SYSTEM SENSITIVE RUNTIME MANAGEMENT OF DYNAMIC APPLICATIONS

BY SHWETA SINHA

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

____

____

____

New Brunswick, New Jersey

May, 2001

ABSTRACT OF THE THESIS

System Sensitive Runtime Management of Dynamic Applications

by Shweta Sinha

Thesis Director: Professor Manish Parashar

In this thesis we investigate the design and evaluation of a system sensitive partitioning and load balancing framework for distributed adaptive grid hierarchies that underlie parallel adaptive mesh-refinement (AMR) techniques for the solution of partial-differential equations. The framework uses system capabilities and current system state to select the appropriate distribution parameters (e.g., partitioning granularity, load per processor) to maximize overall application performance. System state is dynamically monitored, and is used to compute the relative abilities of the computing nodes to do work. These capacities are then used for partitioning and load balancing.

Acknowledgements

I would like to thank my advisor Dr. Manish Parashar for his guidance, encouragement and patience during the work on my thesis. I would like to thank Dr. David Daut and Dr. Ivan Marsic for being members of my thesis committee. I would like to thank my colleagues at TASSL (The Applied Software Systems Lab) for their help and also for creating a nice and friendly work environment. I would also like to thank my family and friends for their constant love and support. Finally, I would like to thank my husband Rajnish for his unfailing encouragement and support throughout the course of my studies at Rutgers.

Dedication

To my parents Col. and Mrs. N. Srivastava

and

my husband Rajnish

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There exist large scientific applications whose solutions require a significant amount of computing resources. Consequently, these applications are distributed over multiple nodes and solved in parallel as shown in Figure 1.1. This requires a method to solve the application and a toolkit to implement it in parallel. In this thesis, the technique that has been used to solve the application is called adaptive mesh refinement (AMR) and the corresponding toolkit to implement it in parallel is called Grid Adaptive Computational Engine (GrACE). The distribution of applications on computing nodes is extremely challenging and much research has been done to address this issue. However, in networked environments which are non-dedicated (or shared), accounting for differences in resource availabilities of the computing nodes is important and is addressed in this thesis.

AMR techniques for the solution of partial differential equations (PDEs) 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. AMR techniques are 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 change is dynamic. Distributed implementations of these adaptive methods offer the potential for the accurate solution of realistic models of important physical phenomena. These implementations, however, lead to interesting challenges in dynamic resource allocation, data-distribution and load balancing, communications and coordination, and resource management.

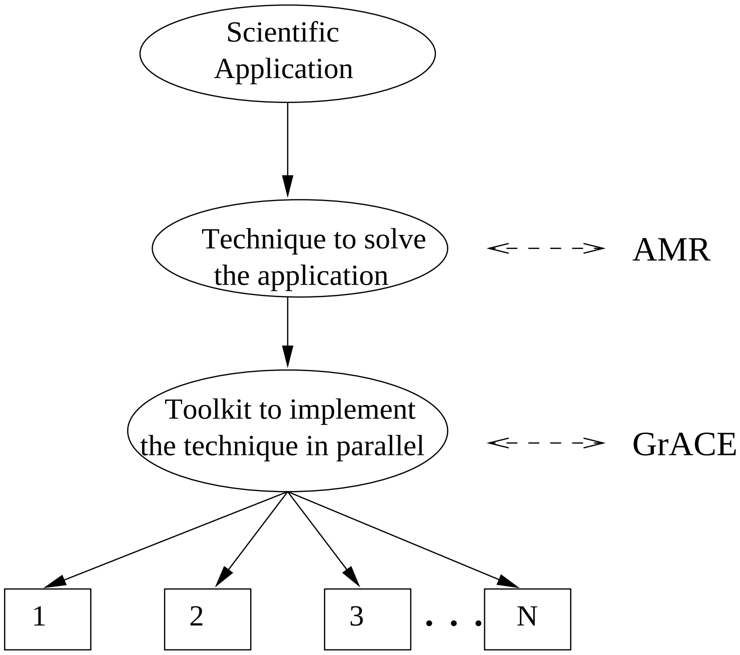

Figure 1.1: *Parallel implementation of solutions to complex scientific applications over N nodes.*

Moving these applications to dynamic and heterogeneous networked computing environments introduces a new level of complexity. These environments require the selection and configuration of application components based on the availability and state of the system resources. However, the complexity and heterogeneity of the environment make selection of the "best" match between system resources, mappings and load distributions, communication mechanisms, etc., non-trivial. System dynamics coupled with application adaptivity makes application and run-time management a significant challenge.

1.1 Overview of the Thesis

In this thesis, we present the design and evaluation of an adaptive system-sensitive partitioning and load balancing framework for distributed AMR applications on heterogeneous and dynamic cluster environments. The framework uses system capabilities and current system state to select and tune appropriate distribution parameters. The system state is the state of resources such as CPU, memory and bandwidth. Current system state is obtained using a resource monitoring tool. The current system state along with system capabilities are then used to compute relative computational capacities of each of the computational nodes in the cluster. These relative capacities are used by a heterogeneous "system-sensitive" partitioner for dynamic distribution and load-balancing. The heterogeneous partitioner has been integrated into the GrACE (Grid Adaptive Computational Engine) infrastructure [1, 2] adaptive runtime system. GrACE is a data-management framework for parallel/distributed AMR, and is being used to provide AMR support for varied applications including reservoir simulations, computational fluid dynamics, seismic modeling and numerical relativity.

1.2 Contributions of the Thesis

This thesis makes the following contributions:

1. The main contribution of this thesis is the design and implementation of a system-sensitive partitioning and load balancing framework. This framework takes the current system state of the processors into account before distributing the work load among them. The system sensitive partitioning/load-balancing framework presented in this thesis addresses both, dynamic heterogeneous environments as well as dynamically adaptive applications.

2. This thesis introduces a linear cost model to compute the relative capacities of the processors which are used by the system-sensitive partitioner to distribute work load among the processors. The relative capacity of a processor is computed by taking into account various parameters such as its CPU usage, memory availability and link bandwidth.

3. In order to create a heterogeneous environment for simulations, a synthetic load generator was implemented to artificially load the processors. The artificial load generator is deterministic in nature and it consumes memory and CPU in a linearly increasing manner until it reaches a desired load level.

4. This thesis also integrated the NWS resource monitoring tool with GrACE. This tool, among other things, monitors the fraction of CPU time available for new processes, the fraction of CPU available to a process that is already running, end-to-end TCP network bandwidth, and free memory, all of which are used in our framework to compute the relative capacities of the processors.

5. The system-sensitive partitioning framework developed in this thesis also allows the partitioning algorithm to dynamically adapt to changes in the environment. This is achieved through periodic sampling of the system state of the processors through NWS followed by updates of their relative capacities. The relative capacities are then used by the partitioning algorithm.

1.3 Outline of the Thesis

This thesis comprises five chapters.

Chapter 1 consists of the introduction and presents the outline and contributions of this thesis. Chapter 2 presents the related work and the background of the thesis. The adaptive grid structure defined by hierarchical AMR techniques is described here along with an overview of GrACE [1, 2]. Chapter 3 introduces the architecture of the system-sensitive framework and also describes the cost model used in the framework for relative capacity calculation and assignment. Chapter 4 presents the simulation setup and an experimental evaluation of the system system partitioning routine. Chapter 5 presents the conclusions of this study and directions for future work.

Chapter 2

Background and Related Work

2.1 Background Work

2.1.1 Problem Description: Distributed AMR Applications

Dynamically adaptive numerical techniques for solving differential equations provide a means for concentrating computational effort to appropriate regions in the computational domain. In the case of hierarchical AMR methods, this is achieved by tracking regions in the domain that require additional resolution and dynamically overlaying finer grids over these regions. AMR-based techniques start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are tagged and finer grids are overlaid on the tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the AMR formulation by Berger & Oliger [4] is shown in Figure 2.1. Operations on the hierarchy defined by this algorithm are outlined below:

Time Integration: Time integration is the update operation performed on each grid at each level of the adaptive grid hierarchy. Integration uses an application specific difference operator.

