate mechanism based on application and system parameters. The characterization defines the following metrics: *Load Balance, Distribution Quality (size, aspect ratio), Grid Interaction Overheads, Data Movement.*
- An evaluation of four partitioning techniques viz., Composite (SFC) technique, *Independent Grid Distribution technique (IGD), Composite Grid Distribution(CGD)* technique, *Independent Level Distribution technique (ILD*), and the optimization of the SFC and CGD techniques. The evaluation is based on the following applications: 1) 3D Buckley-Leverette

equation kernel (BL) that was used in oil reservoir simulations. 2) 3D Wave equation kernel (Wave) that was used in a numerical relativity simulation. Optimization consisted of improving the partition routine for CGD and optimization of the data structure of the SFC scheme. The goal of this optimization is to obtain improved performance of the above-mentioned criteria, and

- The design and implementation of a (AMR) Simulator. The goal is to evaluate the above mentioned distribution techniques and also make it available for run-time/post-operation use.

Acknowledgements

I am grateful to many people who helped me throughout this endeavor, not only for their encouragement and feedback, but also for ideas on how to make things work, or work better. I am grateful to my thesis advisor Professor Manish Parashar not only for his invaluable guidance, encouragement and support during my stay at Rutgers, but also for his patience. I am thankful to the rest of my committee members Professors Deborah Silver and Ivan Marsic for their valuable advice and suggestions regarding my thesis.

I would also like to thank the CAIP computer facility staff for their support. They always were prompt in providing answers to all my questions. My thanks also to the members of the TASSL lab, for their invaluable support, co-operation, for all the discussions we have had and also for putting up with me. My thanks are due to Ms. Barabara Klimekewicz, Graduate Secretary, Department of Electrical and Computer Engineering, for her guidance and advice. I also want to thank my parents and family for their encouragement during my studies in graduate school.

Finally, the person I want to thank the most, without whom I could not have mustered the confidence to tackle this project and my studies at Graduate School, nor the stamina to see it through, is my beloved husband, Shantanu Shee.

Table of Contents

Table of Figures

List of Tables

Chapter 1

Introduction

The accurate numerical simulations of complex physical phenomena using *partial differential equations* (PDEs) require large amounts of computer resources in terms of memory storage and computation time, because their domains are discretized on high-resolution meshes. A characteristic of many such phenomena is that they are usually smooth over most of their domains but contain regions (either boundary or internal) with steep gradients, shocks or discontinuities. Thus, the computational resources mentioned above are often largely wasted on sub-domains that do not have such a high resolution. Such scientific, complex physical phenomena that apply partial differential equations and numerical methods can be easily solved by adaptive mesh refinement (AMR) methods. AMR is a class of strategies 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 [9,11,32] store the solution using graph or tree representation [19,25]; 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. Thus 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.

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.

Dynamically adaptive methods for the solution of partial differential equations that employ locally optimal approximations can yield highly advantageous ratios for cost/accuracy when compared to methods based upon static uniform approximations. These techniques seek to improve the accuracy of the solution by dynamically refining the computational grid in regions of high local solution error. Distributed implementations of these methods offer the potential for accurate solution of physically realistic models of important physical systems. We believe that the next generation simulations of complex physical phenomenon will be built using such dynamically adaptive techniques executing on distributed heterogeneous computational grids, and will provide dramatic insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, oil reservoirs and aquifers, and seismic models of the whole earth.

Distributed implementations of adaptive applications lead to interesting challenges in dynamic resource allocation, data-distribution and load balancing, communications and coordination, and resource management. The overall efficiency of the adaptive algorithms is limited by the ability to partition the underlying data-structures at run-time to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load. A critical requirement while partitioning adaptive grid hierarchies that underlie these algorithms is the maintenance of logical locality, both across different levels of the hierarchy under expansion and contraction of the

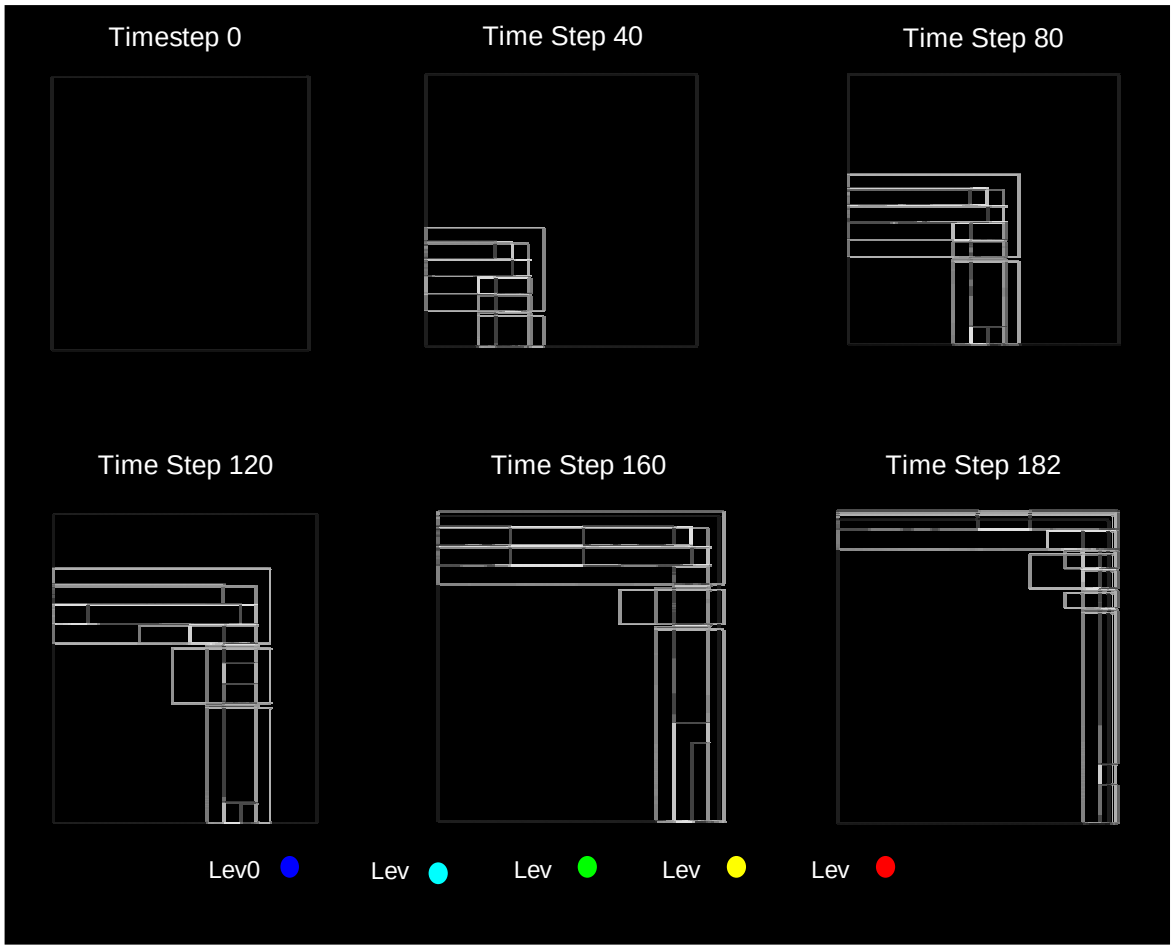


Figure 1.1 AMR Grid Structure for Buckley-Leverette Application

adaptive grid structure, and within partitions of grids at all levels when they are decomposed and mapped across processors. The former enables efficient computational access to the grids while the latter minimizes the total communication and synchronization overheads. Furthermore application adaptivity results in application grids being created, moved and deleted on the fly, making it necessary to efficiently re-partition the hierarchy on the fly so that it continues to meet these goals. A sequence of grid hierarchies for two applications, Buckley Leverette and a Wave application are shown in figures 1.1 and 1.2 respectively.

Figure 1.2 AMR Grid Structure for 3D Wave Application

1.2 Overview of the thesis

This thesis presents a brief design and evaluation of a suite of distribution/load-balancing techniques for distributed adaptive grid hierarchies that underlie parallel adaptive mesh refinement techniques. It also presents a performance characterization of dynamic partitioning and load balancing techniques, the optimization of the data-structure for one of the schemes and an optimization of the partition method that had for CGD. It then presents an AMR simulator that is employed to perform an evaluation of the schemes before and after the optimization in a simulated environment. The workflow associated with the partitioning and performance evaluation is as follows: A trace file consisting of a trace of grid adaptations from a 2D/3D AMR application is fed

to a partitioner module. The partitioner employs the scheme of choice to distribute the load among the processors and redirects the output into an Output parameters file, which the simulator uses as its input to do the performance evaluation.

1.3 Contributions of the thesis

1 An application-centric performance characterization and evaluation of dynamic partitioning and load-balancing techniques for distributed adaptive grid hierarchies that underlie adaptive mesh-refinement algorithms (AMR). The goal of the characterization is to enable the selection of the most appropriate mechanism based on application and system parameters. The characterization defines the following metrics: Load Balance, Distribution Quality (size, aspect ratio), Grid Interaction Overheads, Data Movement.

2 An evaluation of four partitioning techniques viz., Composite/SFC technique, Independent Grid Distribution technique (IGD), Composite Distribution technique(CGD), Independent Level Distribution technique(ILD), and the optimization of the SFC and CGD techniques. The evaluation is based on the following applications: 1) 3D Buckley-Leverette equation kernel (BL) that was used in oil reservoir simulations. 2) 3D Wave equation kernel (Wave) that was used in a numerical relativity simulation. Optimization consisted of improving the partition routine common to the above mentioned techniques and optimization of the data structure of the Composite/SFC scheme. The goal of this optimization is to obtain improved performance of the above-mentioned criteria, and

3 The design and implementation of a (AMR) Simulator. The goal is to evaluate the above mentioned distribution techniques and also make it available for run-time/post-operation use.

1.4 Outline of the thesis

This thesis consists of seven chapters of which this chapter is the first. The next three chapters provide background on adaptive mesh refinement strategy and partitioning issues of concern, while the final three chapters discuss the associated research work.

Chapter 2 presents the background of the thesis by providing an overview of Dr. Marsha Berger's Adaptive mesh refinement strategy, 'Grid Adaptive Computational Engine' (GrACE) which is the infrastructure developed by Dr. Manish Parashar [10,24], for AMR for the solution of complex physical phenomenon, and the related work that deal with partitioning techniques by AMR for solution of PDEs.

Chapter 3 presents a suite of six distribution and load-balancing techniques that have been designed by us and used by current infrastructures supporting adaptive applications. It also describes the metrics to characterize the partitioning schemes and the family of adaptive algorithms itself targeted in this thesis.

Chapter 4 presents the optimization of the partition module for the Combined Grid Distribution scheme and the optimization that is the design and implementation of the data structure for the Composite distribution scheme.

Chapter 5 presents the design and implementation of the AMR simulator that does the evaluation of the suite of the above-mentioned load-balancing and distribution schemes.

Chapter 6 presents an experimental evaluation of the load-balancing and distribution schemes.

Chapter 7 presents our conclusions and directions for future work.

Chapter 2

Background and Related work

2.1 Formulation of structured AMR

Dr. Marsha Berger developed a formulation of the adaptive mesh refinement strategy for structured meshes based on the notion of multiple, independently solvable grids, all of identical type, but each 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. 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 Layout of the hierarchy:

This section describes the grid hierarchy , each of which contains a set of grids as shown in figure 2.1. The implementation represents the hierarchy as a directed graph that is acyclic on relationships between levels (e.g, from parent to child and vice versa, figure 2.1). Every grid is completely covered by some non-empty set of parent grids; both its active computational interior and its ghost boundaries are covered, except those portions of the ghost boundary that lie on the exterior of the overall computational domain. In addition, the finer grids abut the coarser cells, so that the number of non-boundary cells along each axis of each finer grid is an integer multiple of the refinement factor.

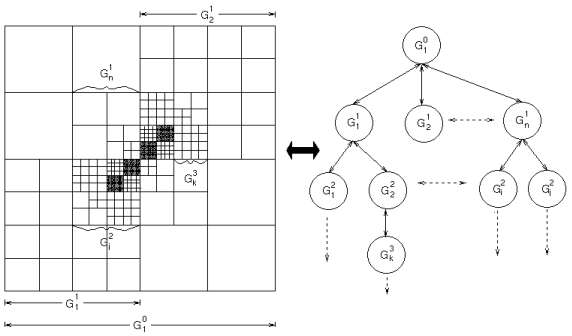


Figure 2.1 AMR Grid Hierarchy

At the root level, each grid consists of a computational interior and a ghost region. At all finer levels, each grid consists of a region of inerest that has been refined from the immediately coarser level; a buffer region, which allows the existing grids to cover the region of interest and the ghost boundary as shown in figure 2.2

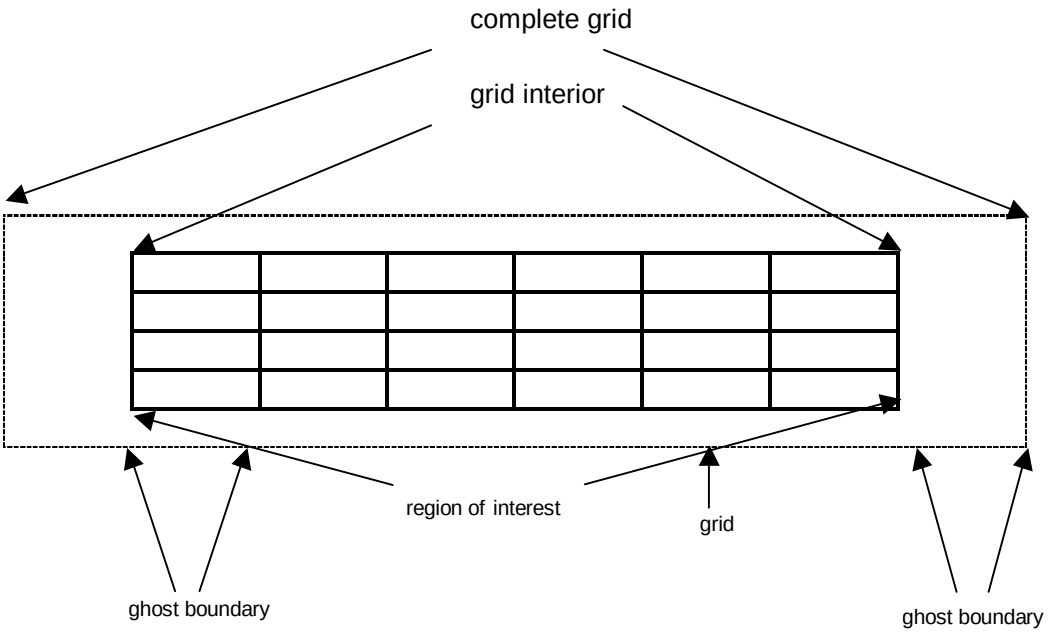


Figure 2.2 Components of a Grid

Sometimes, the buffer region may be partially absorbed if the grid abuts another grid at the same level. In any case, each grid will retain its full boundary region, even if it is overlapped by other grids at the same level.

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 interval, then recursively advancing the next finer level by enough iterations at its (smaller) time interval to reach the same physical time as that of the newest solution of the current level.

The AMR algorithm

```
Integrate (level)
begin
        evolve(level)
        if "level isn't finest " then begin
                for =0 to time_refinement_factor - 1 do
                        intergrate (level +1)
        end
end
```

The integrations at each level are recursively interleaved between iterations at coarser levels. Thus, a majority of the time computing is spent on the finest level, as a direct result of the fact that Berger's AMR refines in time as well in space: 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, in order to consist of the same total number of cells for the level, because every coarse cell covers r^d fine cells. 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 1/16 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:

The Refined AMR Algorithm

```
Integrate (level)
begin
        if "time to refine" then Refine(level)
                Collect boundary values
                evolve(level)
        if "level isn't finestexisting " then begin
                for r=0 to time_refinement_factor - 1 do
                        intergrate (level +1)
        end
        incrementTime(level)
        if "level isn't finest existing" then begin
                do any corrections
                Prolongate(level, level+1)
        end
```

2.2 GrACE

The work presented in this thesis is based on the GrACE [10, 24] infrastructure, which is an approach to distributing AMR grid hierarchies, developed by Dr. Manish Parashar. GrACE [10,24] 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). 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 mapping which efficiently map N-dimensional space to 1-D dimensional space.

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. In this paper we characterize the distribution mechanism underlying 5 such systems, viz. BATSRUS, PARAMESH, Parallel Mesh Database, SAMRAI and GrACE.(please refer section 2.2)

2.3.1 BATSRUS

BATSRUS[1] 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 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 [20] 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 [29] 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 [26] 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.

Chapter 3

Run-Time Partitioning Techniques for Structured AMR Grid Hierarchies

3.1 Run-time Partitioning Dynamic AMR Grid Hierarchies

This chapter presents six dynamic partitioning and load balancing schemes that have been implemented and evaluated. These schemes encapsulate key ideas underlying the approaches used by the AMR infrastructures described in Chapter 2. It then presents metrics to measure and evaluate the load-balancing techniques.

3.1.1 Composite Distribution/Space Filling Curves

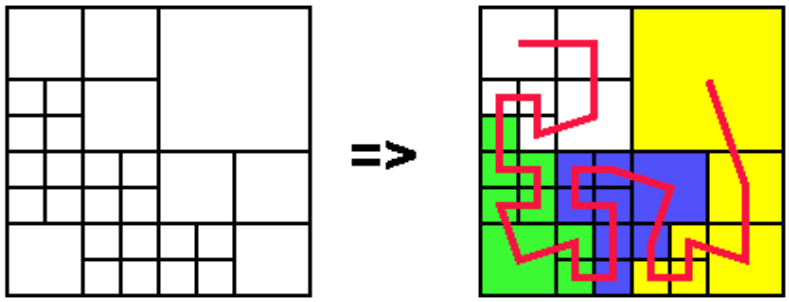


Figure 3.1 Space-Filling Curve Representation of an Adaptive Grid Hierarchy

Space-filling curves (SFC) are a class of locality preserving mappings from d-dimensional space to 1-dimensional space i.e., $N^d \rightarrow N^1$, such that each point in N^d is mapped to a unique point or index in N^1. The self-similar or recursive nature of these mappings can be exploited to represent a hierarchical structure and to maintain locality across different levels of hierarchy. The SFC representation of the adaptive grid hierarchy is a 1-D ordered list of composite grid blocks where each composite block represents a block of the entire grid hierarchy and may contain more than one

grid level; i.e. inter-level locality is maintained within each composite block. Figure3.2 illustrates the composite representation for a two dimensional grid hierarchy. Using the space-filling curve representation as shown in figure 3.1,

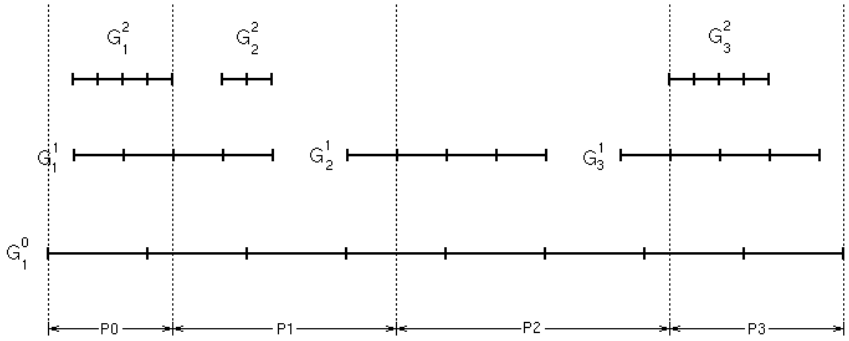


Figure 3.2 Space-Filling (Composite) Distribution

the adaptive grid hierarchy can be simply partitioned by partitioning the composite list to balance the total work assigned to each processor. This decomposition using the Peano-Hilbert space-filling ordering for a 1-D grid hierarchy is shown in figure 3.2

As inter-level locality is inherently maintained by the composite representation, the decomposition generated by partitioning this representation eliminates expensive gather/scatter communication and allows prolongation and restriction operations to be performed locally at each processor.

3.1.2 Independent Grid Distribution

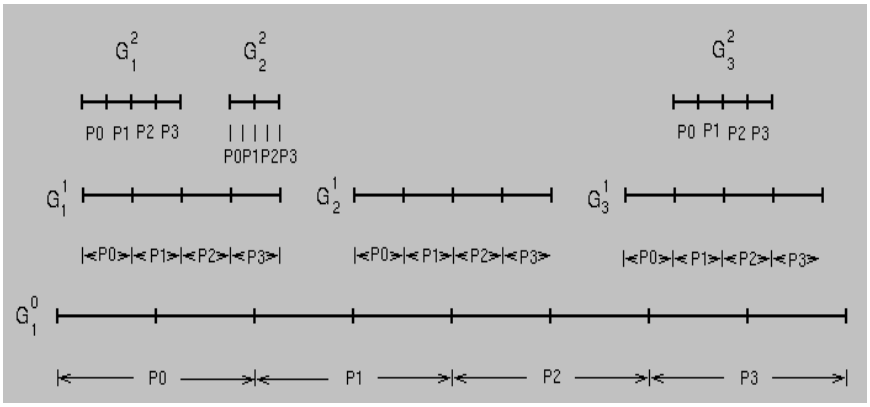


Figure 3.3 Independent Grid Distribution

The independent grid distribution (IGD) scheme, shown in Figure 3.3, distributes the grids independently across the processors. This distribution leads to balanced loads and no redistribution is required when grids are created or deleted. However the decomposition scheme can be very inefficient with regard to inter-grid communication. In the adaptive grid hierarchy, a fine grid typically corresponds to a small region of the underlying coarse grid. If both the fine and coarse grids are distributed over the entire set of processors, all the processors will communicate with the small set of processors corresponding to the associated coarse grid region, thereby causing a serialization bottleneck. For example, a restriction from grid G22 to grid G11 requires all the processors to communicate with processor P3.

3.1.3 Combined Grid Distribution

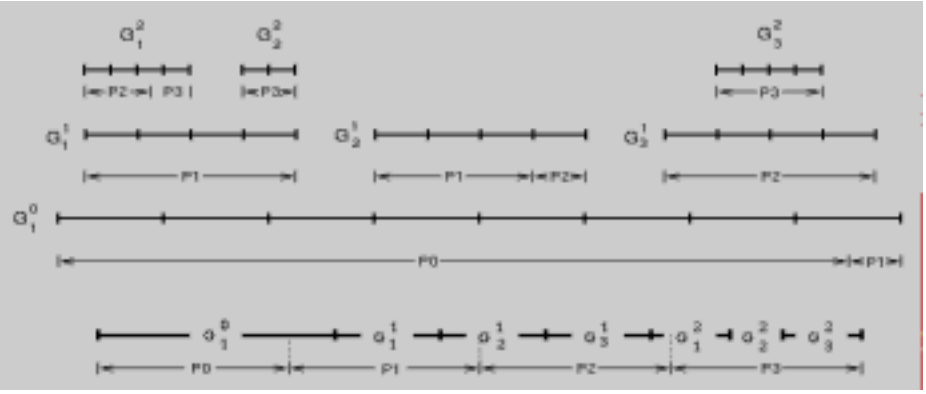


Figure 3.4 Combined Grid Distribution

The combined grid distribution (CGD), shown in figure 3.4, distributes the total work load in the grid hierarchy by first forming a simple linear structure by abutting grids at a level and then decomposing this structure into partitions of equal load. The combined decomposition scheme also suffers from the serialization bottleneck described above but to a lesser extent. For example, in Figure, G21 and G22 update G11 requiring P2 and P3 to communicate with P1 for every restriction. Regriding operations involving the creation or deletion of a grid are extremely expensive in this case, as they require an almost complete redistribution of the grid hierarchy. The combined grid decomposition does not exploit the parallelism available within a level of the hierarchy. For example, when G01 is being updated, processors P2 and P3 are idle and P1 has only a small amount of work. Similarly when updating grids at level 1 (G11, G12 and G13) processors P0 and P3 are idle, and when updating grids at level 2 (G21, G22 and G23) processors P0 and P1 are idle.

3.1.4 Independent Level Distribution

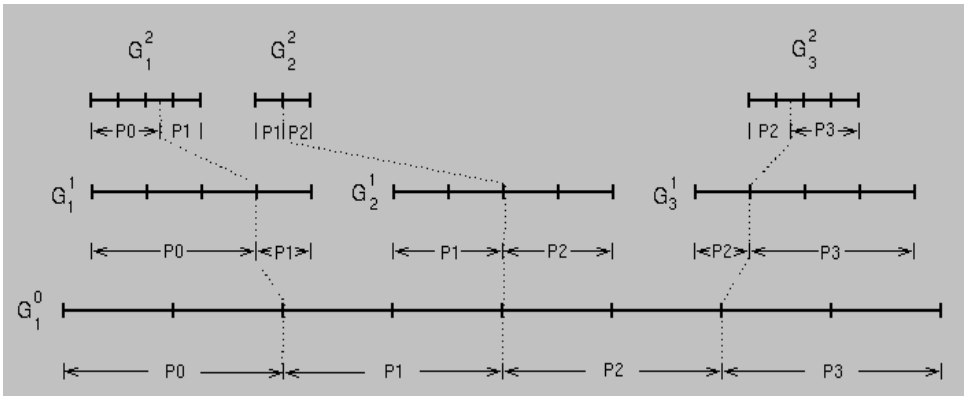


Figure 3.5 Independent Level Distribution

In the independent level distribution (ILD) scheme (see Figure 3.5), each level of the adaptive grid hierarchy is individually distributed by partitioning the combined load of all component grids at the level is distributed among the processors. This scheme overcomes some of the drawbacks of the independent grid distribution. Parallelism within a level of the hierarchy is exploited. Although the inter-grid communication bottleneck is reduced in this case, the required gather/scatter communications can be expensive. Creation or deletion of component grids at any level requires a re-distribution of the entire level.

3.1.5 Iterative Tree Balancing

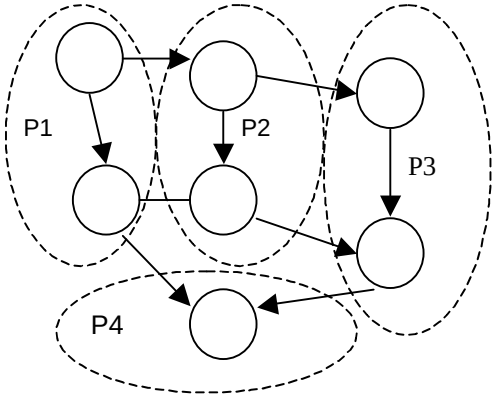


Figure 3.6 Iterative Tree Balancing

The iterative tree balancing (ITB) scheme (see Figure 3.6) treats the dynamic partitioning and load-balancing problem as a graph-partitioning problem. A table is created from the grids at each timestep, which keeps pointers to neighboring and parent grids. A breadth first search is made on this graph i.e. for every grid immediate neighbors and children are also considered along with load distribution. Thus load balancing, inter level communication and intra level communication are addressed together. This scheme is promising from the point of view that all the constraints are dealt with, to some extent.

3.1.6 Weighted Distribution

The weighted distribution scheme is a heuristic based hybrid scheme that attempts to combine the features of the other schemes described in this section. As previously observed, there are three primary parameters that need to be controlled to minimize the overheads of an adaptive grid hierarchy distribution, viz. intra-level communication, inter-level communication and data movement at each regrid. In the weighted distribution we first assign a weight to each of these

overheads. This weight defines the significance and contribution of the overhead to the overall application performance and depends on the system architecture and dynamic application behavior. The next step uses these weights to compute the affinity of each component grid to the different processors. Initially grids have no affinity for any processor. For each grid, the affinity of the processor(s) housing its parents is now increased by the inter-level communication weight. Similarly the affinities of processors housing the neighbors of the grid are increased by the intra-level communication weight, and affinity of the original location of the grid is increased by the data-movement overhead weight. The grid is now assigned to the available processors (i.e. total assigned load is below threshold for load balancing) to which the grid has maximum affinity. If the grid has equal affinity to more than one processor, the grid is either split among the processor (if its size is greater than the size threshold) or assigned to the processor with least load. Weights assigned to the different parameters can change dynamically depending on the current application and system states. For example if the application has many component grids and uses a large stencil, then the dominating weight is associated with intra-level communication. Similarly if the application is very dynamic and needs to regrid very often, the data-movement weight dominates.

3.2 Characterization Criteria

We use four criteria to characterize distribution mechanism for AMR adaptive grid hierarchies, viz. load balance, distribution quality, grid interaction overheads (inter-processor communication and memory copy), and data-movement overheads.. These criteria are described below.

3.2.1 Load Balance

The load balance metric measures a combination of the distribution of load across the processors, the time taken to achieve the distribution. AMR applications require re-distribution and load balancing at regular intervals; consequently the time spent in this effort is critical. The goal of this

metric is to define operational points that represent the best balance between the effort spent in balancing the load and the balance achieved. In this paper we only address the quality of the load-balance and not the effort required.

3.2.2 Distribution Quality

Distribution quality is quantified by the number of grid components created on each processor and the quality (size, aspect ratio) of these components. The former captures the overheads due to the allocation, operation, and management and de-allocation of grid components. Large number of small grid increases the number of memory copies required for inter-level and intra-level communications. The size and shape of the grids also affects the communication/memory copy behavior. Bad aspect ratios result in larger interfaces between sibling grids and increased intra-level communications. Finally grid size also affects the overall cache behavior. Our goal is to use this metric to determine an acceptable range for the shape and size of grid components for different architectures, and use this to drive the distribution. In this paper we evaluate the number of boxes for each scheme studied.

3.2.3 Grid Interaction Overheads

The grid interaction overhead metric aims at characterizing the ability of the distribution scheme to capture and maintain application locality. Here we measure the overheads of four kinds of communications: inter-grid communications between grids at different levels, intra-Grid communication along ghost boundaries, and inter- and intra grid memory copies for co-located grid components. Maintaining locality to minimize these overheads can lead to conflicting optimizations. The objective of this metric is to identify a balance between the two overheads based on system memory architecture and communication characteristics that can achieve best overall performance.

3.2.4 Data Movement

Every refinement step in the AMR algorithms typically causes the adaptive grid hierarchy to change requiring redistribution. The redistribution should be incremental so as to minimize the data that has to be relocated. The objective of the data movement metric is to characterize the ability of the distribution scheme to minimize redistribution costs by reassigning grids to their original location. Optimizing this metric can lead to conflicts with requirements for optimizing load balancing and interaction overheads.

Chapter 4

Design, Implementation and Optimization of Composite (SFC) and Combined Grid Distribution (CGD) techniques

The design and implementation of each load balance and distribution technique consist of three steps in order to partition the adaptive grid hierarchy across the processors. They are:

i) definition of the data structure

ii) arrangement of the grids into a list in an order dictated by the distribution technique (for example: for SFC scheme, arrange as per parent – child relationship), and

iii) the partition routine that actually partitions the grid hierarchy and distributes the grids to the processors.

This chapter presents the design and optimization of the data structure and the arrangement of the grids in an order dictated by the SFC technique and the design and optimization of the partition routine of the CGD technique. The goal of the optimization is to provide improved performance criteria that we use to characterize the distribution mechanism (section 3.2).