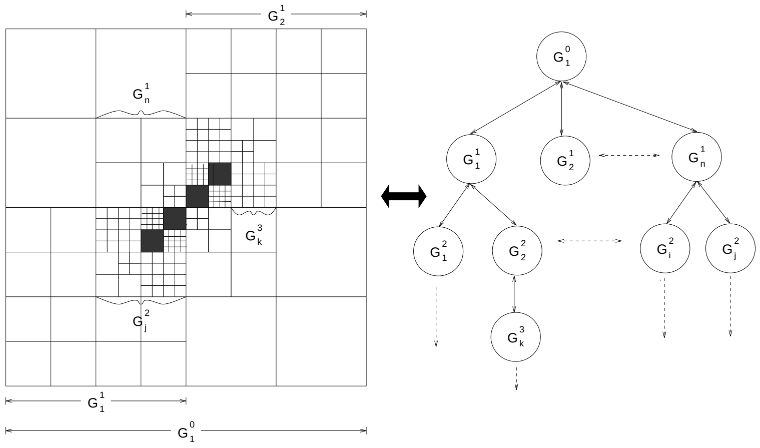

Figure 2.1: *Adaptive grid hierarchy - 2D (Berger-Oliger AMR scheme)*

Inter-Grid Operations: Inter-grid operations are used to communicate solutions values along the adaptive grid hierarchy. Two primary inter-grid operations are *Prolongation* operations defined from a coarser grid to a finer grid and *Restriction* operations defined from a finer grid to a coarser grid.

Regriding: The regriding operation consists of three steps: (1) flagging regions needing refinement based on an application specific error criterion, (2) clustering flagged points, and (3) generating the refined grid(s). The regriding operation can result in the creation of a new level of refinement or additional grids at existing levels, and/or the deletion of existing grids.

2.1.2 Decomposing the Adaptive Grid Hierarchy

Key requirements of a decomposition scheme used to partition the adaptive grid hierarchy across processors are: (1) expose available data-parallelism; (2) minimize communication overheads by maintaining inter-level and intra-level locality; (3) balance overall load distribution; and (4) enable dynamic load re-distribution with minimum overheads.

A balanced load distribution and efficient re-distribution is particularly critical for parallel AMR-based applications as different levels of the grid hierarchy have different computational loads. In case of the Berger-Oliger AMR scheme for time-dependent applications, space-time refinement result in refined grids which not only have a larger number of grid elements but are also updated more frequently (i.e. take smaller time steps). The coarser grid are generally more extensive and hence its computational load cannot be ignored. Furthermore, the AMR grid hierarchy is a dynamic structure and changes as the solution progresses, thereby making efficient dynamic re-distribution critical.

2.1.3 Grid Adaptive Computational Engine (GrACE)

The "system-sensitive" runtime management mechanisms presented in this thesis has been integrated into the GrACE [1, 2] distributed and dynamic data-management infrastructure. GrACE is an object-oriented toolkit for the development of parallel and distributed applications based on a family of adaptive mesh-refinement and multigrid techniques. It provides a high-level programming abstraction and allows users to simply and directly express distributed computations on adaptive grids, and is built on a distributed dynamic data-management substrate implementing "semantically specialized" distributed shared memory.

GrACE Distributed Dynamic Data Management Substrate

The primary objective of the GrACE substrate is to provide distributed and dynamic data-management support for the development, implementation and execution of large-scale parallel adaptive applications. The lowest layer of the infrastructure implements a Hierarchical Distributed Dynamic Array (HDDA). The HDDA provides array semantics to hierarchical and physically distributed data. HDDA objects encapsulate dynamic load-balancing, interactions and communications, and consistency management. The next layer adds application semantics to HDDA objects to implement application objects such as grids, meshes and trees. This layer provides an object-oriented programming interface for directly expressing multi-scale, multi-resolution AMR computations. The upper layers of the infrastructure provide abstractions, components and modules for method-specific computations.

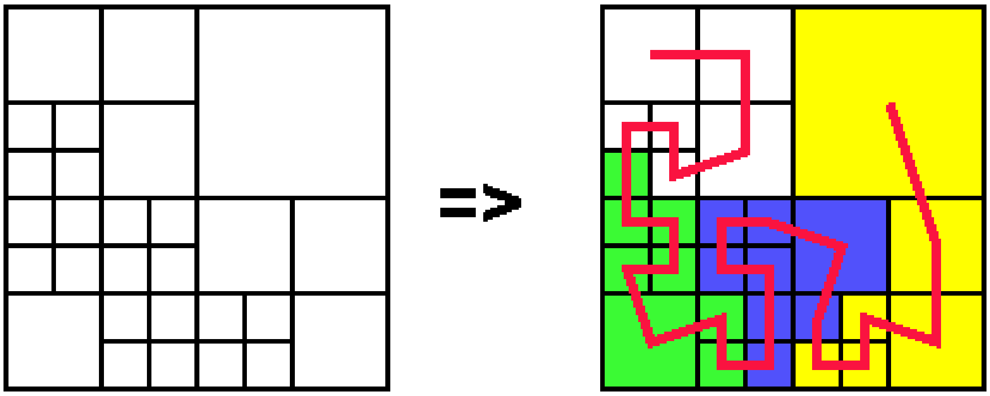

Figure 2.2: *Hierarchical space-filling mappings.*

Hierarchical Distributed Dynamic Array

The primitive data structure provided by the data-management infrastructure is an array which is hierarchical in that each element of the array can recursively be an array, and dynamic in that the array can grow and shrink at run-time. The array of objects is partitioned and distributed across multiple address spaces with communication, synchronization and consistency transparently managed for the user. The lowest level of the array hierarchy is an object of arbitrary size and structure. The primary motivation for defining such a generalized array data-structure is that most application domain algorithms are formulated as operations on grids and their implementation is defined as operations on arrays. Such array based formulations have proven to be simple, intuitive and efficient, and are extensively optimized by current compilers. Providing an array interface to the dynamic data-structures allows implementations of new parallel and adaptive algorithms to reuse existing kernels at each level of the HDDA hierarchy. Like conventional arrays, HDDA must translate index locality (corresponding spatial application locality) to storage locality, and must maintain this locality despite its dynamics and distribution. It is decomposed into two components: hierarchical index spaces, and distributed dynamic storage and access. These components are described below.

Hierarchical Index Spaces

The hierarchical extendible index space component of the HDDA is derived directly from the application domain using space-filling mappings, which are computationally efficient, recursive mappings from N-dimensional space to 1-dimensional space. An example is shown in Figure 2.2. The solution space is first partitioned into segments. The space filling curve then passes through the midpoints of these segments. The mapping functions are computationally efficient and consist of logical bit-interleaving operation on the N-dimensional coordinates. Space filling mappings encode application domain locality and maintain this locality through expansion and contraction. 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 the hierarchy. The hierarchical index-space is used by the HDDA as the basis for application domain partitioning, as a global address space for allocating storage to application objects, as a global name-space for name resolution, and for communication scheduling.

Distributed Dynamic Storage and Access

Data storage is implemented using extendible hashing techniques with contractions of the hierarchical index-space indices serving as hash keys. Extendible hashing provides efficient management mechanisms for dynamic data-bases. Spans of the hash keys space are mapped to units of storage called hash buckets and expansion and contraction of the key space are handled efficiently by splitting and merging these buckets. These operations are local involving at most two buckets. As spans of the index space are mapped to contiguous storage within a bucket by the hashing scheme, index locality (which encodes applications domain locality) is translated into storage locality. Data locality is preserved without copying. Partitioning of the applications domain and associated distribution of HDDA objects is achieved by partitioning the index space among processing elements and assigning ownership to HDDA buckets. Buckets are used as the units of communications and caching. The overall HDDA distributed dynamic storage scheme is shown in Figure 2.3.