4.1 Optimization of the Composite Grid Distribution (SFC) Technique

The original scheme read in the boxes and stored them in a linked list. It then compared each *bounding box*[10,24] with every other bounding box in the list to identify its parent. It is then placed it behind its parent. The complexity of time spent was O(n) .We have come up with a tree general tree data structure to hold the boxes in parent child relationship. We have employed the strategy of representing a general tree as a binary tree and a preorder traversal [30] leads to the formation of the bounding boxes in the desired parent-child-grandchild ordering. The complexity of

doing a search on the binary tree is O(lg n). Hence we see considerable improvement in the time taken to perform the parent-child-grandchild ordering.

4.1.1 Design of the data structure

At each timestep, the grid hierarchy is such that, after the refinement process, a grid may have more than two children. In general, we find it desirable to deal with a data structure for which any data object may have a relationship with an arbitrary (but finite) number of child data objects. In this scenario, the number of children per node is independent between nodes and varies dynamically with the growth or decay of the overall tree. We have chosen such a tree structure as our data structure and it is called a *general tree* or a *hierarchical tree*. Indicative of its name, a general tree is applicable to a wide and varied class of problems and is particularly suited to our application. A general tree is a tree such that (a) it is empty, or (b) it contains a root node along with a finite number of disjoint general (sub) trees. Hence, the number of children belonging to each node of the general tree depends upon the grid hierarchy for that timestep.

4.1.2 Representation of the general tree data structure

There are many possible ways of representing a general tree. Among them are those employing

i) variable size nodes, ii) *M*-ary trees, and iii) binary trees.

We implement the general tree as a binary tree because, an implementation based on the variable sized nodes can be considerable complex that one based on fixed sized nodes. The use of an *M-ary* tree is also not considered because, it is for the most part an impractical means of representing a general tree. Such an implementation requires an upper limit for the degree of the tree to be fixed in advance, limiting flexibility and ultimately leading to a lot of wasted storage space. Each and every node must contain the storage required for *M* child pointers, regardless of the actual number of children present at any given time. For an *M*-ary tree the ratio of the NULL to non-NULL child

pointers is about 2/3 for a 3-ary tree (M=3) and approaches unity with increasing *M*. A binary tree on the other hand has a NULL to non-NULL child pointer ratio about 1/2 for large trees. This favorable use of space is readily exploited given the realization that *any tree may be represented by means of a binary tree*. Hence, we have considered a binary tree for our representation of the general tree.

Figure 4.1 illustrates the general tree concept taking the *call tree* of a set of functions "f1( )", "g1( )", "h1( )". The function "f1( )" calls "f11( )" ; "f12( ); function "g1( )" calls "g11( )" , "g12( )., "g13( ); the function "f1( )" calls "f11( )" and "f12( )". The parent child relationship is established by " who calls whom". Leaf nodes, such as functions "f11( )", "f12( )" and "g11( )" are those that do not call any functions. Figure 4.2 represents the abstract form of a general tree for the call tree of functions (in terms of nodes). Figure 4.3 shows its equivalent form using child-sibling relationship.

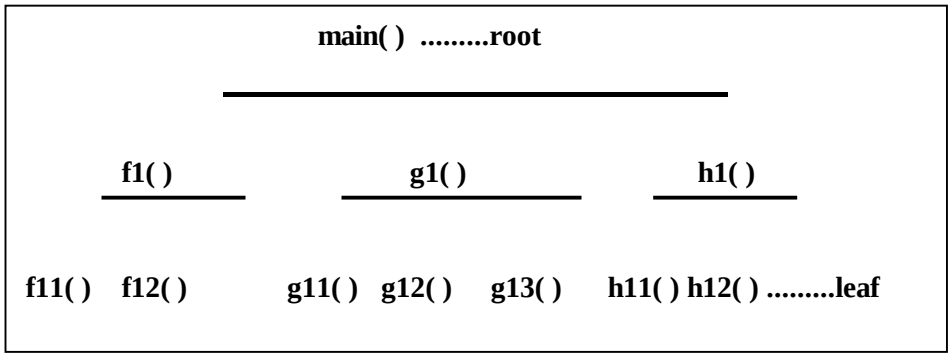


Figure 4.1 Representation of the abstract form of a general tree

This is also called binary tree or linked list implementation of a general tree. Children and siblings are both structures as a linked list. Like binary tree, each node contains one or more data fields and two pointers: one pointer pointing to the head of the singly linked list of its children, and a second pointer pointing to the head of the singly linked list of its siblings. For example, the node containing "f1( )" as data has one pointer (forward direction) pointing to the list of its siblings, "g1( )" and

"h1( )", and a second pointer (downward direction) pointing to the list of its children functions "f11( )" and "f12( )".

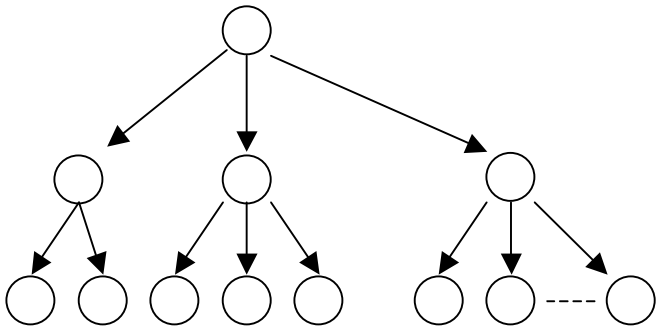


Figure 4.2 Abstract form of a general tree depicted as a hierarchy of nodes

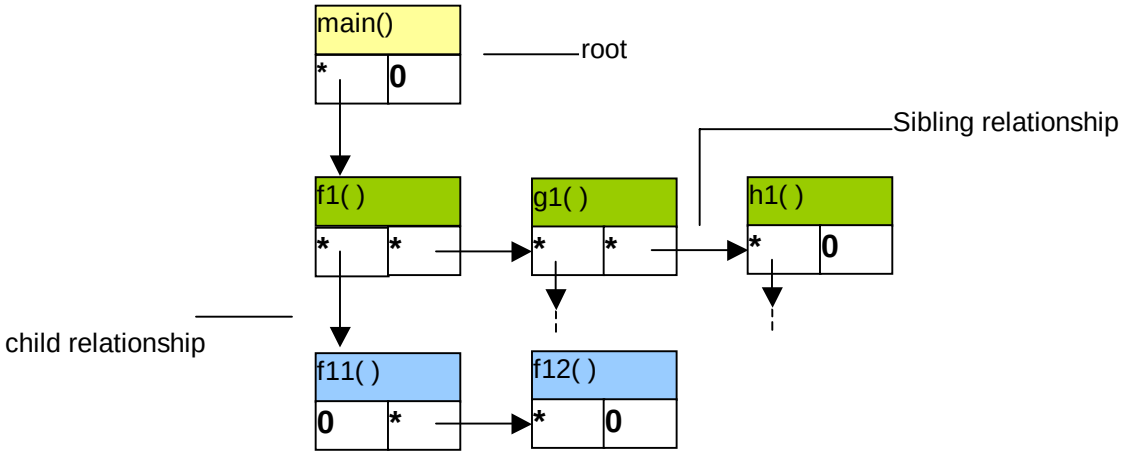


Figure 4.3 Binary tree implementation of a general tree with siblings

To avoid complexity in figure 4.2, children of the nodes with "g1( )" and "h1( )" are not shown. Figure 4.3 shows the implementation of the general tree using linked lists. The siblings "f1( )", "g1( )"and "h1( )" belong to the same singly linked list; "f1( )" is assumed to be the first child of the root node "main". The node containing "f1( )" contains one pointer(it's right pointer) pointing to the next sibling, and a second pointer (it's left pointer) pointing to the singly linked list of its children function nodes, "f11( )" and "f12( )".

4.1.3 Implementation of the general tree data structure

We have used an object-oriented design approach for a general tree, the object is specified by an Abstract Data Type (ADT) general tree and its methods are actions which are directly derived from the ADT definition of the general tree. To implement the object-oriented characteristics of data hiding, encapsulation, and methods for message passing, the General Tree class is defined in a header file.

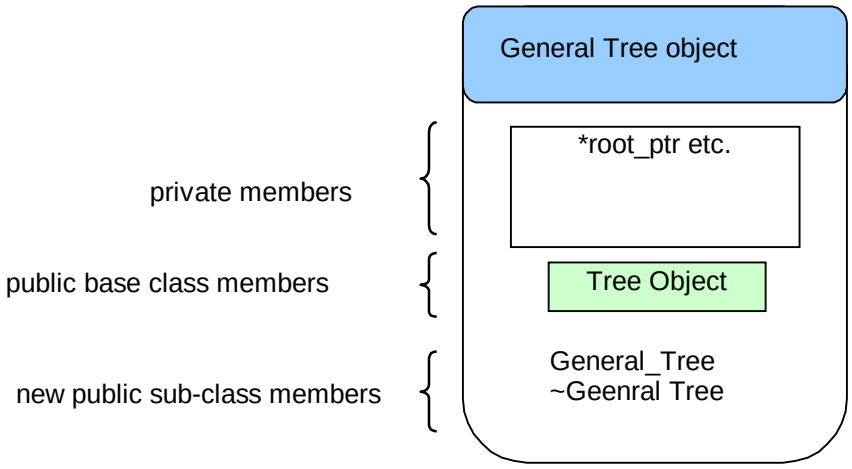


Figure 4.4 General Tree Object

The General tree class defines a type for a general tree object, encompassing both data and operations. Its parent class is the tree class. A *tree* object is specified by and ADT tree. An ADT *tree* is defined as a data structure that has a set of node objects and a set of the generalized operations that are used for defining, manipulating such node objects. The data that are contained in node objects are as abstract as possible, and are abstracted (taken out) by using specified set of operations. (e.g., construct/destroy a tree object, create/delete a node object in a tree object, sort a tree object or traverse a tree object). Hence the tree object is treated as an abstract base class. The

General tree class is hence an implementation specific derived class, and the *general tree* object is a derived tree object formed by the instantiation of the derived General tree class.

4.1.4 Arrangement of the grids into a list

The grids in the adaptive grid hierarchy are maintained in the form of a bounding box list (appendix a). Following are the steps to create the bounding box list.

i) Read in the parameters for the bounding box from the input file and create a bounding box.

ii) Arrange them in the form of the binary representation of the general tree

iii) Construct the bounding box list from the above representation:

This is done as follows: a preorder traversal (appendix a)of the binary tree gives us the bounding boxes in the composite (SFC) order. In order to partition this bounding box list we create units where each unit consists of a parent, one of its children say A, children of A and so on to the last level of the hierarchy. It is noted that a bonding box may be divided across several units. For instance, a node that has two children will belong to each unit that houses its children. In order to create these units we perform the following steps:

- determine the number of nodes: this is done during the preorder traversal of the tree
- determine the number of units: this is also done during the preorder traversal where each time we encounter a node which doesn't have a left child, we increment the number of units by one.
- determine the participation number of each. node. We define participation number of a node as the number of times each node participates in all the units. This is also the sum of participation numbers of its children (recursively). The purpose is to find out how many units each node will belong to. Algorithm 4.1 describes the process to find out the participation number of each node (please refer appendix a).
- Divide each node by its participation# into sub-nodes and place them in their units. Algorithm 4.2 describes the algorithm to place the sub-nodes into their units. (please refer appendix a).
- call the partition routine to perform to do the load-balance and distribution of the grids across the processors. (Sections 4.2.2 and 4.2.4)

4.2 Optimization of the Combined Grid Distribution 'Partition' Method

The goal of the optimization of the partition method is to generate better quality partition in terms of the number of boxes generated and the load balance achieved.

4.2.1 Analysis of the original partition method

A high level description of the partitioning process of the single list of work elements (bounding boxes) to be partitioned and assigned to the processors comprises of the following steps below:

- Calculate the total work and the average work
- Add the work of a box to the existing work of current processor
- Check if this work is more than or equal to the average work
- Allocate to current processor
- Recalculate the average work

The partition method of the combined grid distribution scheme is a recursive scheme where in the total load and the average load on each processor is calculated. The load is then evenly distributed among the processors. At the point of imbalance, the current bounding box is broken into two smaller blocks depending on the amount of work needed to even out the imbalance. The first block is assigned to the current processor. We then moved forward and start the redistribution process all over again.

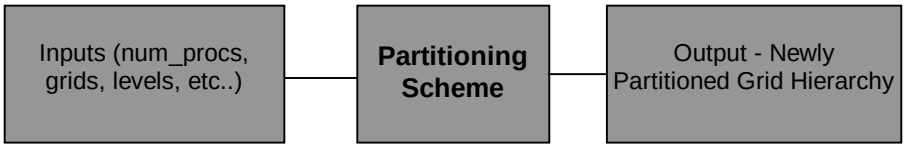


Figure 4.5 Characterization & Evaluation Process flow

A very good load balance is achieved because at the point imbalance, the bounding box that is under consideration is broken by an amount the serves to satisfy the imbalance and the processors receive almost equal amounts of load. However, the drawbacks to the method are: a) the quality of boxes generated is bad (section 4.2.3.2) and b) a single box may be recursively broken over and over again depending upon the size of the bounding box. These are addressed in the new partition method.

4.2.2 Flow diagram for original partition method

There are two traversals made of the list of work elements, one to calculate the average work and one to partition the load. Hence the complexity of the algorithm is O(n). The detailed algorithm 4.1 can be found in Appendix A. The flow diagram *FD1* below illustrates the partition process.