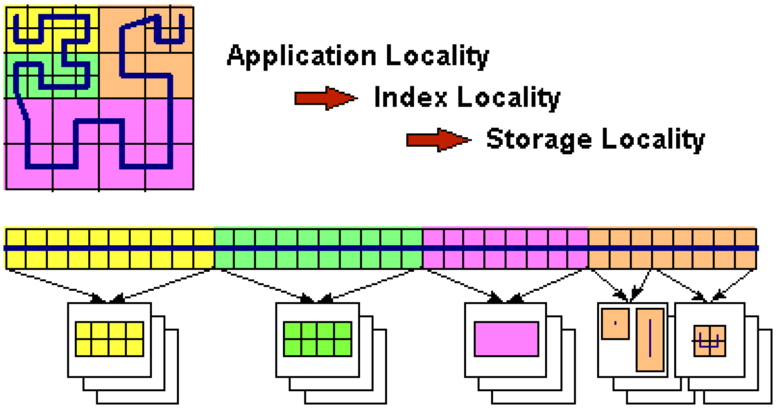

Figure 2.3: *Distributed storage of dynamic objects.*

2.2 Related Work

A taxonomy of related work in partitioning/load-balancing techniques is shown in Figure 2.4. Techniques are classified based on whether they address system and/or application dynamics. We define a dynamic application as one which has time varying computational requirements and a static application as one which does not. The four quadrants shown in Figure 2.4 are discussed below. Note that in this discussion we focus on techniques that address system heterogeneity.

2.2.1 Static Application and Static Environment

There exists a large body of research addressing load balancing techniques where the system as well as the application are static. Most of these techniques, however, do not address system heterogeneity. One of the most recent works that addresses this issue is by Figueira et. al in [5]. They have defined an algorithm which determines the best allocation of tasks to resources in heterogeneous systems. They have discussed the mapping of parallel applications with multiple tasks to heterogeneous platforms. Their work however does not take into account the dynamism in the environment. Furthermore, it addresses static applications. Also, M. Ripeanu in [9] has developed a measure of efficiency that considers both the relative computational power of each processor and the share of time used on each processor. The relative speed is measured

Static Application , Static System State [5]	Static Application , Dynamic System State [3, 6, 7, 9, 13, 14, 15]
Dynamic Application, Static System State [8, 17, 18, 19]	Dynamic Application, Dynamic System State [20]

Figure 2.4: *Taxonomy of related work on load balancing based on system and application state. The references are shown in brackets above.*

using synthetic benchmarks or by using the real application itself as a benchmark.

2.2.2 Static Application and Dynamic Environment

Techniques described in [6], [9], and [14] address dynamic heterogeneous environments. These techniques primarily differ in the way they quantify system capabilities in the heterogeneous system. Watts et. al in [6] have presented different techniques to measure the resource capacities. One method uses a benchmarking program to determine the relative speeds of various machines. These offline performance metrics, along with other specifications, such as the machines' memory capacities, can be placed in a file which is read at the beginning of the computation. Another method to measure the resource capacities' is the use of system-measured utilization numbers. Cai et. al in [14] have simulated a heterogeneous environment based only on CPU processing power for their study. Gross et. al in [15] have also emphasized the need for external measurements of network mechanisms to support effective adaptive execution of large distributed scientific applications.

Various researchers have proposed "matching and scheduling" algorithms for a dynamic heterogeneous environment. For example, Topcuoglu, et al in [22] have presented two algorithms, the Heterogeneous Earliest-Finish-time and the Critical-Path-on-a -processor for scheduling DAGs on a bounded number of heterogeneous processors. Maheswaran et. al in [7] have suggested a new dynamic algorithm called the "hybrid-remapper " which uses the run-time values that become available upon completion of subtasks as well as machine availabilities during application execution time. Leinberger et. al in [16] have addressed a workload distribution problem for a computational grid with multi-resource servers. Servers in the grid have multiple resource capacities and the applications submitted by the users have multiple resource requirements. To fully utilize all K resources at each site, they suggest a heuristic to *match* the job mix at each server with the capabilities of that server, in addition to balancing the load across servers. Andersen et. al in [13] have also studied dynamic load-balancing for a static application. However, their approach to redistribution is through initiation of requests to a target machine from either idle machines asking for work or from busy machines desiring to relinquish excess work.

2.2.3 Dynamic Application and Static Environment

Much research has been done that addresses dynamic applications and static environments, however, they do not take heterogeneity of the system into account. M. Parashar et. al in [2] have addressed the partitioning of dynamic applications on a static environment. They have not addressed the heterogeneity of the system. References [8], [17], and [18] have also looked at the partitioning of dynamic applications, but they have also not taken system heterogeneity or dynamic system state into account. G. Karypis, et. al in [19] have implemented algorithms for partitioning unstructured graphs and for computing fill-reducing orderings of sparse matrices. B. Hendrickson, et. al in [21] have developed a library called Zoltan that includes a suite of dynamic load-balancing algorithms, including both geometric and graph-based algorithms.

2.2.4 Dynamic Application and Dynamic Environment

Most existing partitioning and load balancing research addresses either dynamic applications or dynamic system environments, and fall in one of the quadrants discussed above. The system-sensitive partitioning/load-balancing framework presented in this thesis addresses both *dynamic heterogeneous* environments and *dynamic* adaptive applications. Furthermore, it introduces a cost model that defines the relative capacities of system resources based on current system parameters such as CPU availabilities, memory usage and link bandwidth. System parameters are obtained using a resource monitoring tool.

Chapter 3

System Architecture

3.1 System-Sensitive Runtime Management Architecture

This section outlines the design of the system-sensitive runtime management framework. A block diagram of the adaptive, system sensitive runtime partitioning framework is shown in Figure 3.1. The framework consists of three key components - the system state monitoring tool, the capacity calculator, and the system-sensitive partitioner. In this framework, first, the state of resources associated with the different computing nodes in the cluster is monitored. This information is then used to compute the relative computational capacities of the nodes which in turn are used by the heterogeneous system-sensitive partitioner for dynamic distribution and load-balancing. The different components are described below.

3.1.1 Resource Monitoring Tool

System characteristics and state are determined at run-time using a resource monitoring tool. The resource monitoring tool gathers information regarding the CPU availability, memory usage and link-capacity of each processor. This information is then passed to the Capacity Calculator as shown in Figure 3.1 and discussed below.

3.1.2 Capacity Metric

Using system information obtained with the resource monitoring tool, a relative capacity metric is computed for each processor using a linear model. Let us assume that

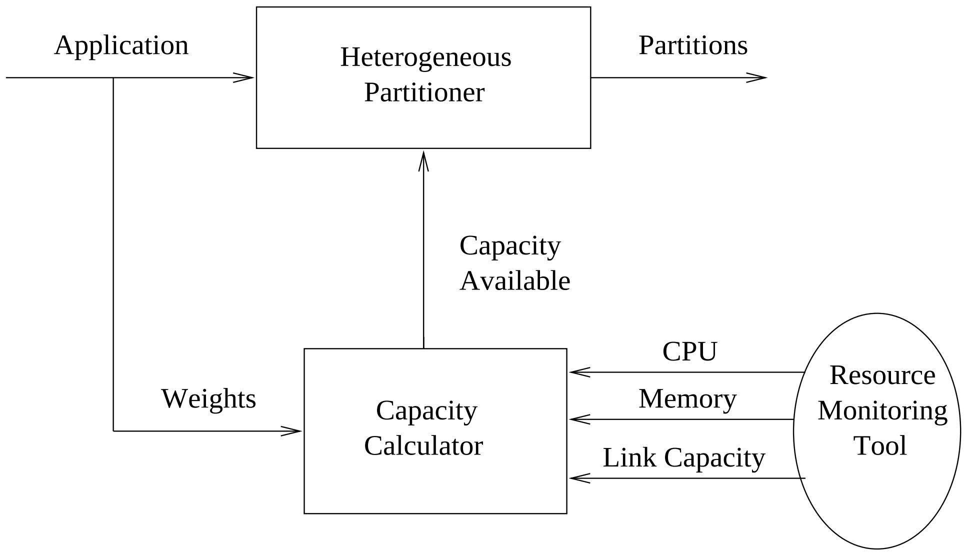

Figure 3.1: *Block diagram of the system model*

there are K processors in the system among which the partitioner distributes the work load. For node k, let $\mathcal{P}_k$ be the percentage of CPU available , $\mathcal{M}_k$ the available memory, and $\mathcal{B}_k$ the link bandwidth, as provided by NWS. The available resource at k is first converted to a fraction of total available resources, as

$$P_k = \frac{\mathcal{P}_k}{\sum_{i=1}^{K} \mathcal{P}_i} \tag{3.1}$$

$$M_k = \frac{\mathcal{M}_k}{\sum_{i=1}^{K} \mathcal{M}_i} \tag{3.2}$$

$$B_k = \frac{\mathcal{B}_k}{\sum_{i=1}^{K} \mathcal{B}_i} \quad . \tag{3.3}$$

The relative capacity C_k of processor k is then defined as the weighted sum of these normalized quantities as

$$C_k = w_p P_k + w_m M_k + w_b B_k \tag{3.4}$$

where w_p, w_m, and w_b are the weights associated with the relative CPU, memory, and link bandwidth availabilities, respectively, such that $w_p + w_m + w_b = 1$. The weights are application dependent and reflect its computational, memory, and communication requirements. Note that

$$\sum_{k=1}^{K} C_k = 1. \tag{3.5}$$

If the total work to be assigned to all the processors is denoted by L, then the work L_k that can be assigned to the kth processor is calculated as $L_k = C_k L$.

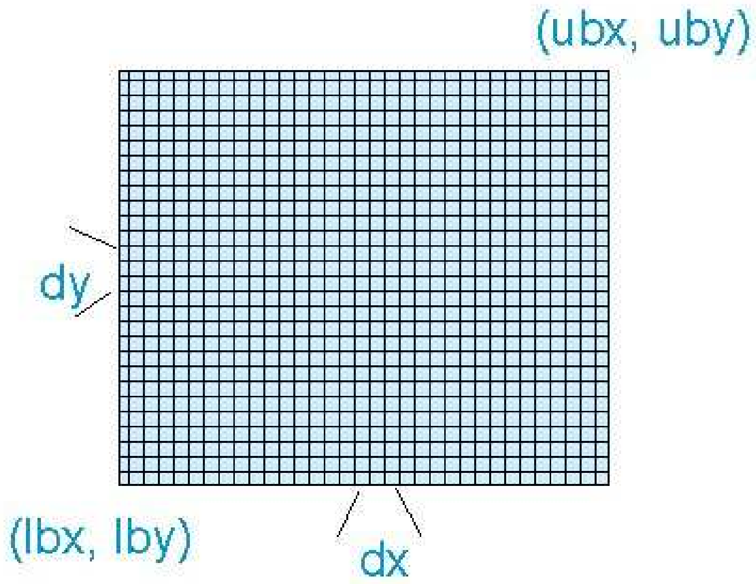

Figure 3.2: *Diagram of a 2-D bounding box*

A non-linear model for the calculation of relative capacities may be more accurate but the complexity and overheads associated with it are non-trivial.

3.1.3 ACEHeterogeneous: The System-Sensitive Partitioner

The system-sensitive heterogeneous partitioner, ACEHeterogeneous, has been integrated into the GrACE infrastructure and provides load-balancing support for AMR applications. In GrACE, the grids in the adaptive grid hierarchy are maintained in the form of a bounding box list. An example of a 2-D bounding box is shown in Figure 3.2. The bounding box is an abstraction which addresses a region in the computational domain. At every time-step of the simulation, the partitioning routine gets a list of bounding boxes as input from GrACE. The partitioning routine then partitions this bounding box list. A high level description of the partitioning process is shown in the flowchart in Figure 3.4 and is comprised of the following steps:

- The relative capacities $C_k, k = 1, \ldots, K$ of the K processors on which the application will run are obtained from the Capacity Calculator as shown in Figure 3.1.

- The total work L associated with the bounding box list is calculated.

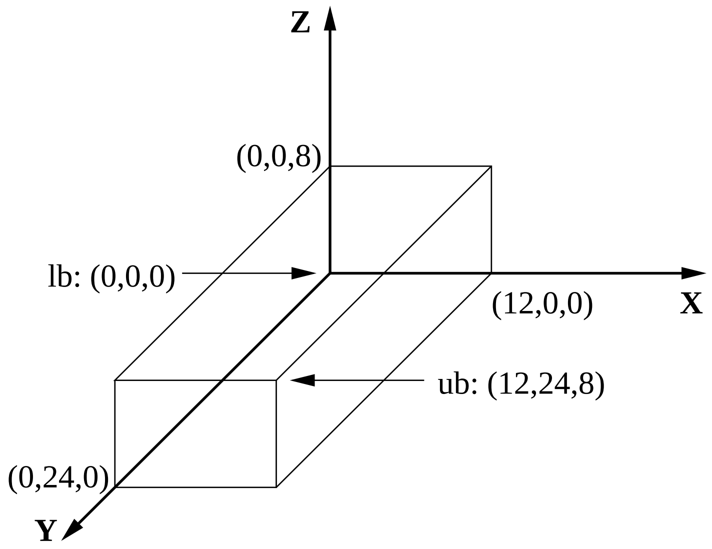

Figure 3.3: *Diagram of a 3-D bounding box. The lower and upper bounds are denoted by lb and ub, respectively.*

- Using the capacity metric, the ideal work load L_k that can be assigned to the kth processor is calculated.

- The bounding boxes are then assigned to the processors, with the kth processor receiving a total work load of W_k which is approximately equal to L_k.

- At any point, if the work associated with a bounding box exceeds the work the processor can perform, then it is broken into two in such a way that the work associated with at least one of the two broken boxes is less than or equal to the work the processor can perform.

- Both the list of bounding boxes as well as the relative capacities of the processors are sorted in an increasing order, with the smallest box being assigned to the processor with the smallest relative capacity. This eliminates unnecessary breaking of boxes, and hence, reduces the associated overhead.

- The local output list of bounding boxes is returned to GrACE which then allocates the work to the particular processor.