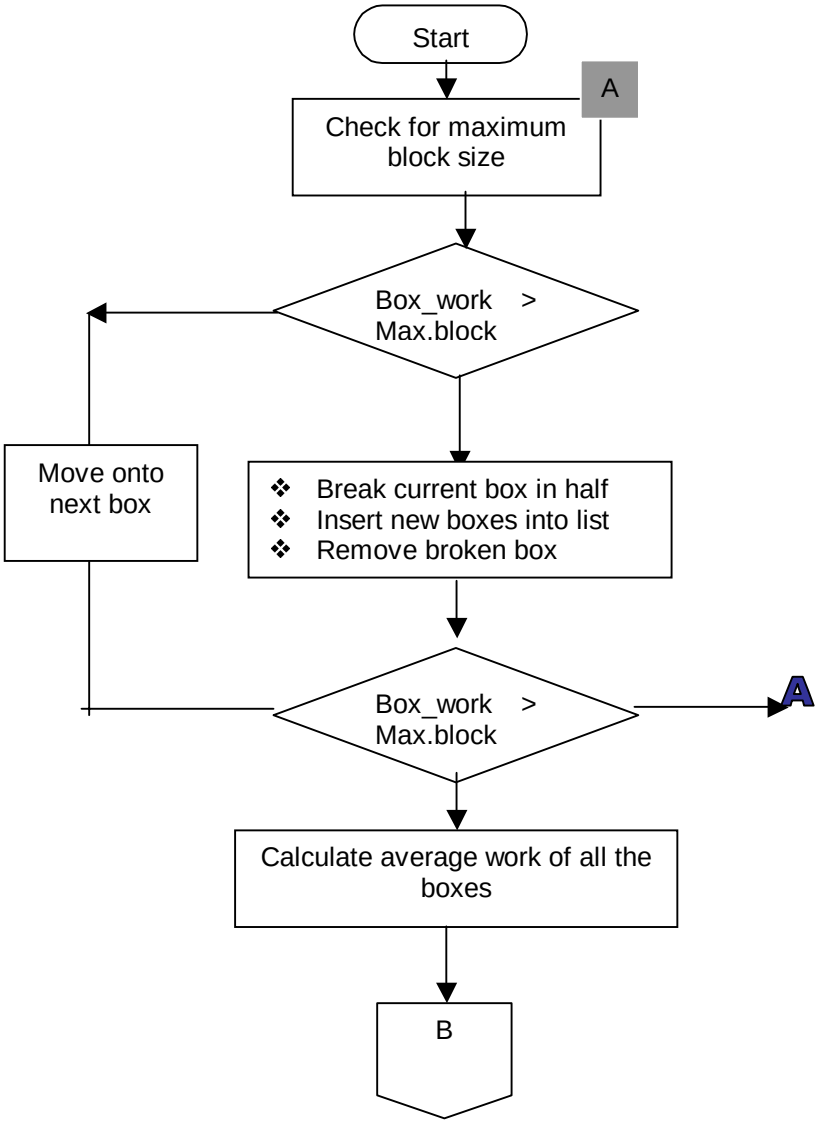


Figure 4.6 Flow Diagram FD1: Original Partition Method

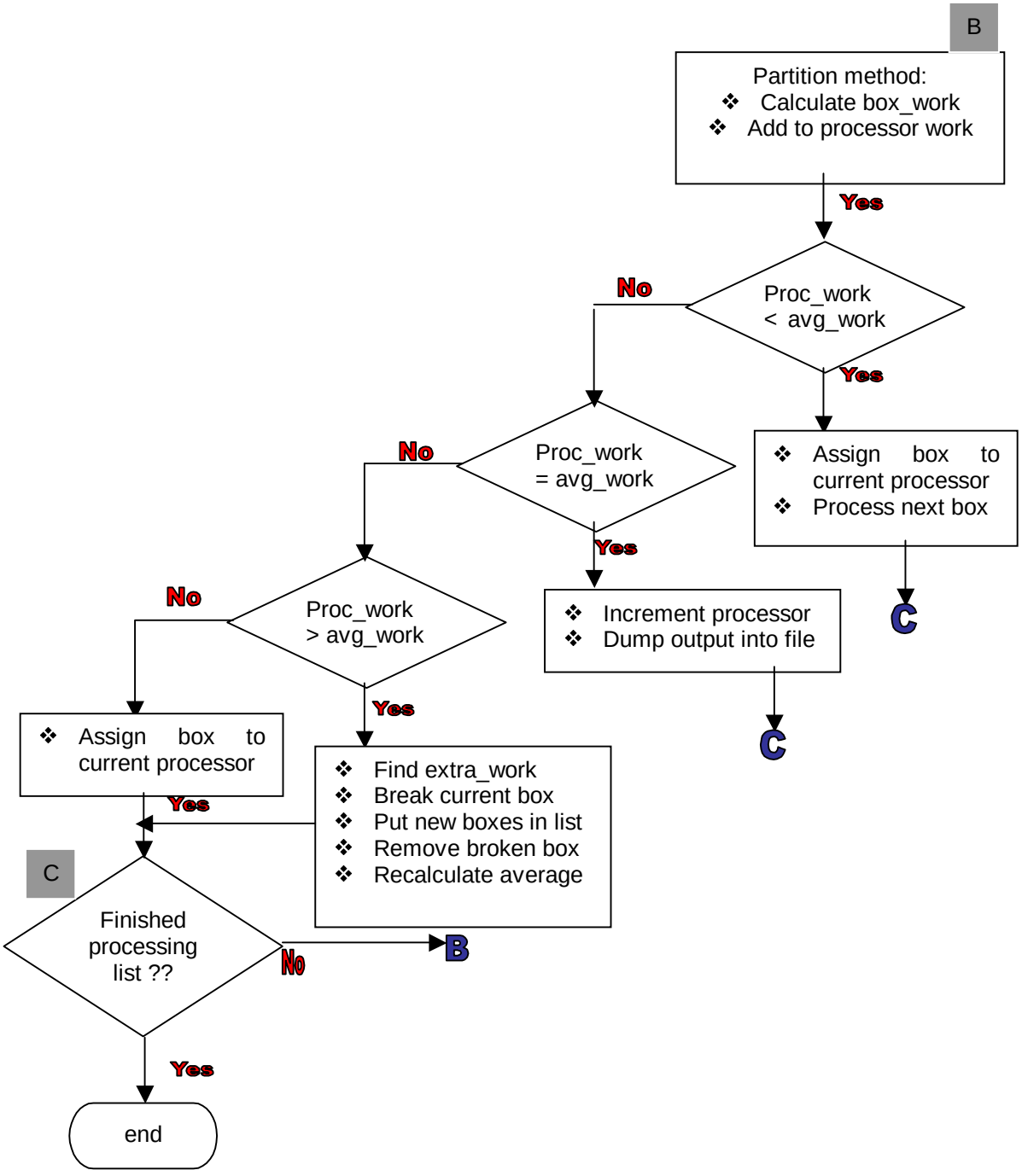


Figure 4.7 Flow Diagram FD1 (Continued …): Original Partition Method

4.2.3 Analysis of the optimized partition method

We shall refer to the initial partition method as *method1* and the optimized partition method as *method2*.

4.2.3.1 Complexity Analysis

The amount of time required to partition is of O(n) because we parse the bounding box list once during the partition process. Although this is the same as *method1*, it performs better by a constant factor, because the traversal to calculate the average work is eliminated as the total work is calculated as and when the boxes are read in, recursive breaking of the boxes is avoided and number of boxes generated is lesser. This however is slightly offset by the fact that, in *method2*, at the point of imbalance, we parse the bounding box list and check to see if a bounding box of suitable size is present. If the imbalance is at the very end of the list or if the box resides at the beginning of the list, then, we make a very few comparisons else we parse the whole list.

4.2.3.2 Criteria Optimized

Although method 1 offered very good load balance, there were a few disadvantages to this method. They are: a) the quality of boxes generated is bad (section 4.2.3.2) and b) a single box may be recursively broken over and over again depending upon the size of the bounding box, c) the number of boxes generated were more affecting the quality of implementation. The optimized technique has addressed these issues as well as some other criteria that *method1* has not addressed. Following are the criteria:

- **Box quality**

During the partition process [Algorithm 4.2.2, step 3], when the point of imbalance is reached, the current bounding box is broken by an amount that is required to satisfy the imbalance. The quality of boxes formed is thus driven by the amount of imbalance, as there is no evaluation of the size of the boxes being created. *Method2* addresses this issue by checking to see if the amount of

imbalance is lesser than a predefined minimum block size (appendix b). If the test is true, the current processor is not allocated any more work. Thus we avoid formation of bad quality boxes, but have to live with a slight imbalance. Another case where quality of boxes is preserved is when average work is less than predefined minimum block size. In such a situation, *method2* sets the average work to be equal to the minimum block size. This avoids the situation of producing very small boxes.

- **Aspect ratio**

  Aspect ratio is defined as the ratio of the longest side to the smallest side. We consider only the longest axis of the bounding box and break along that direction. For example,

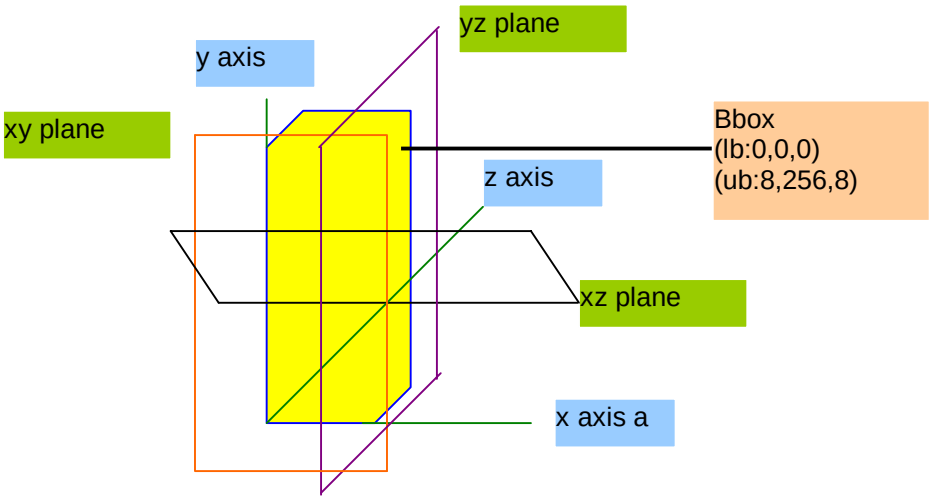


Figure 4.8 bounding box and its parameters

consider a box with dimensions: lower bound(0,0,0) and (16,256,16). In figure 4.6 above, we consider breaking the box along the xz plane, as the length of the bounding box is longest in the y direction. Breaking the box along any of the other axes would result in bounding boxes of a very uneven size.

- **Recursive breaking of the boxes**

Method1 breaks the boxes under consideration recursively. This could have a multi-fold impact: on the quality of boxes as mentioned above and recursive breaking of boxes ie., if the bounding box to be broken is of a very large size and the average work is very small, this box is broken repeatedly in a recursive fashion. In order to avoid the above situations, *method2* parses the list of bounding boxes from the current bounding box onwards. If it finds a bounding box of size equal to extra work that balances the workload; it assigns this bounding box to the processor. In the event that such a box is not found, it current bounding box is broken. This results in savings in the breaking boxes effort, which results in a considerable savings in time.

- **Reducing the number of boxes**

Before the partition method, a check is made to see if the all the boxes a less than a maximum block size because dealing with boxes of extremely large size could be unwieldy. Method 1 breaks such a box into exactly 2 parts, the first box being equal to Maxblock size. It rechecks the second block recursively and breaks it into 2 parts, until the Maxblock condition is met. In order to avoid recursive checking and breaking, method 2, calculates the number of boxes the bounding box has to be broken in a single step and a second step it breaks the box into the required number. This results in a considerable savings in breaking and checking time.

- **Improved performance (time)**

This is elaborated in section 4.2.3.1 . Chapter 6 presents the comparison of the two partition methods in terms of time, communication overheads, and the memory copies. Algorithm 4.2 in appendix A explains optimized partition method. The text in “bold” font denotes the optimization that was performed.

4.2.4 Flow diagram for optimized partition routine

The flow diagram *FD2* for the optimized partition method is presented below. Appendix A presents algorithm 4.4

A

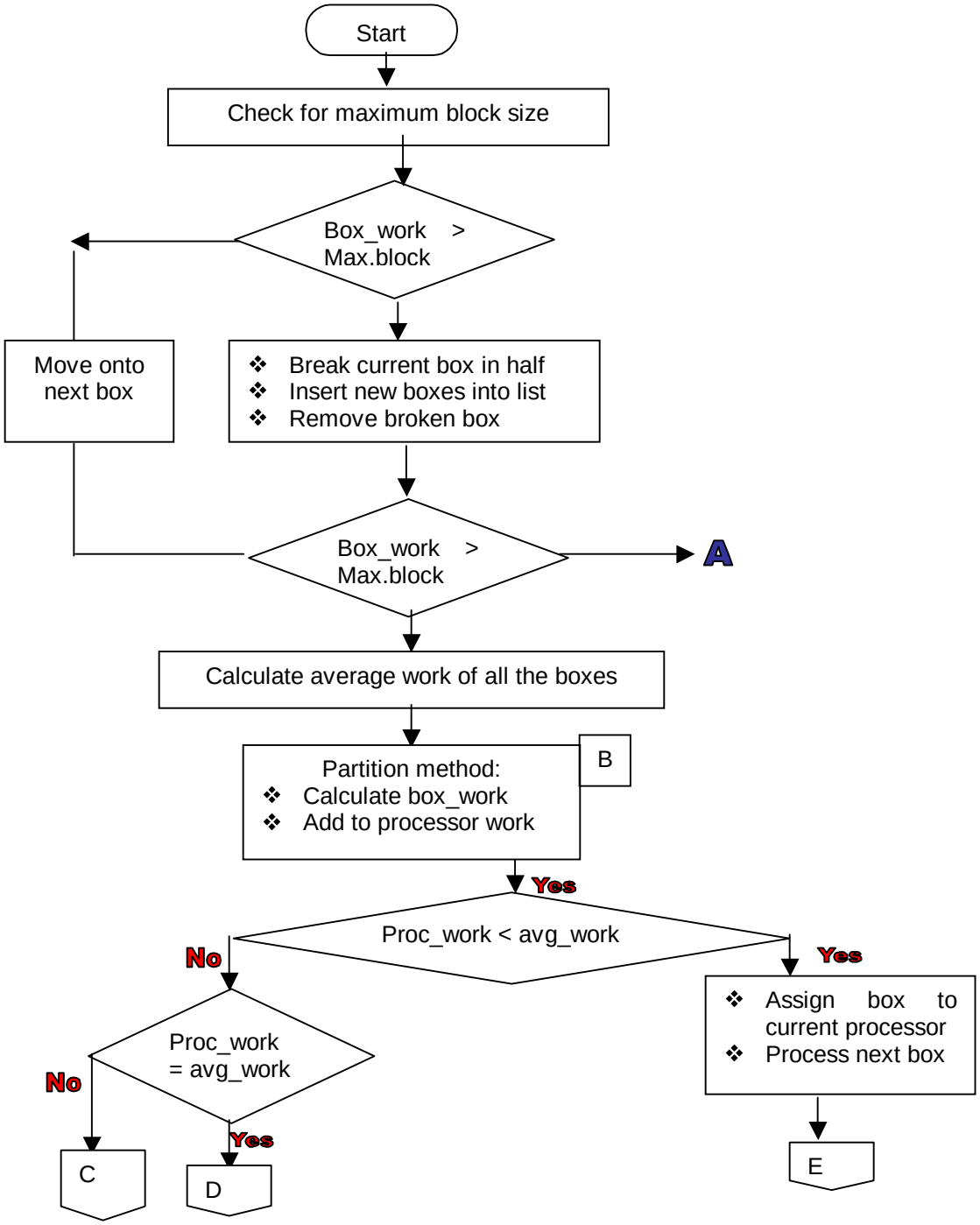
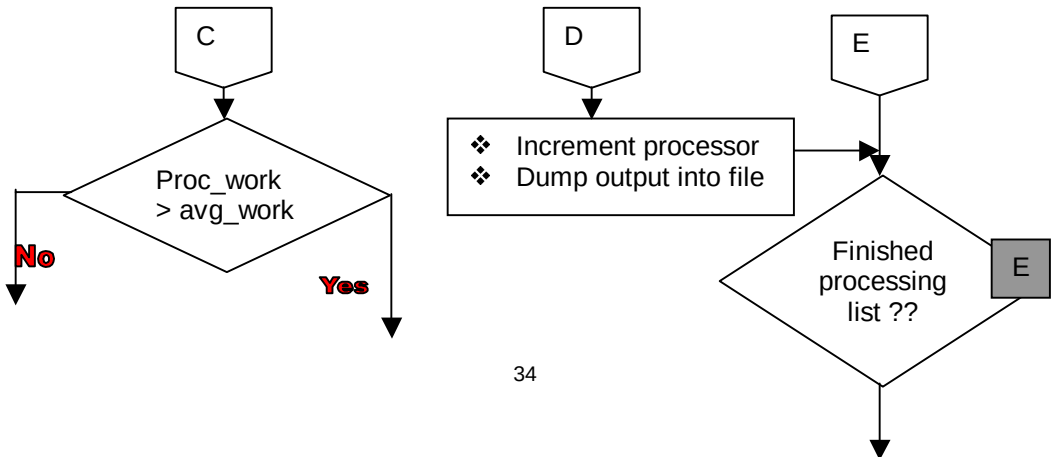


Figure 4.9 Flow Diagram FD2: Optimized Partition Method



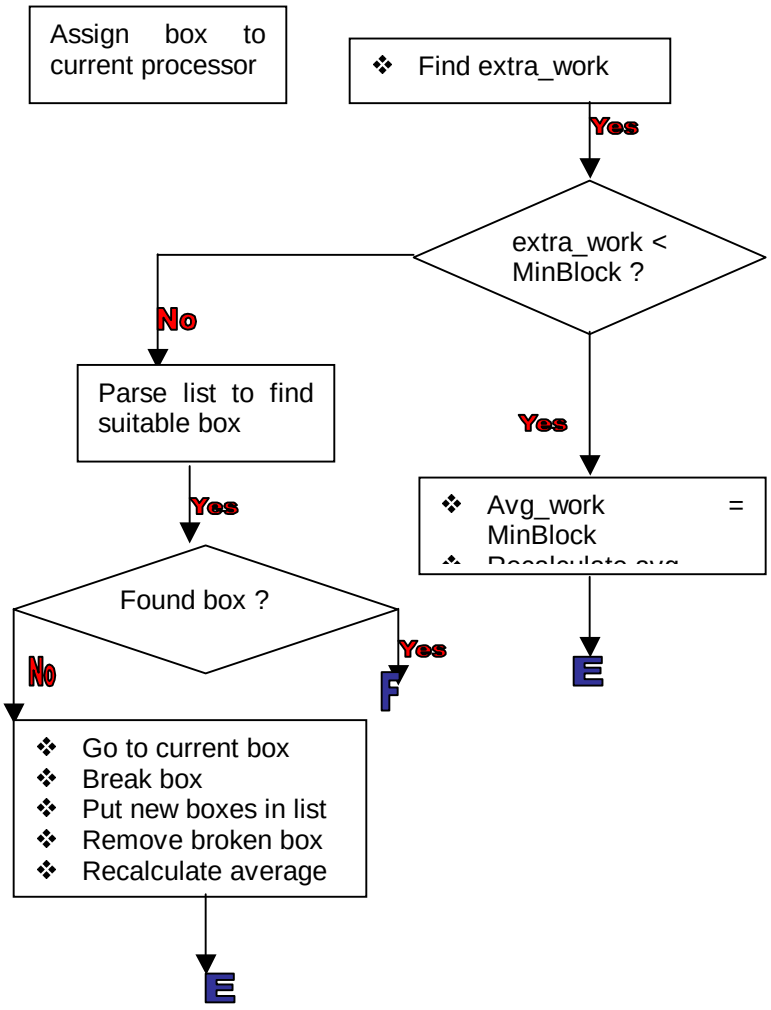


Figure 4.10 Flow Diagram FD2 continued..: Optimized Partition Method

Chapter 5

Design of the simulator

This chapter presents the design and implementation of the AMR Simulator. The AMR simulator is a module that measures the characterization criteria of the distribution techniques.