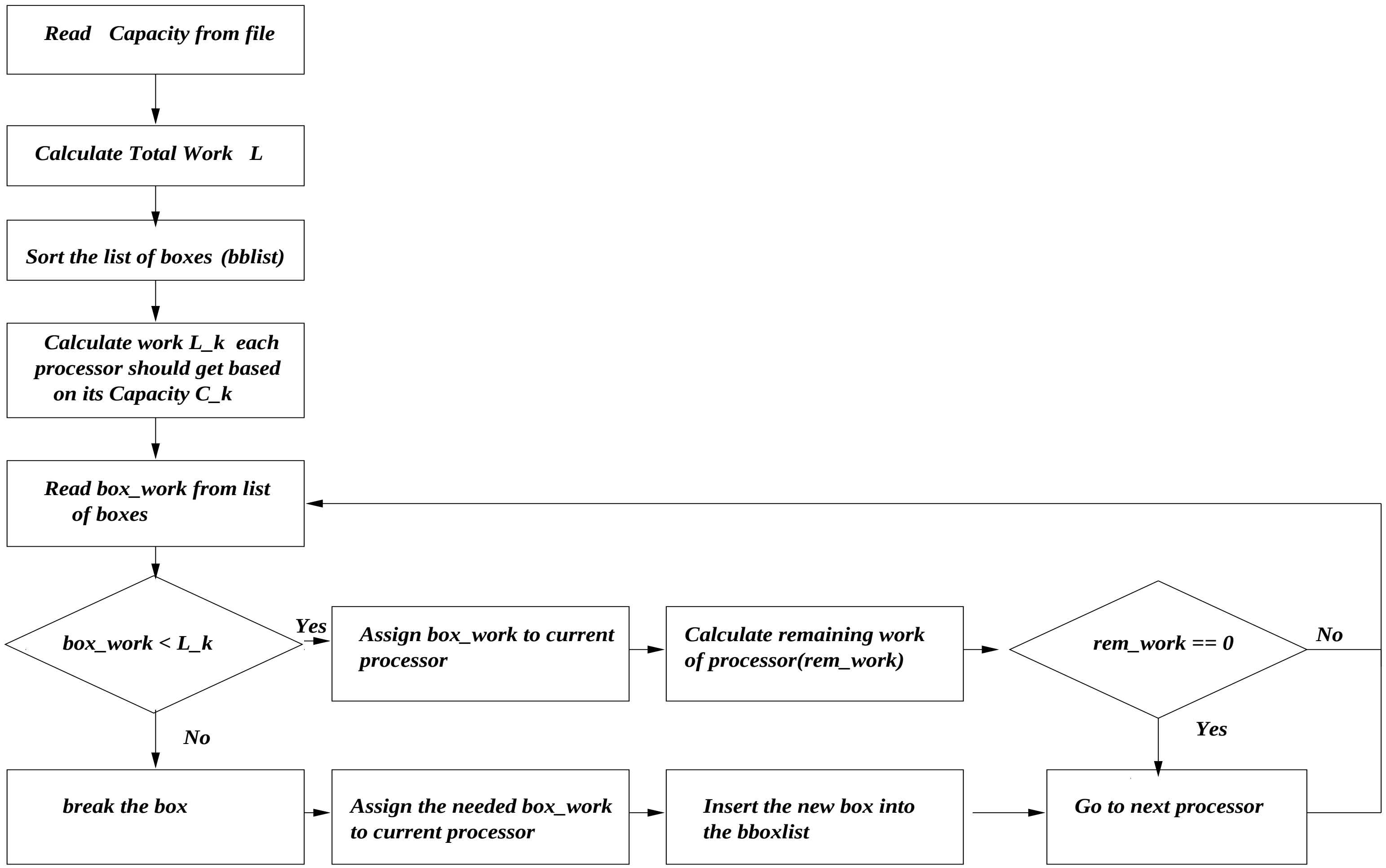

Figure 3.4: *Flow chart of the partitioning process.*

- Since GrACE has a limit on the minimum size of a box, the bounding boxes have to be broken under this constraint. As a consequence, at times, the total work load W_k that is assigned to processor k may differ from L_k thus leading to a slight load imbalance.

- The *aspect ratio* of a bounding box, defined as the ratio of the longest side to the shortest side, is also taken into account while breaking the box. We consider only the longest axis of the bounding box and break along that direction. For example, consider a box with the coordinates of the lower and upper bounds as (0,0,0) and (12,24,8) respectively, as shown in Figure 3.3. In this figure, since the length of the bounding box is longest in the y direction, we break the box along the xz plane. Breaking the box along any other axis would result in bounding boxes of very uneven sizes [8].

Figure 3.5 shows the system sequence diagram.

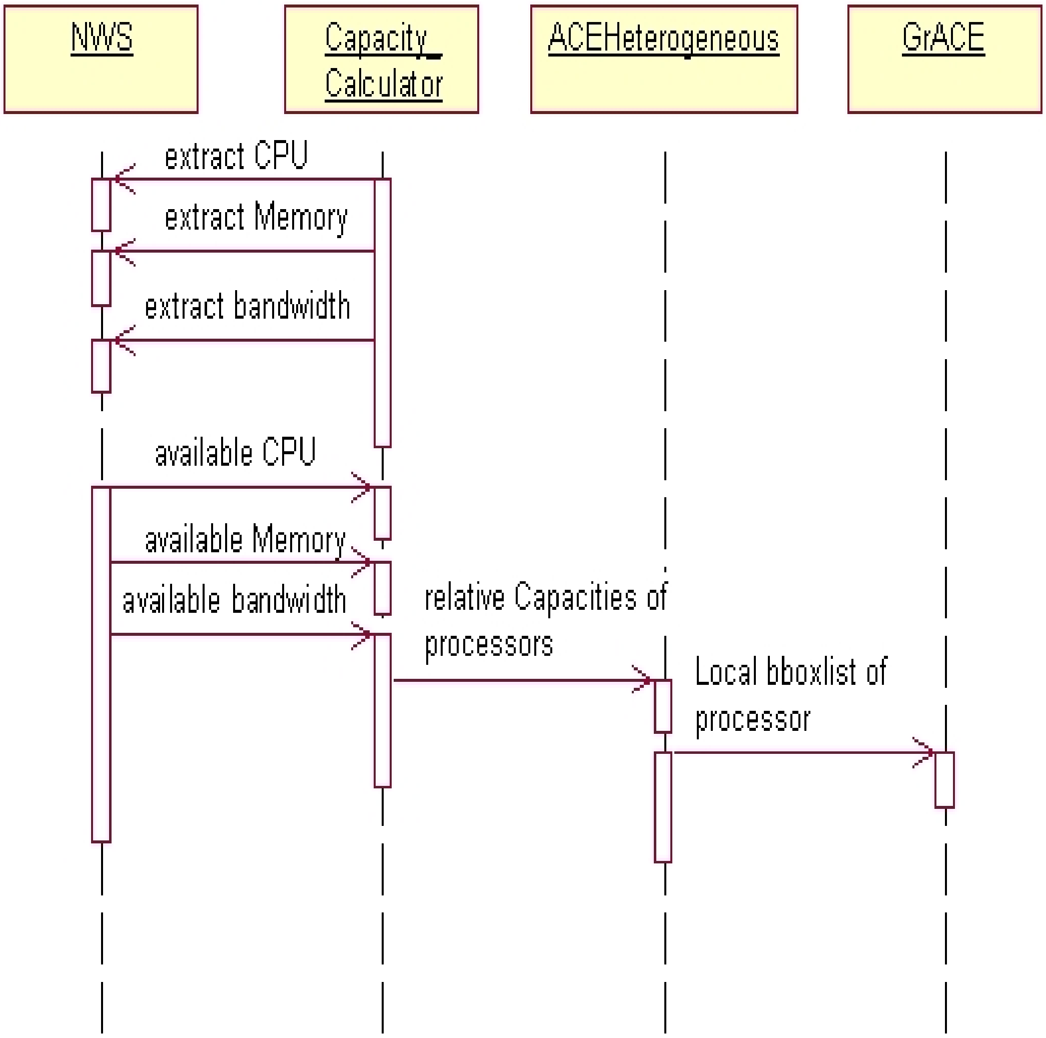

Figure 3.5: *System sequence diagram.*

Chapter 4

Experimental Setup and Simulation Results

4.1 Experimental Setup

A 3-D compressible turbulence kernel executing on a Linux based workstation cluster is used to experimentally evaluate the adaptive, system-sensitive partitioning framework. The kernel solved the Richtmyer-Meshkov instability, and used 3 levels of factor 2 refinement on a base mesh of size 128x32x32. The cluster consisted of 32 nodes interconnected by fast ethernet (100MB). The experimental setup consisted of a synthetic load generator and an external resource monitoring system. The evaluation consisted of comparing the runtimes and load-balance generated for the system-sensitive partitioner (ACEHeterogeneous) with those for the default space-filling curve based partitioning scheme (ACEComposite) provided by GrACE. This latter scheme performs an equal distribution of the workload on the processors. The adaptivity of the system-sensitive partitioner to system dynamics, and the overheads of sensing system state were also evaluated.

4.1.1 Synthetic Load Generation

In order to compare the two partitioning schemes, the experiment needed to be performed in a controlled environment such that the dynamics of the system state were the same in both cases. This was achieved using a synthetic load generator to load processors with artificial work. The load generator decreased the available memory and increased CPU load on a processor, thus lowering its capacity to do any work. As shown in Figure 4.1, the load generated on the processor increased linearly at a specified rate until it reached the desired load level. Note that multiple load generators were run on a processor to create interesting load dynamics.

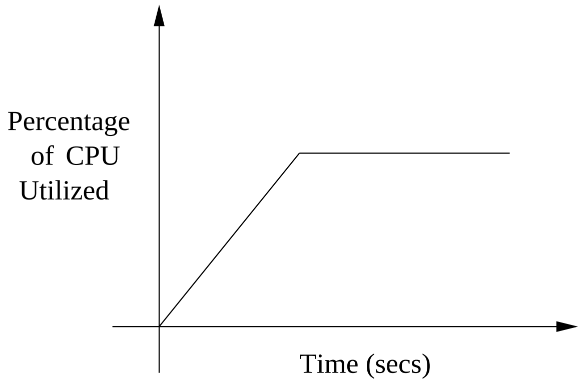

Figure 4.1: *Linear growth of the load generator*

4.1.2 Resource Monitoring

In our experimental setup, system characteristics and state are determined at run-time using the NWS resource monitoring tool which is explained at length in Appendix A and also in [3]. NWS is a distributed system that periodically monitors and dynamically forecasts the performance delivered by the various network and computational resources over a given time interval. NWS currently monitors the fraction of CPU time available for new processes, the fraction of CPU available to a process that is already running, end-to-end TCP network latency, end-to-end TCP network bandwidth, free memory, and the amount of space unused on a disk. In our setup, NWS daemon processes were run on each of the processors.

Overheads Associated with NWS

The memory, storage and CPU consumption of NWS are as follows [3]:

- CPU: less than 3% utilization – about what is measurable with either *vmstat* or *uptime*.
- memory: 3300K total vmsize, RSS 2500K on the average

- storage: 10 to 20 megabytes for code and data storage (depending on how much data will be captured, but can be tuned as needed)
- network: the system will conduct a series of performance experiments every K seconds, where K is a tunable parameter. The default is K=120, but the periodicity can be set. A performance experiment consists of a 64K data transfer, a 4 byte data transfer, and two storage requests to a local NWS persistent state server. If the file space is NFS mounted, the persistent state server may also generate traffic as it writes the data out to disk. This data (which is essentially the measurement data) is stored on a locally mounted file system to avoid traffic. In short, the system has been engineered to be as non-intrusive as possible.

4.1.3 System Sensitive Load Distribution

The following example illustrates the operation of the system sensitive partitioner. Consider a cluster with four nodes with the synthetic load generator used to load two of the nodes. Using the current system state provided by NWS, the Capacity Calculator computes the relative capacities C_1, C_2, C_3 and C_4 as approximately 16%, 19%, 31% and 34%, respectively. In these calculations, the three system characteristics, viz. CPU, memory, and link bandwidth, were assumed to be equally important to the application, i.e. $w_p = w_m = w_b = 1/3$. The relative capacities are then fed to the ACEHeterogeneous partition, which uses them to partition the overall work load L among the four processors. In this case the four processors are assigned work loads of $L_1 = 0.16L$, $L_2 = 0.19L$, $L_3 = 0.30L$ and $L_4 = 0.34L$, respectively. The partitioner appropriately assigns boxes (breaking large boxes if necessary) to processors to satisfy this distribution.

In our experiments, we have chosen the weights to be equal, but they do not have to be so and can in fact be chosen more carefully according to the computational needs of a particular application. For example, if the application is memory intensive, then a larger value can be assigned to w_m in comparison to w_p and w_b.

4.1.4 Dynamic Load Sensing

The system-sensitive partitioner queries NWS at runtime to sense system load, computes the current relative capacities of the processors and distributes the workload based on these capacities. The sensing frequency depends not only on the dynamics of the cluster but also on the overheads associated with querying NWS and computing relative capacities, and therefore has to be chosen to balance the two factors. More frequent sensing (every few regrid steps) allows the system to adapt to rapid changes in system load but increases the sensing overheads. On the other hand, infrequent sensing (a few times during application execution) reduces these overheads but may increase overall execution time by preventing system from reacting to the cluster's load dynamics. The load could be sensed in two different ways:

1. NWS could be probed using a separate thread and the system state could be stored in a database. This can be done while the application is still running
2. NWS could be probed using a system call from the application itself.

The various overhead associated with dynamic sensing are discussed below:

- When NWS is probed using a separate thread, the overhead is mainly the "partitioning overhead" due to the unsorted list of capacities obtained every time NWS is queried. In MPI, the mapping of processors to machines can be specified using the "machinefile" and this file is initialized right at the beginning of the simulation. MPI schedules the jobs on these machines as it receives the load. So, when NWS is queried after the simulation is started, we need to read the machine names from the "machinefile" and store their respective capacity in the "Capacity file". Hence, we get an unsorted list of capacities. However, when we query NWS only once before the start of the simulation, we sort the list of capacities and put the machines in the "machinefile" accordingly. This sorting helps in reducing the breaking of boxes because the capacities as well as the box list are sorted. A smaller capacity requires a smaller box size and hence there is less

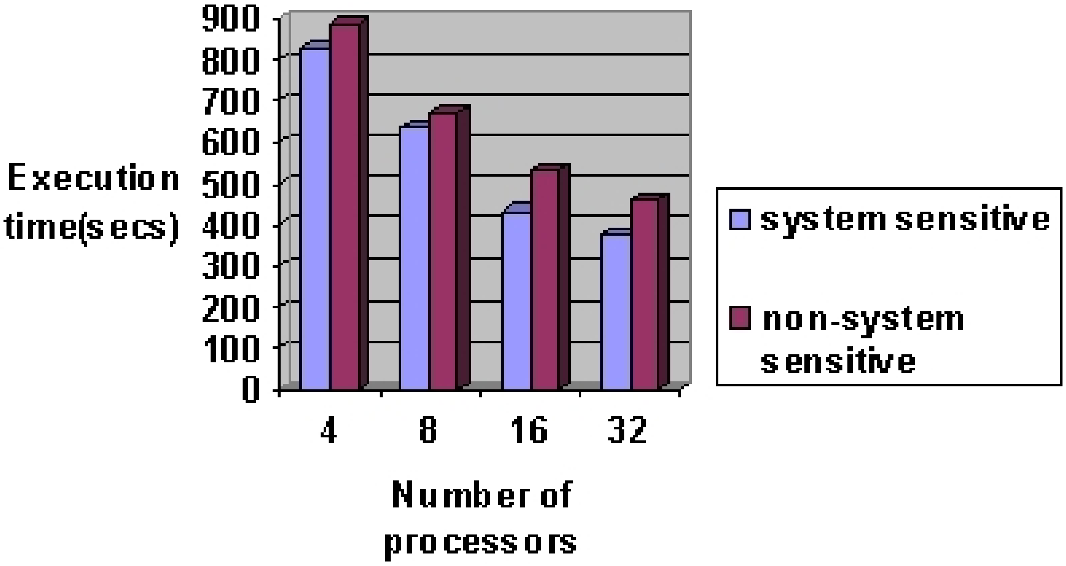

Figure 4.2: *Comparison of execution time of the two partitioning schemes*

breaking of boxes. On the other hand if the capacities are unsorted which is the case when we query NWS often, then we might have to break more boxes which leads to an overhead. Hence, this overhead should be considered before sensing very frequently.

- When NWS is probed using a system call from the application itself, the application would have to bear the overhead of "probing NWS" as well as the "partitioning overhead." Our experiments show that the overhead of probing NWS on a node, retrieving its system state, and computing its relative capacity is about 0.5 secs per node. The advantage of sensing more frequently is that more up to date information is received.