5.1 The AMR Simulator

We use four criteria to characterize distribution mechanism for AMR adaptive grid hierarchies, viz. load balance, distribution quality, grid interaction overheads (inter-processor communication and memory copy), and data-movement overheads as mentioned in section 3.1

The AMR simulator was designed to measure the above-mentioned criteria. In addition, it also measures the time taken for various partitioning schemes. Section 5.3 discussses th einputs to the simulator and the outputs generated. Figure 5.2 illustrates the same.

The input to the simulator is obtained by the following 3 steps which are as follows:

1. A trace of the Grid hierarchy is obtained by performing a run of the application for a single processor. and the resulting parameters are dumped into a trace file.

2. This trace file is fed to a partitioner. The partitioner implements the partitioning scheme of choice. The partitioning scheme allocates various bounding boxes at different levels and timesteps to processors. This result is dumped to an output Parameters file.

3. This Output Parameters file is the input file to AMR simulator. The AMR simulator measures the different parameters that characterize the distribution mechanisms.

Figure 5.1 below shows a high level view of this process.

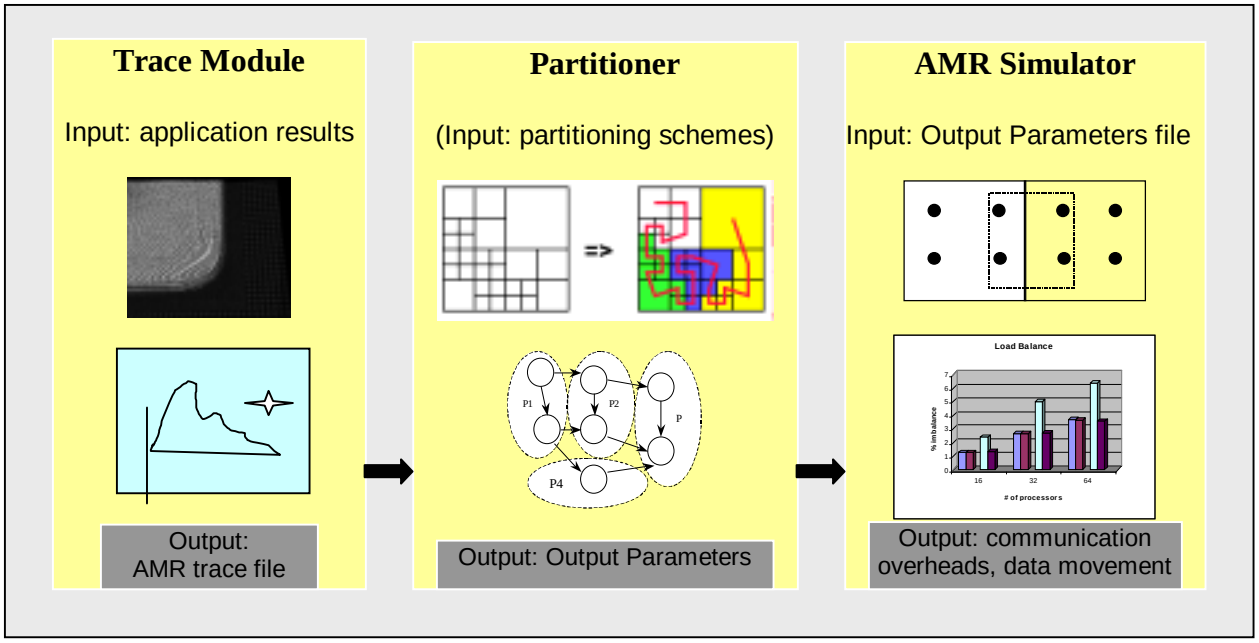


Figure 5.1 Partitioning process

5.2 Input/Output

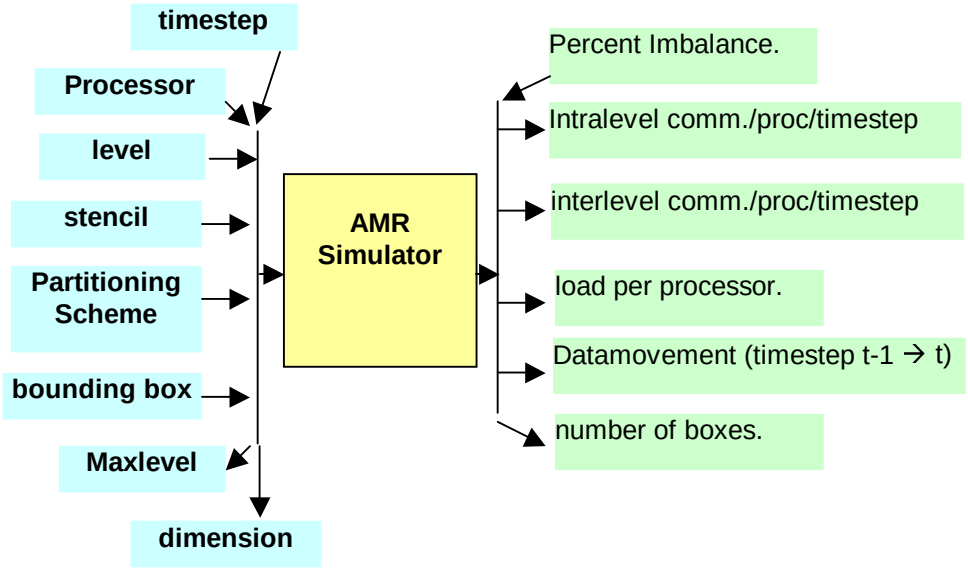


Figure 5.2 Inputs and Outputs to the Simulator

The input to the simulator is essentially a list of bounding boxes obtained as a output of the partitioning schemes, across processors at different levels of the hierarchy. The simulator hence reads in time step, processor number, level and the corresponding bounding box parameters.

Additionally, the user must specify the rank (2 dimension or 3 dimension), initial time step, time step (the amount by which a time step changes for instance in steps of 1 , 4 or 8 etc), and the filenames that can optionally be specified to redirect output values of the parameters that being measured (communication overheads, data movement, time, percent imbalance etc), variable number of processors and the stencil size through the command line interface.

5.3 Measurement of Characterization Criteria

5.3.1 Measurement of intra-level communication

At each time step, some amount of communication is present between processors at the same level. This is measured as follows: Let list 1 be the boundingbox list for processor1 at level1, and Let list 2 be the boundingbox list for processor2, also at level1. bounding box (bb) of list1 is grown by a specified stencil size in all directions such that it maintains an overlapping zone at its boundaries with its neighboring processing elements (bb in this case) for synchronization of data during the updating process. This overlapping zone is called the '*ghost region'*. Figure 2.2 and 5.3 illustrates such a ghost region for a bounding box. The bounding box can be logically divided into 3 regions of interaction : face, corner and edge. These interaction boxes of each bounding box is intersected with each of the interaction boxes on list 2. The intersection of these boxes detects the presence of intra-level communication.

The inputs to determine intra-level communication are:

- stencil size, bblist at a particular timestep,level,processor , rank, maximum number of levels, number pf processors (num_procs), time

You can find the implementation of measurement of intra-level communication in algorithm 4.5 in appendix a.

Ghost communication: Figure 5.3 shows the ghost communication between two processors.

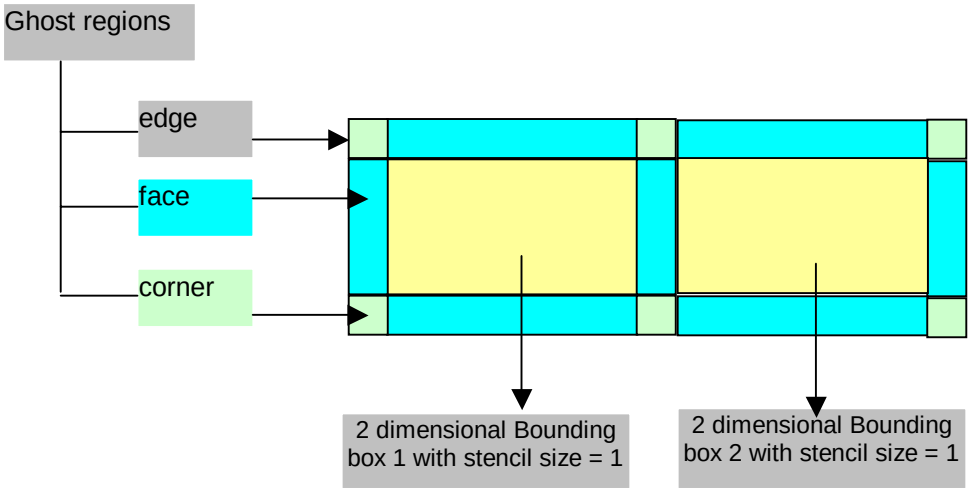


Figure 5.3 Ghost communications betweeen 2 bounding boxes

5.3.2 Measurement of inter-level communication

When a refined grid is created, its boundaries are usually interior to some coarser grid values. After updating the function values on a fine grid, the underlying coarse grid values are updated through restriction. Hence we have interlevel communication. This is measured for each of the timesteps, across the processors at different levels of refinement. The measurement process is as follows

- At a given timestep, each bb for a bblist for a given processor and level is compared to each bb of a bblist that is either one level up (child) or one 1 level below(parent)
- If there is an intersection between the two bbs, interlevel communication exists and it is measured in the same way as described in the section above "measurement of intralevel communication".

The algorithm used for determine the interlevel communication for a given processor at a given timestep is:

Algorithm for interlevel communication

```
Find out the total number of levels
If the number of levels > 1 then,
  for each level (starting from level=1, ie., bounding boxes of level 0 have children at level 1,
we     are concerned with the children)
          for each bounding box
          // interlevel communication is twice the boundary points of itself (because both
          // prolongation and restriction are involved
                    calculate the upper bound, lower bound, step size
                              interlevelcomm = 2* ((ub[x]-lb[x])/step[d]) + 1
```

5.3.3 Measurement of Data Movement

At the end of each timestep, the partitioning scheme redistributes different bb to different processors due to the change in the grid hierarchy as a result of refinement. Hence bounding boxes that belonged to one processor may/may not belong to it at the next step. Therefore, data movement is the number of points sent from one processor at on etimestep to another afterevery regridding. Data movement is measured as follows:

At each timestep, compare the list of bbs at different timesteps. If there is an intersection between the 2 bbs, then there indeed is data movement and it is measured by the size of the intersection bb. The inputs to determine data movement are:

- 2 bounding box lists, one holding the boxes of the current timestep and the other holding the boxes of the previous timestep., number of processors (num_procs), number of levels

The algorithm is to determine data movement is as follows:

Algorithm to determine datamovement

```
At each timestep
for(processor=0 to processor =num_proc-1)
        for(each box in bblPrev)
                intersect with each box in bblCurrent to obtain the
intersection_bb
                if( intersection_bb != NULL)
                        data movement+= intersecton_bb.size( )
```

5.3.4 Interlevel and intralevel memory copy

This is the amount of grid memory copies for co-located grid components.

It is found out in the same way as intra/interlevel communication except that at each timestep, we are considering the communication overheads within each processor.

5.3.5 Percentage load imbalance

Algorithm below outline the processof determining the percentage load imbakance:

inputs: Output parameters file, number of timestep, average work, processor number, number of processors

Algorithm to determine percentage load imbalance

Sum of loads on all the bounding boxes of all the processors.

$$\text{Total load} += \sum_{i=0}^{nproc-1} \sum_{j=1, i \neq j}^{n} (\text{size}(bb_i))$$

```
load on each processor(average work) = Total load / nproc
for each bounding box
        difference in work = average work ~ processor work
total difference in work += difference in work
average difference in work = total difference in work/num_procs
percent imbalance = (average difference in work*100)/average work
cumulative percent imbalance += percent imbalance
final percent imbalance = cumulative percent imbalance/number of timesteps
```

5.3.6 Number of boxes

This is equal to the number of lines obtained in the OutputParameters file that is obtained from the partitioner and the input to the AMR simulator.

Chapter 6

Experimentation and results

The experimental evaluation consisted of two sets of experiments: the first evaluated the performance of the partitioning and load-balancing schemes without the optimization, while the second evaluated the same schemes with the optimized partitioning routine.

6.1 Experimental Setup

To evaluate the six distribution/load-balancing schemes outlined in Section 5 we use grid adaptation traces from 2 different AMR applications:

1. 3D Buckley-Leverette equation kernel (BL) that was used in oil reservoir simulations.
2. 3D Wave equation kernel (Wave) that was used in a numerical relativity simulation.

These applications demonstrate two very different refinement patterns. In BL, the refined grids track a front, moving diagonally across the grid as shown in figure 1.1. The Wave kernel trace consisting of refined grids, tracks the crest of the wave as it dissipates at the center of the grid as shown in figure 1.2. The two traces were generated from a single processor run with 5 levels of factor 2 refinement.

6.2 Experimental process

After obtaining a trace as above, it was then fed to a partitioning module that partitioned the boxes across the required number of processor using each of the six schemes. The simulation was run for 16, 32 and 64 processors. An AMR simulator (Chapter 5) then evaluated various costs for the partition generated. The evaluation was performed using architecture independent measurements.

6.3 Metrics

Four criteria were used to characterize distribution mechanism for AMR adaptive grid hierarchies, viz. load balance, distribution quality, grid interaction overheads (inter-processor communication and memory copy), and data-movement overheads. These criteria are described in section 3.2. The measurements for the different metrics are summarized in table1 below. In case of communication and data-movement overheads, this corresponds to the amount of information communicated. In case of distribution quality metrics this corresponds to the number of grids per processor and the percentage load imbalance (a perfect load balance corresponds to 0%). The motivation for keeping the measurement architecture independent was to enable them to be used to select appropriate schemes for a wide range of architecture using a very simple architectural description.