Number of Processors	Percentage Improvement
4	7%
8	6%
16	18%
32	18%

Table 4.1: *Percentage improvement that the system-sensitive partitioner yields over the default partitioner.*

4.2 Results

4.2.1 Application Performance Improvement

The total application execution time using system-sensitive partitioning and the default non system-sensitive partitioner is plotted in Figure 4.2. The percentage improvements are listed in Table 4.1. In this experiment, the application was run under similar load conditions using the two partitioners. We calculate the relative capacities of the processors once before the start of the simulation. System-sensitive partitioning reduced execution time by about 18% in the case of 32 nodes. We believe that the improvement will be more significant in the case of larger cluster and in cluster with greater heterogeneity and load dynamics. Furthermore, increasing the sensing frequency also improves performance as shown in a later experiment.

4.2.2 Load Balance Achieved

This experiment investigates the load assignments and the effective load balance achieved using the two partitioners. The work load assignment for the 4 processor run, under the ACEComposite partitioning scheme has been plotted in Figure 4.3. The plot shows the work load assignment after every 5 iterations. The work assignments for the processors are similar to each other because the ACEComposite partitioning scheme does not consider the relative capacities and distributes the work equally among the processors.

The work load assignment under the ACEHeterogeneous partitioning scheme has been

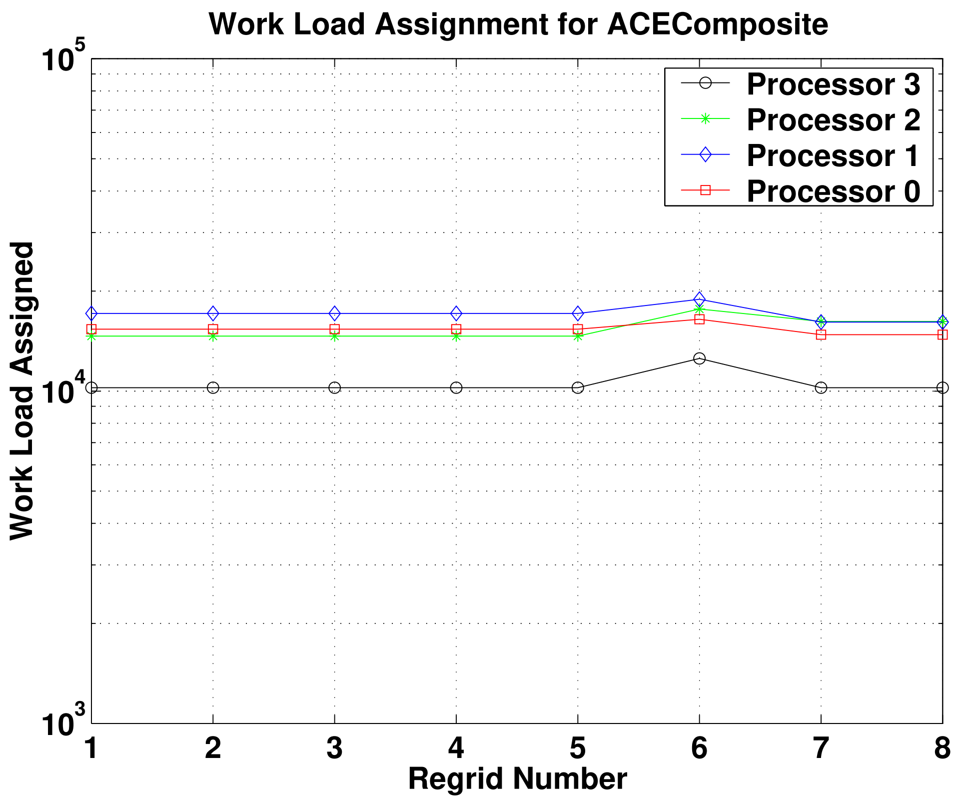

Figure 4.3: *Work-load assignments for the ACEComposite partitioning scheme (relative capacities of processors* $0-3$ *are* $16\%, 19\%, 31\%,$ *and* 34% *respectively).*

plotted in Figure 4.4. The relative capacities of the processors were approximately 16%, 19%, 30% and 34%. In this case, it can be seen from the plot that the work load was assigned in proportion to the relative capacities of the processors.

Figure 4.5 shows the percentage of load imbalance for the system-sensitive scheme and the default scheme. For the kth processor, the load imbalance I_k is defined as

$$I_k = \frac{|W_k - L_k|}{L_k} \times 100 \quad \% \tag{4.1}$$

As expected, since the default partitioning scheme distributes work equally regardless of the different relative capacities, the load imbalance is higher than that of the system-sensitive scheme. Note that the load imbalances in the case of the system sensitive partitioner are due to the constraints (minimum box size and aspect ratio) that have to be satisfied while breaking boxes. In spite of this, the load imbalance associated with it is significantly lower than that of the default partitioning scheme.

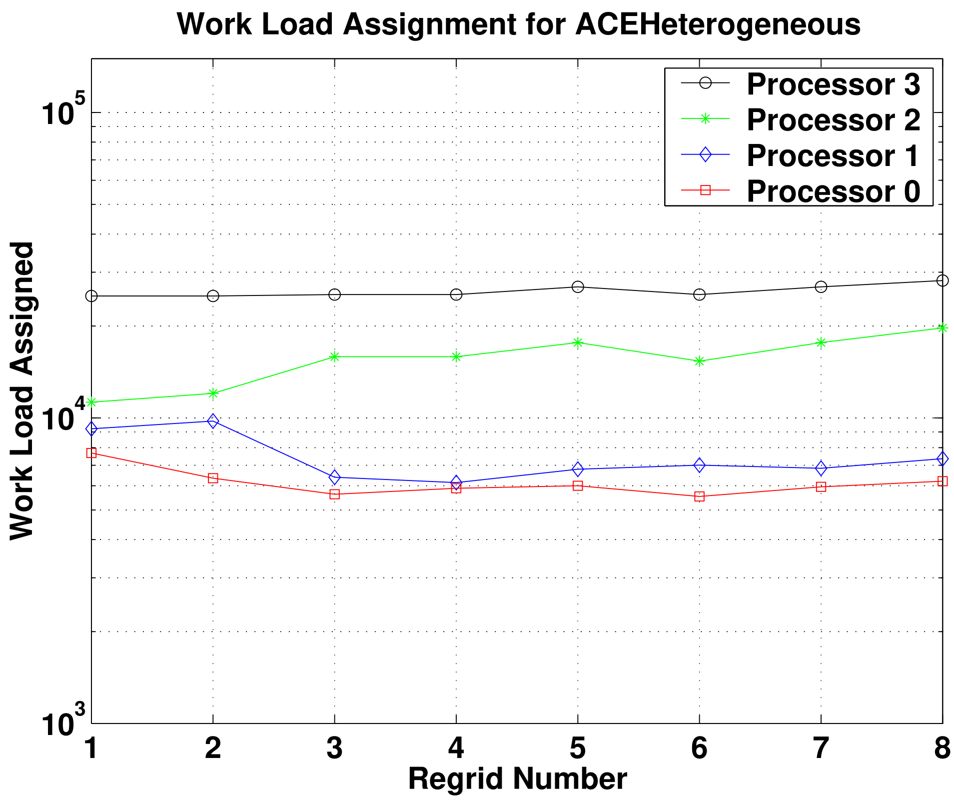

Figure 4.4: *Work-load assignments for the ACEHeterogeneous partitioning scheme (relative capacities of the processors* $0-3$ *are* $16\%, 19\%, 30\%,$ *and* 34% *respectively).*