Table 6.1: Metrics and measurement

Metric	Measurement
Load Balance	Percentage load imbalance
Intra-level/Inter-level communication overhead	Megabytes communicated
Intra-level/Inter-level memory copy	Megabytes copied
Data movement	Megabytes moved
Distribution time	Seconds
Distribution quality	Number of Boxes (x 1000)

The plots presented in this section represent cumulative costs for each scheme. The vertical axis in each of these plots is the relevant measurement for each metric, while the horizontal axis is the number of processors. In this experimentation we used three configurations of 16, 32 and 64 processors.

6. 4 Results

The plots in Figure 6.1 show the intra and inter level communication overheads, while Figure 6.2 shows the memory copies in each case. In both the sets of plots, the IGD distribution scheme tends to be particularly bad. The other schemes tend to be comparable with the SFC and CGD being slightly better than the rest. The plots in Figures 6.3 and 6.4 show the overheads due to dynamic distribution and load balancing. Once again IGD results in the maximum data-movement while CGD has the least data movement. The ILD scheme requires the maximum time for dynamic distribution and load balancing. Once again IGD results in the maximum number of boxes, however, it results in a near perfect load balance. ITB produces the overall best quality distribution with SFC and CGD schemes comparable. ILD produces the most load imbalance.

6.4.1 Buckley-Leverette (BL) Application Trace

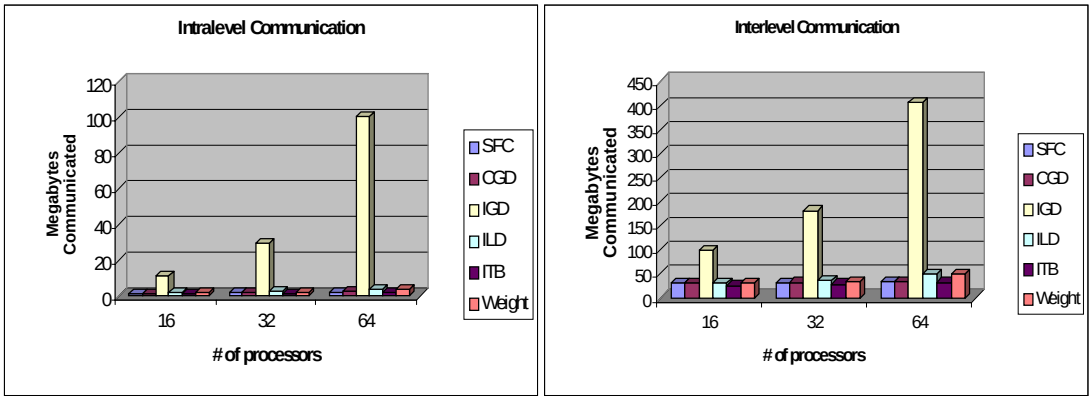


Figure 6.1 BL - Communication Overheads

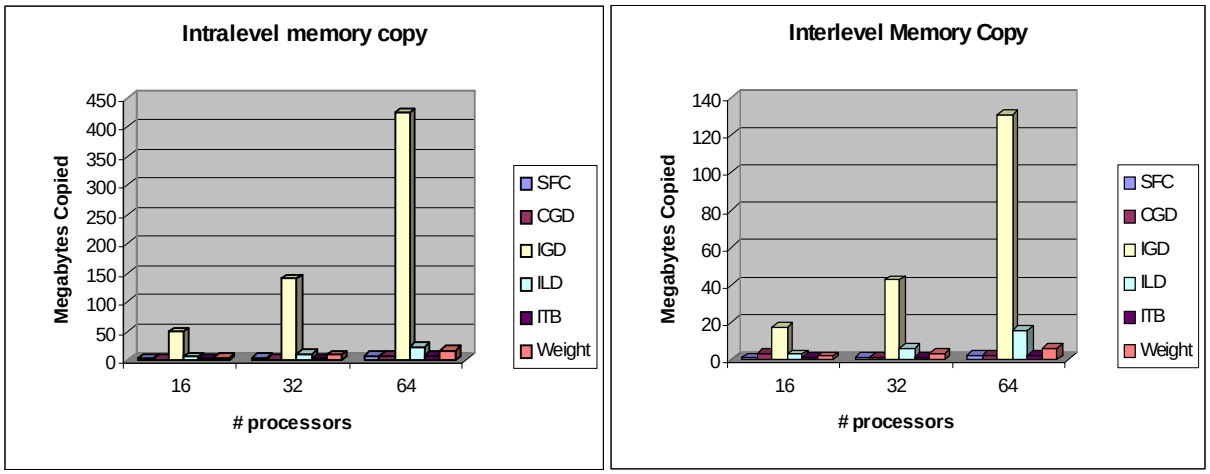


Figure 6.2 BL-Intralevel & Interlevel Memory Copies

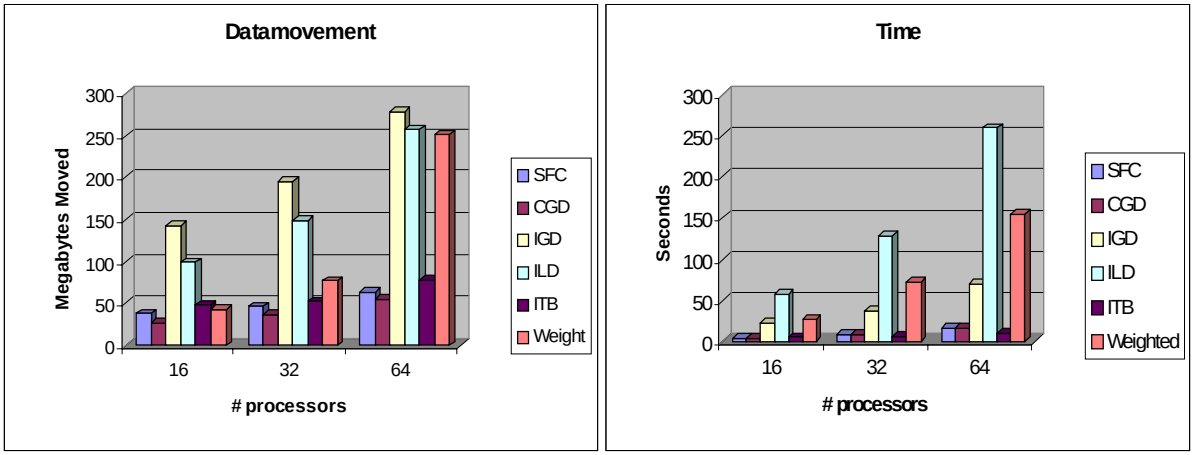


Figure 6.3 BL-Data movement and Distribution/Load-balancing Time

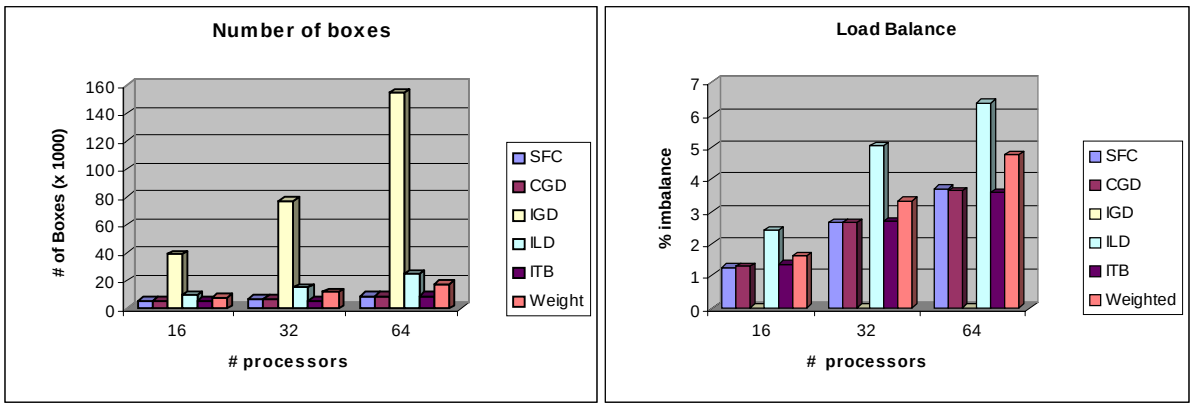


Figure 6.4 Distribution Quality (Number of Boxes, Load Balance)

6.4.2 Analysis

This application is a simulation moving diagonally across the grid (figure 1.1). Refined levels are formed across the diagonal of the domain and move forward as the simulation progresses. Hence, the application is dynamic in nature leading to larger amounts of data movement. Consequently, intra level and inter level communication are concentrated along the diagonal, and so is the work associated with the grids. Hence, Partitioning schemes, that inherently produce lesser data movement and communication overheads would be best suited for such applications. Following is the analysis of the six schemes with respect to the results plotted above. (Figures 6.1 – 6.4).

a) Space Filling Curves

This technique (section 3.2.1) maintains locality across different levels of hierarchy. Thus interlevel communication is less compared to other techniques. Even though the wave progresses along the axis diagonally, boxes are redistributed among the neighboring processors when partitioned and so data movement is less compared to other schemes. The distribution quality is better as the number of boxes generated is very less and hence the intra level memory copies and inter level memory opiesare also minimized.

b) Combined Grid Distribution

This scheme (section 3.2.3) does not focus on reference of locality. Hence, it tends to have more communication overheads compared to SFC. From the plots above it can be seen that communication overheads are more than SFC. Inter grid operations are very expensive because if the grids are distributed across the entire set of processors, then there is more communication. Data movement is less because , at the levels where there is no refinement, existing grids are redistributed to the same processors . The distribution quality and time overhead are comparable to SFC.

c) Independent Grid Distribution

This scheme (section 3.2.2) is based on a simple approach of distributing individual grids across the processors irrespective of locality, level or size of the grid. Inter level and Intra level communication rises exponentially with the increase in number of processors because every grid on a processor has to communicate with other grids on other processors. Load Balance is very good since every box is exactly divided among processors and the effort required to do so is negligible due to the simplicity of the algorithm.

d) Independent Level Distribution

This technique (section 3.2.4) exploits parallelism within a level and provides control over inter level communication. This can be seen from the plots above. Inter level communication is minimum compared to all the other schemes, but the intra level communication scales up. This is because grids at every level are distributed among processors and pose the same problem as seen in the case of Independent Grid Distribution.. In ILD, the points at different levels under a domain are distributed among the same processors and so the inter level communication is much less. This should increase inter level memory copy significantly which is seen from the plot above. Distribution Quality is better since Load Imbalance; number of boxes and the effort taken are comparable.

e) Iterative Tree Balancing

The technique (section 3.2.5) views this problem as a graph-partitioning problem with underlying constraints. Partitioning starts from the highest level of grids and progresses making a breadth first search on the neighbors and then on the parents, resulting in immediate neighbors and parents in the same processor. This maintains inter level communication and intra level communication to a minimum as seen in the plots above. Data movement is a drawback of this scheme because of the following reason. When a new grid is formed at any level the graph changes and so does the representation of the nodes, their neighbours and their parents. Since partitioning is based on this

graph the boxes usually end up in neighboring processors as compared to the previous iteration of the simulation.

f) Weighted Distribution

The approach of this scheme (section 3.2.6) is to prioritize the focus on constraints according to the characteristic distribution of refinement and dynamic behavior of the application. The constraints are inter level communication, intra level communication and data movement. In this experiment equal priority i.e. weight is assigned to all the constraints so all the overheads are kept under control. Plots show that all these three constraints are well in bound as compared to the other implementations.. The number of boxes formed is less which keeps inter level memory copy and intra level memory copy less. The main disadvantage of this scheme is the overhead of its implementation i.e. the effort required in terms of time taken for its execution. This is because every box to be partitioned calculates the total weight for all the processors to know its affinity towards a particular processor. In order to calculate this, a traversal is made on all the boxes of the previous iteration, which makes it an $O(n^2)$ algorithm.

Tables 4.2, 4.3 and 4.4 present some results for the various characterization criteria for the Buckley Leverette Application for all the techniques. All the results are in terms of 10^6 units.

Table 6.2: Intralevel & Interlevel communication and Data Movement for Buckley Leverette application

Scheme	Intralevel communication			Interlevel communication			Data Movement		
	16	32	64	16	32	64	16	32	64
SFC	1.244	1.526	1.97	30.58	32.156	33.37	3.15	4.46	7.17
CGD	1.31	1.59	2.38	30.83	32.38	33.45	1.01	1.53	7.24
IGD	11.208	29.32	100.5	100.5	181.3	407.42	49.498	140.275	424.9
ILD	1.71	2.27	3.369	31.11	37.66	50.45	5.55	10.28	22.26
ITB	1.302	1.28	2.009	29.15	26.447	32.86	3.18	3.474	7.209
Weight	1.47	2.08	3.369	31.18	34.26	50.45	4.69	8.39	16.62

Table 6.3: Intralevel & Interlevel Memory Copies and Time for Buckley Leverette application

Scheme	Intralevel Memory Copies			Interlevel Memory Copies			Time		
	16	32	64	16	32	64	16	32	64
SFC	3.15	4.46	7.17	1.01	1.36	2.284	4.97	9.166	17.536
CGD	1.01	1.53	7.24	3.19	1.542	2.283	5.06	8.92	17.286
IGD	49.498	140.275	424.9	17.27	42.56	131.27	23.448	37.983	70.448
ILD	5.55	10.28	22.26	2.98	6.19	15.47	58.3	127.8	259.05
ITB	3.18	3.474	7.209	1.14	1.189	2.24	5.416	7.126	10.82
Weight	4.69	8.39	16.62	1.83	3.068	6.27	27.24	72.71	155.56

Table 6.4: Number of boxes and Load Balance for Buckley Leverette application

Scheme	Number of Boxes			Load Balance		
	16	32	64	16	32	64
SFC	6090	7162	9306	1.275	2.653	3.71
CGD	6090	7162	9306	1.285	2.643	3.649
IGD	38672	77344	154688	0	0	0
ILD	9870	14974	24974	2.41	5.04	6.39
ITB	6086	6087	9306	1.367	2.71	3.596
Weight	8248	11784	18052	1.624	3.34	4.78

6.4.3 3D Wave Application Trace

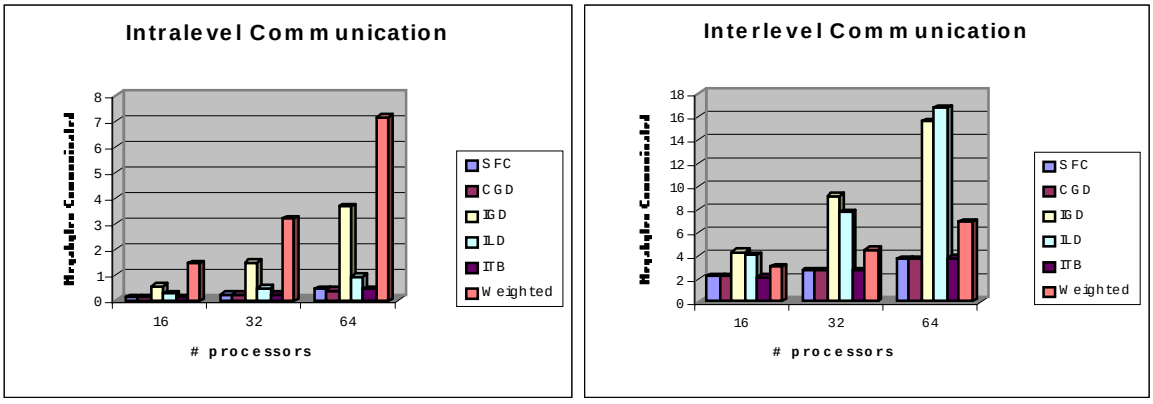


Figure 6.5 Wave – Communication Overheads

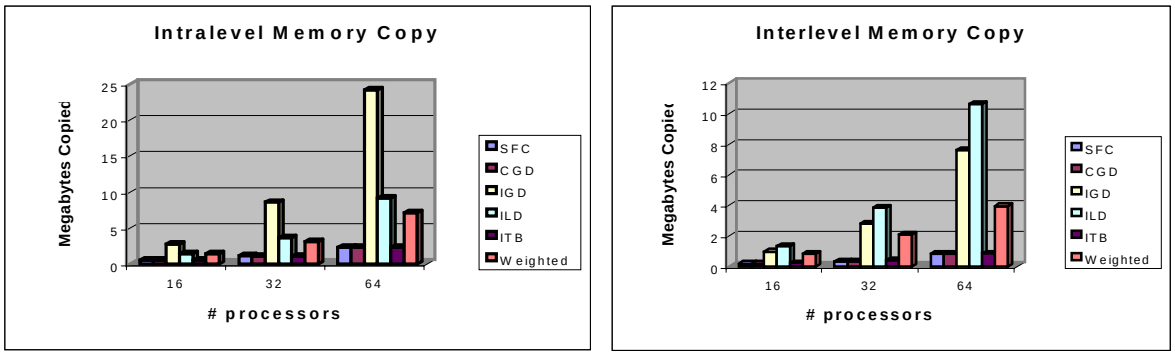


Figure 6.6 Wave – Communication Memory copies

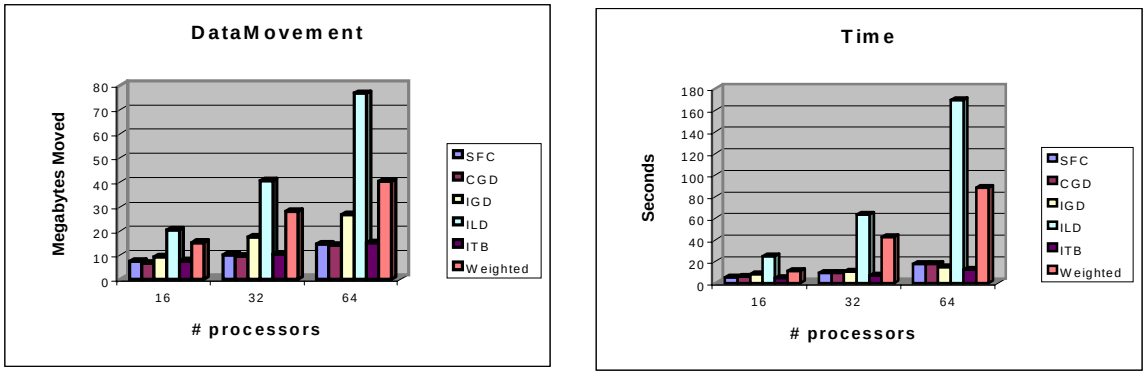


Figure 6.7 Wave – Data Movement and Load Balancing/Distribution Time

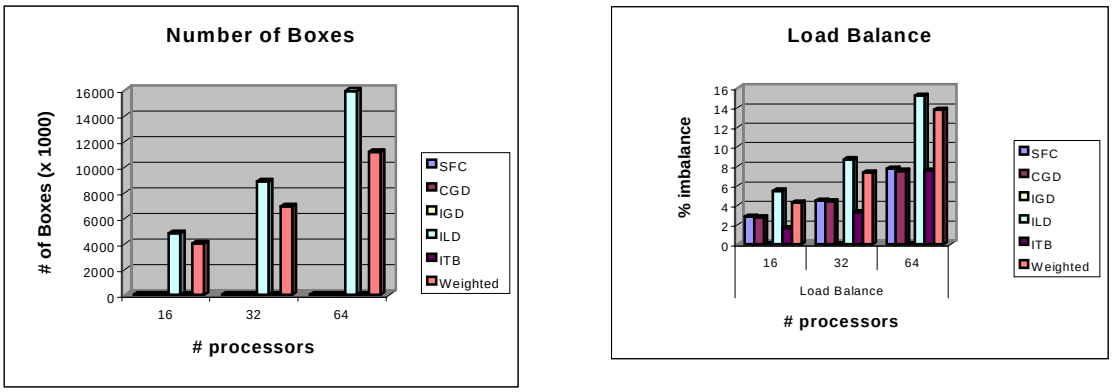


Figure 6.8 Wave – Distribution Quality (Number of Boxes, Load Balance)

6.4.4 Analysis

This application is a 3 dimensional wave simulation moving vertically across levels of the grid and so the refinement is focused vertically across levels throughout the simulation. Refined levels are formed across levels of the domain and shift up and down as the simulation progresses. Hence, the application is dynamic and has significant amount of inter level communication and data movement. The intra level and inter level communication is more throughout the center of the domain. Partitioning schemes, that control more of inter level communication and also focus on data movement overhead would be best suited for such applications. Following is the analysis of all the six schemes with respect to the results plotted in figures 6.5 - 6.8 for the 3 dimensional wave application trace.

a) Space Filling Curves

The only significant overhead is the data movement overhead compared to the Buckley-Leverette application. The reason for this is: in Buckley-Leverette, the wave front moves slowly along the diagonal and so the major movement of grids would be around the diagonal and between few levels. But in wave application trace the refinement is heavily done across levels and so when the grids are redistributed, due to load balancing criteria the grids from the previous time step end up in different processors.

b) Composite Grid Distribution

This technique performs quite similar to space filling curves for reasons explained above.

c) Independent Grid Distribution

This scheme behaves as expected in both Wave application and Buckley leverrete application, except for inter level communication for wave application. This is because at every level the number

of grids are less. So when partitioned, a certain domain resides in one processor from the coarsest to the finest level which makes the interlevel communication less. This is a special case and if the domain of computation is larger or the refinement is more sparse within the domain, then interlevel communication would be higher as seen in Buckley-Leverette application.

d) Independent Level Distribution

Interlevel Communication is the prime focus of the scheme and is low, as seen from the plots above. Data Movement is less since refinement adds grids at the finest levels, and only grids at these levels add to the data movement overhead. Number of Boxes, time taken and Load imbalance are slightly higher than all the other schemes except independent grid distribution. Intra level communication is not addresses in this scheme which is a big drawback.

e) Iterative Tree Balancing

The application characteristic does not have a significant impact on the results of this scheme. Inter level and intra level communication is controlled and rest of the parameters from communication overhead within a processor, distribution quality to time overhead are well below other schemes. The only drawback is data movement, which is inherent characteristic of this scheme.

f) Weighted Distribution

Equal priority is assigned to all the constraints for evaluating the wave trace application. The results of this scheme show significant changes compared to other schemes when evaluated from Buckley Leverette to Wave trace application. Compared to Buckley Leverette application, intra level communication, memory copies are higher and data movement is lower than most other schemes. This shows that giving equal priority to all the constraints benefits inter level and data

movement but not intra level communication. Since the refinement in this application is more across levels, the existing grids have the neighboring grids and parent grids in the same processor as the previous iteration. Thus data movement is less. The increase in intra level communication comes from the newly formed grids at the finer levels, which have more work. These grids do not have neighbors till the next iteration as they are newly formed also add to imbalance in load. In order to alleviate this, they are assigned randomly to any processor who has work assigned below average and thus the intra level communication is increased.

Tables 4.5, 4.6 and 4.7 present some results for the various characterization criteria for the Buckley Leverette Application for all the techniques. All the results are in terms of 10^6 units.

Table 6.5: Intralevel & Interlevel communication and Data Movement for Wave application

Scheme	Intralevel communication			Interlevel communication			Data Movement		
	16	32	64	16	32	64	16	32	64
SFC	1.244	1.526	1.97	30.58	32.156	33.37	3.15	4.46	7.17
CGD	1.31	1.59	2.38	30.83	32.38	33.45	1.01	1.53	7.24
IGD	11.208	29.32	100.5	100.5	181.3	407.42	49.498	140.275	424.9
ILD	1.71	2.27	3.369	31.11	37.66	50.45	5.55	10.28	22.26
ITB	1.302	1.28	2.009	29.15	26.447	32.86	3.18	3.474	7.209
Weight	1.47	2.08	3.369	31.18	34.26	50.45	4.69	8.39	16.62

Table 6.6: Intralevel & Interlevel Memory Copies and Time for Wave application

Scheme	Intralevel Memory Copies			Interlevel Memory Copies			Time		
	16	32	64	16	32	64	16	32	64
SFC	3.15	4.46	7.17	1.01	1.36	2.284	4.97	9.166	17.536
CGD	1.01	1.53	7.24	3.19	1.542	2.283	5.06	8.92	17.286
IGD	49.498	140.275	424.9	17.27	42.56	131.27	23.448	37.983	70.448
ILD	5.55	10.28	22.26	2.98	6.19	15.47	58.3	127.8	259.05
ITB	3.18	3.474	7.209	1.14	1.189	2.24	5.416	7.126	10.82
Weight	4.69	8.39	16.62	1.83	3.068	6.27	27.24	72.71	155.56

Table 6.7: Number of boxes and Load Balance for Wave application

Scheme	Number of Boxes			Load Balance		
	16	32	64	16	32	64
SFC	6090	7162	9306	1.275	2.653	3.71
CGD	6090	7162	9306	1.285	2.643	3.649
IGD	38672	77344	154688	0	0	0
ILD	9870	14974	24974	2.41	5.04	6.39
ITB	6086	6087	9306	1.367	2.71	3.596
Weight	8248	11784	18052	1.624	3.34	4.78

6.5.5 Optimized results:

Figures 6.9 – 6.12 show graphs of *methods 1* and *2* (called as m1 & m2) for the SFC and CGD techniques. It is seen from figure 6.12 that the criteria optimized like distribution quality for implementation is indeed lesser than *method2*, while *method1* has better loadbalance. Figures 6.9 shows the intralevel and interlevel communication while figures 6.10 show memory copies in each case. For the first set of plots, *method2* has better results, while in the second set of plots, the results are comparable.. Figures 6.11 shows the datamovement and time. *Method2* again has better performance in time,but datamovement is comparable for the two.

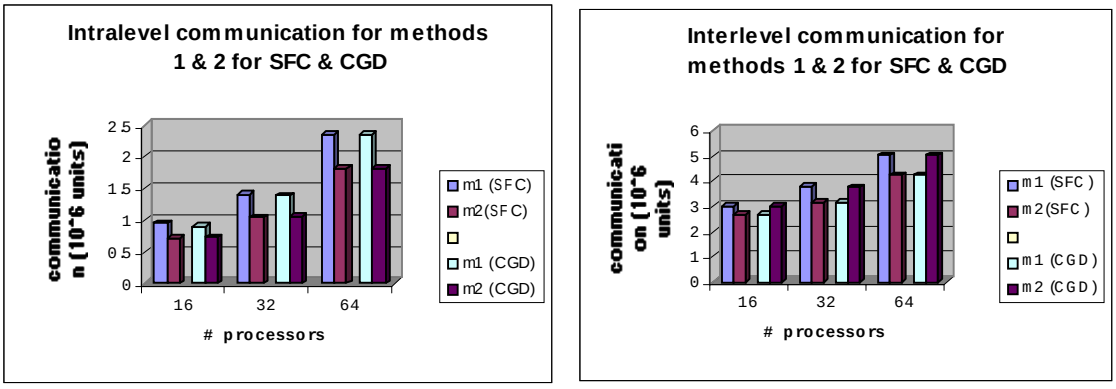


Figure 6.9 Interlevel Communication & Interlevel Communication

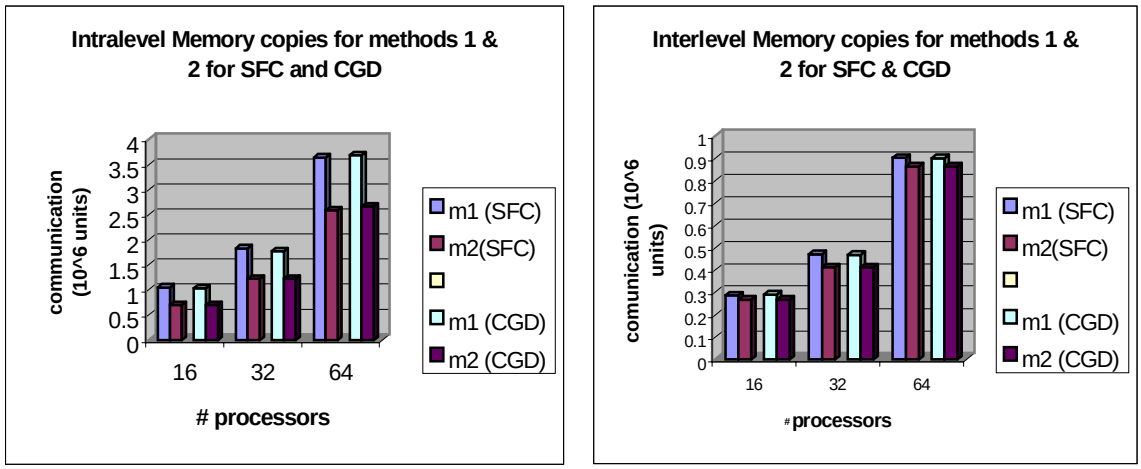


Figure 6.10 Intralevel memory copies & Iinterlevel memory copies

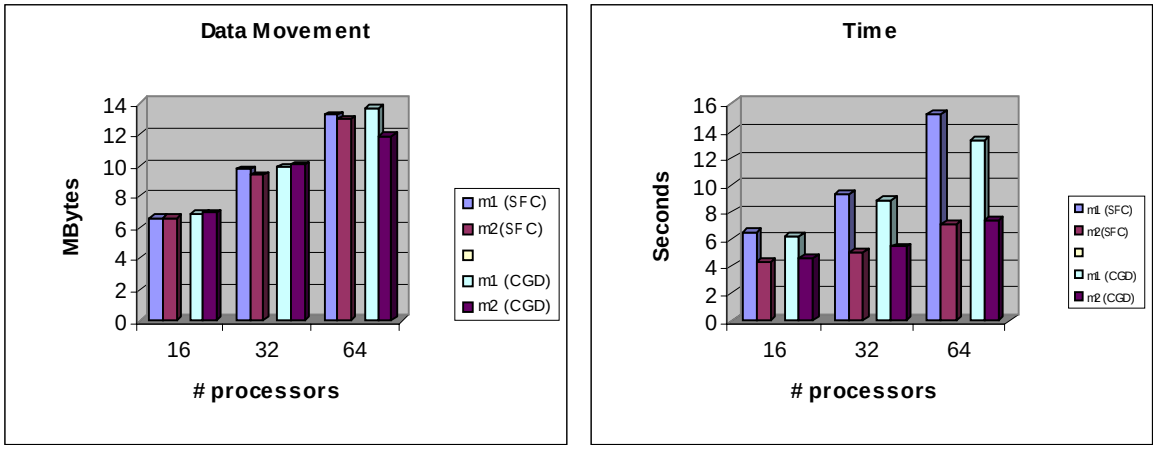


Figure 6.11 Datamovement &Time

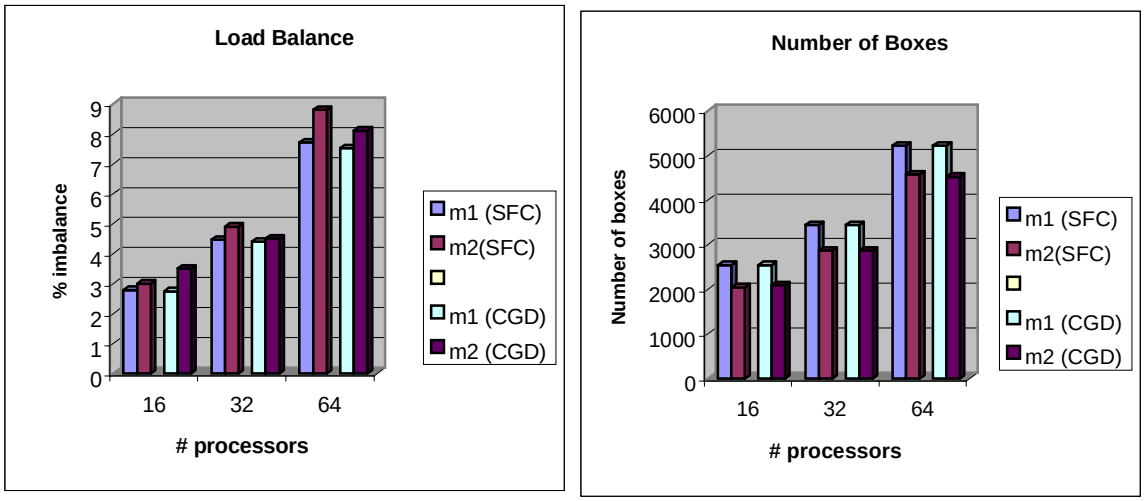


Figure 6.12 Load Balance & Number of Boxes

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

This thesis presented the evaluation and optimization of a suite of six dynamic partitioning and load-balancing techniques for distributed adaptive grid hierarchies that underlie parallel adaptive mesh-refinement (AMR) techniques for the solution of partial-differential equations. It was based upon a performance characterization of these six techniques. The evaluation consisted of the design of the AMR simulator that was used to evaluate the six distribution and load-balancing schemes. The simulator took in as its input, the Output parameters file that was created by the partitioning module as a result of the partitioning scheme. The partitioner accepted as its input the trace of grid adaptations from two 3D AMR applications.

This is part of an ongoing project for developing policy driven “smart” tools for automated distribution/load balancing of the problems in heterogeneous distributed environments. The characterization consisted of 3 metrics: Interaction overheads (inter- and intra- level communications and copies), Distribution Quality (load-balance, number of grids) and Data Movement.

The presented results show that space-filling curve, iterative tree balancing, and weighted distribution are clearly superior for all metrics. The reason is that these techniques use application information to determine the partitioning rather than pure heuristic. The weighted scheme in particular is tuned to these classes of applications and tends to do better than the other schemes.

We also see that the optimization of the partition routine resulted in a much greater performance with respect to time, intralevel interaction overheads, interlevel memory copies, better distribution quality and data movement. The reason for this is that, the new partition routine checks for the availability of a bounding box, given a size criteria based on the imbalance produced. This

built in logic avoids the recursive partitioning of bounding boxes over and over again. Hence we see a better distribution quality in terms of number of boxes, and lesser data movement.

The optimization of the data-structure for composite grid distribution technique (also called SFC technique) also resulted in a greater performance with respect to time, intralevel interaction overheads, interlevel memory copies, better distribution quality and data movement. The reason for this is that the representation of the grid hierarchy in the form of a parent-child relationship used a general tree data structure as opposed to a linked list. Hence it had lesser cost involve with respect to time and since it used the optimized partitioning scheme, it had greater performance with respect to intralevel interaction overheads, interlevel memory copies, better distribution quality and data movement

7.2 Future Work

Current work looks at completing this characterization and encoding the results into a policy rule base that can drive an automated partitioning and load-balancing tool. This introduces challenges in extracting the application information to determine the partitioning, design of data structures as well as partition routines that allow for as much improvement in performance as possible. The thrust of the future work would be in two directions: a). design and implementation of techniques that are based more on garnering information from the application rather than being more heuristic based This would be based towards the completion of the automated partitioning and load-balancing tool.

b) an optimized partition routine for the composite grid distribution technique in particular. . What this means is that the partition routine could be implemented in a way such that it could preserve locality as much as possible in order to further minimize communication overheads. This would be based on the optimization of this specific scheme within the automated partitioning and load-balancing tool.

References

[1] BATSRUS Software Package, http://hpcc.engine.umich.edu/HPCC/codes\\/2/ BATSRUSv2.html

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

[3] M. J.Berger, P.Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82:64-84, 1989.

[4] M. J. Berger. Data Structures for Adaptive Mesh Refinement. In Ivo Babuska, Jagdish Chandra, and J.E.Flaherty, editors, *Adaptive Computational Methods for Partial Differential Equations*, pp 237-251, 1983

[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, March 1994

6] T. Cormen C.Liersorson, R. Livest. *Data Structures and Algorithms.* The MIT Press, McGraw Hill Book Company, 1991

[7] M. J. Berger. *Adaptive Mesh Refinement for Time Dependant Partial Differential Equations*. PhD thesis, Stanford University, Technical Report No. STAN-CS-82-924,1982.

[8] Henry J. Neeman. *Autonomous Hierarchical Adaptive Mesh Refinement for MultiScale Simulations*. PhD thesis, University of Illinois at Urbana-Champaign, March 1996, Technical Report No. 035, July1996.

[9] R.Das, D.J.Mavriplis, J.Saltz, S.Gupta, and R.Ponnusamy. *The Design and Implementation of a Parallel Unstructured Euler Solver using Software Primitives*. Technical Report.92-12, ICASE, Hampton, VA, March 1992

[10] GrACE–Grid Adaptive Computational Engine, www.caip.rutgers.edu/~parashar/TASSL

[11] J,E,Flaherty, R.M.Loy, C.Ozturan, M.S.Shepard, B.K.Szymanski, J.D.Teresco and L.H.Ziantz. Distributed Octree Data Structures and Local refinement method for theParallel Solution of 3-D Conservation Laws. In *Grid Generation and Adaptive Algorithms*, in series *The IMA volumes in Mathematics and its Applications*, pp 113-134, 1999 Springer.

[12] J,E,Flaherty, R.M.Loy, C.Ozturan, M.S.Shepard, B.K.Szymanski, J.D.Teresco and L.H.Ziantz. Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation. *Association for Computing Machinery*, 26:241-263, 1998

[13] J,E,Flaherty, R.M.Loy, C.Ozturan, M.S.Shepard, B.K.Szymanski, J.D.Teresco and L.H.Ziantz. Load Balancing and Communication Optimization for Parallel Adaptive Finite Element Methods. In *Proceedings of the XVII Int. Conf. Chilean Computer Science Society*, pp 246-255, 1997, Loa Alamitos CA.

[14] J,E,Flaherty, R.M.Loy, C.Ozturan, M.S.Shepard, B.K.Szymanski, J.D.Teresco and L.H.Ziantz. Predictive Load Balancing for Parallel Adaptive Finite Element Computation. In *International Conference on Parallel and Distributed Processing Techniques and Applications '97*, 1:460-469, 1997

[15] G.C. Fox. A Review of Automatic Load-Balancing and Decomposition methods for the Hypercube. In M Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pp.63-76, Springer-Verlag, New York, 1988.

[16] G.C. Fox, M. Johnson, D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall 1988

[17] S. R. Kohn and Scott. B. Baden. A Parallel Software Infrastructure for Structured Adaptive Mesh Methods. In *Proceedings of Supercomputing '95*, San Diego CA, 1995

[18] S. R. Kohn and Scott. B. Baden. Software Abstractions and Computational Issues in Parallel Structured Adaptive Mesh Methods for Electronic Structure Calculations. In *Proceedings workshop on Structured Adaptive Mesh Refinement Grid Methods*

[19] Shahid H Bokhari and M.Berger. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, C-36(5):570-580, May 1987

[20] PARAMESH Software Package, http://sdcd.gsfc.nasa.gov/ESS/easydir/inhouse/nacneice/paramesh/paramesh.html

[21] M. Parashar & J. C. Browne. On Partitioning Dynamic Adaptive Grid Hierarchies. In *Proceedings of 29th Annual Hawaii International Conference on System Sciences*, pp 604-613, Jan 1996.

[22] M. Parashar, J. C. Browne, "Distributed Dynamic Data-Structures for Parallel Adaptive Mesh Refinement. In *Proceedings of the International Conference for High Performance Computing*, pp 22-27, Dec 1995.

[23] M. Parashar & J.C. Browne. Object Oriented Abstractions for Parallel Adaptive Mesh Refinement. In *Proceedings of Parallel and Object Oriented Methods and Applications*, pp 203-207, Feb 1996.

[24] M. Parashar & J. C. Browne. *A Data Management Infrastructure for Parallel Adaptive Mesh-Refinement Techniques*. Technical Report 2.400, 1995 Department of Computer Sciences, University of Texas at Austin, TICAM, Texas.

[25] M.C.Rivaria. Design and Data Structure of fully Adaptive, multigrid, finite Element Software . In *ACM Transactions on Mathematical Software*, 19:242-264, 1984

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

[27] H.Samet. *The Design and Analysis of Spatial Data Structures,* Addison-Wesley Publishing Company, 1989

[28] SAMRAI, *A Parallel Software Infrastructure for Structured Adaptive Mesh Methods*, Scott R. Kohn and Scott B. Baden. Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114

[29] SCOREC *Parallel Scientific Computation Software Package:* www.scorec.rpi.edu/programs/parallel/ParallelScientific.html.

[30] S. Sengupta, C.P. Korobkin, "*C++ Object Oriented Data - Structures* ", Springer-Verlag Publishers, 1994

[31] M. Shee, S. Bhavsar, M. Parashar. An Application Centric Characterization of Distribution and Load Balancing Techniques for Dynamic Adaptive Grid Hierarchies. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)* , Las Vegas, Nevada, pp. 2530-2536, June 1999.

[32] M. Shee, S. Bhavsar, M. Parashar. A Characterization of Distribution Techniques for Dynamic Adaptive Grid Hierarchies. *Fifth U.S. National Congress on Computational Mechanics (USACM)* pp. 198-206, Boulder, CO, August 1999.

[33] L.Stalls. *Adaptive Multigrid in Parallel*. The School of Mathematical Sciences, Mathematical Research Report MRR95-95.027, Australian National University, Feb. 1995.

[34] A. Tenenbaum, Y.Langsam, M.J.Augenstein. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996.

Appendix A

Algorithm 4.1 : To determine the participation number of each node

```
int  participation_number(TREE_PTR)
{
        if(TREE_PTR!= NULL)
        {
                if(leftchild  of TREE_PTR ==NULL)
                {
                        increment participation number
                }
                else
                {
                        participation number = participation number of children( leftchild)
                }
        }
        return TREE_PTR
}
int participation_number_of_children(TREE_PTR)
{
        pno= participation_number(ptr2)
        while(right_child of TREE_PTR != NULL)
        {
                pno += participation_number(TREE_PTR (rightchild))
        }
        return pno
```

Algorithm 4.2 : To place the sub-nodes into their relevant units

```
BboxList&  path_left(TREE_PTR , node_no)
{
        if(TREE_PTR != NULL)
        {
                if(leftchild  of ==NULL)
                {
                        pnum of leftchild = 1
                        add(celList[path_no], TREE_PTR ->bb)
                        increment node_no
                }
                else left_child of TREE_PTR != NULL)
                {
                        add(celList[path_no], TREE_PTR ->bb)
                        increment node_no

                        path_right(TREE_PTR ->leftchild, node_no)
                }
        }
        return *celList
```

```
//-------------------------------------------------------
 void path_right (TREE_PTR, node_no)
{
        left_path(TREE_PTR,node_no);
        while(right_child of TREE_PTR != NULL)
        {
                increment path_no
                pno += path_left(rightchild of TREE_PTR, cellist)
                for(x=1,bb2=celList[path_no-1].first, x<node_no;x++,bb2=celList[path_no-
1].next())
                {
                        add(celList[path_no],*bb2);
                }
                left_path(rightchild of TREE_PTR,node_no)
                ptr2= rightchild of TREE_PTR
        }
```

Algorithms: 4.3 The detailed algorithm for the interface to the actual partitioning method is given below:

initial partition_scheme (bounding box list, processor, timestep, level, minblock, maxblock)

Note: the Bounding box list is the list at each level of the hierarchy for each timestep
//1. Check for Maxblock size
If box_work > Maxblock {

- *use the method partition_exact to break the box into half*
- *insert the new boxes in the Bounding Box List (BBL) at that place*
- *Remove the bounding box that was partitioned*

//2. Calculate the average work of the BBL
for(each box in the BBL)
{
calculate work
total work += work
}
average work = total work/num of processors
//3.Partition the work among the processors
for(each box in theBBL do)
{

- *find the box_work , and add it to processor work*
- *if (processor work < average work)*
  assign this bounding box to the current processor
  process the next box
  else if(processor work = average work)
  we are done with this processor, increment the processor count
  set the processor work to zero
  else if (processor work > average work)
  {
  find out the extra work and partition that current box under consideration by using partition_exact method to break thebox into 2 smaller boxes. We get 2 boxes
  - *one the size of extra work*
  - *one the size of (original size - extra work)*

  put them in the BBL, and remove the partitioned box
  add first new box (original size - extra work) to processor work.

This is the amount of work for the first processor
} // end if
recalculate the average
- *Repeat the process for the next processor, continue till all processors and all the boxes in the BBL are processed.*

} //end for

Algorithm 4.4: The detailed algorithm for the interface to the optimized partitioning method is given below: Please note that the optimization done is in bold text.

optimized partition_scheme (bounding box list, processor, timestep, level, minblock, maxblock)

Note1: the Bounding box list is the list at each level of the hierarchy for each timestep
Note2:The total work is already calculated at the time of reading in the boxes.
1. Average work = total work/number of processors
//2. Check for Maxblock size
- If box_work > Maxblock {
  - *6 divide the box by Maxblock size to obtain number of boxes to break it into*
  - ***7 use the method partition_all to break the box into the calculated number of boxes (from above)***
  - *8 insert the new boxes in the Bounding Box List (BBL) at that place*
  - *9 remove the box that was partitioned*

//3. Now partition the work among the processors
for(each box in theBBL do)
{
10 find the box_work , and add it to processor work until it exceeds the average work.
11 if (processor work < average work)
assign this bounding box to the current processor
process the next box
else if(processor work = average work)
assign this box to the current processor
we are done with this processor, increment the processor count
set the processor work to zero
else if (processor work > average work)
{
delete the work of this bbox from processor work
extra work = average work – processor work
if (average work is less than Minblock and extra work is < MinBlock)
{
make average work = Minblock size
}
find out the extra work and partition that current box under consideration by using partition_exact method to break the box into 2 smaller boxes. We get 2 boxes
11.5.2one the size of extra work
11.5.3one the size of (original size - extra work)
11.6put them in the BBL, and remove the partitioned box
11.7add first new box (original size - extra work) to processor work.
11.8This is the amount of work for the first processor
Else if(only extra work is < MinBlock)
{
recalculate average
move on to the next processor

```
                }
                else //extra work is > MinBlock
                {
                        parse the bblist starting for the current bbox and find a box either equal to or
                within a 10% tolerance of extra work.
                        If (found)
                        {
                                assign to current processor
                                recalculate average
                                go back to the original bbox and start the process over again
                        }
                        else
                        {
                                go back to the original bbox and start the process over again
                                break the bbox
                        }
                }
        } // end if
        //Repeat the process for the next processor, continue till all processors and all the boxes in the
        BBL are processed.
} //end for
```

Algorith 4.5 : Measurement of Intra-level communication

```
At each timestep
                for(level =0 to level=max_levels)
                  for (proc=0 to proc=num_proc-1)
                    //find out the intersection of each bounding box (bb) at this timestep, for this
                    processor and level, with all the bbs that belong to all other processors at the same
                    level. If there is an intersection between two such bbs, then there indeed is intralevel
                    communication else there is not intersection for this bb.
                    (for each bb in the bblist[timestep][processor][level] and each number of
                    interactions in all the directions)
                        find the interaction  bb, which is given by the Ibbox routine of
        DAGHGhostinteraction
                        Intersect this with all the bbs for all the processors at the same level as above
                        to obtain the intersection bb.
                        if (intersection bb is != null)
                        {
                                intraclevel comunication += intersection_bb.size( )
                        } }} }
```