4.2.3 Adaptivity to Load Dynamics

This experiment evaluates the ability of the system-sensitive partitioner to adapt to the load dynamics in the cluster, and the overheads involved in sensing the current state. In this experiment, the synthetic load generator was used on two of the processors to dynamically vary the system load. The load assignments at each processor were computed for different sensing frequencies. Figure 4.6 shows the load assignment in the case where NWS was queried once before the start of the application and two times during the application run. The figure also shows the relative capacities of the processors at each sampling. It can be seen that as the load (and hence the relative capacities) of the processors change, the partitioning routine adapts to this change by distributing the work load accordingly. Also note that as the application adapts, the total work load to be distributed varies from one iteration to the next. As a result, the work load assigned to a particular processor is different in different iterations even

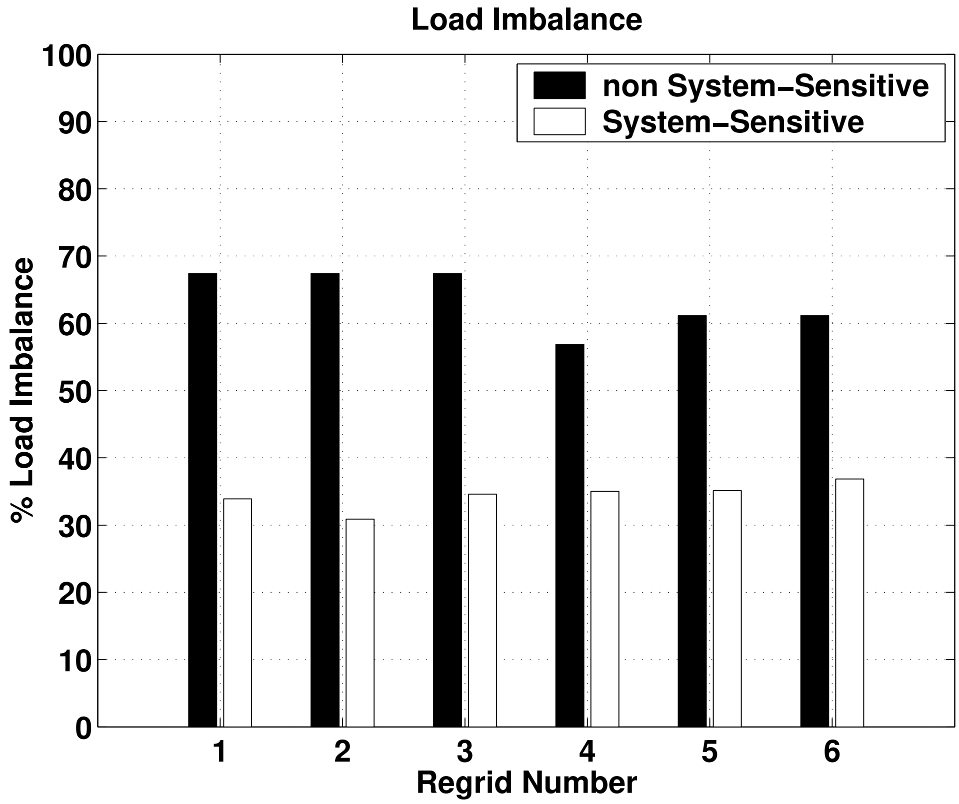

Figure 4.5: *Comparison of load imbalance for the two schemes.*

though the relative capacity of the processor does not change.

Tables 4.2 and 4.3 illustrate the effect of sensing frequency on overall application performance. The synthetic load dynamics are the same in each case. Table 4.2 compares the application execution times for the cases where the system state is queried only once at the beginning and where it is queried every 40 iterations. It can be seen that dynamic runtime sensing significantly improves application performance. Table 4.3 presents application run time for different sensing frequencies - i.e. sensing every 10, 20, 30 and 40 iterations. Figures 4.7, 4.8, 4.9 and 4.10 show the relative processor capacities and load assignments for each case. The best application performance is achieved for a sensing frequency of 20 iterations. This number largely depends on the load dynamics of the cluster overhead and the sensing overhead.

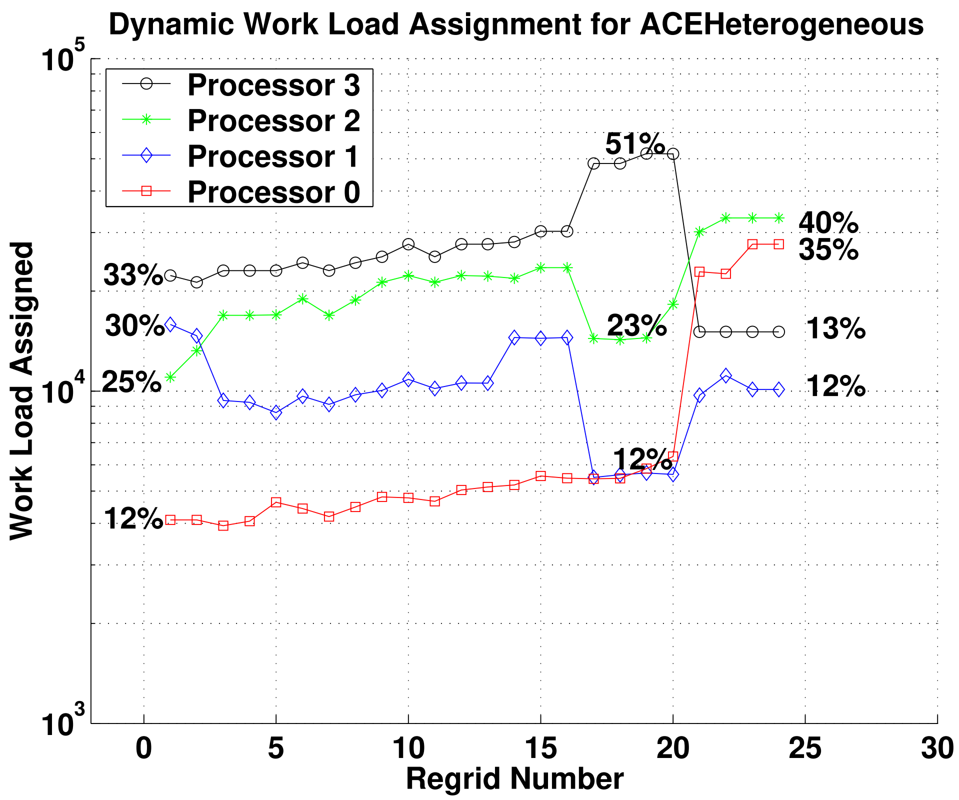

Figure 4.6: *Dynamic load allocation using the system-sensitive partitioner.*

4.3 Summary of Results

- System-sensitive partitioning reduced execution time by about 18% in the case of 32 nodes as compared to the default partitioning scheme.

- The system-sensitive partitioning scheme distributes the work load in proportion to the relative capacities of the processors. Furthermore, the load imbalance associated is up to 45% lower than that of the default partitioning scheme.

- Through dynamic sensing, the system-sensitive partitioner adapts to the load dynamics of the cluster. As the load (and hence the relative capacities) of the processors change, the partitioning routine adapts to this change by distributing the work load accordingly. Dynamic runtime sensing also improved application performance by as much as 45% compared to sensing only once at the beginning of the simulation.

Number of Processors	Execution time with Dynamic Sensing (secs)	Execution time with Sensing only once (secs)
2	423.7	805.5
4	292.0	450.0
6	272.0	442.0
8	225.0	430.0

Table 4.2: *Comparison of execution times using static and dynamic sensing*

Frequency of calculating capacities	Execution time (secs)
10 iterations	316
20 iterations	277
30 iterations	286
40 iterations	293

Table 4.3: *Execution time for a four processor run when NWS is probed at different iterations.*

- The sensitivity of application performance to the frequency of sensing the system state was also studied. In our experimental setup sensing very frequently adversely effected the application performance. The best application performance was achieved for a sensing frequency of 20 iterations. The frequency of sensing largely depends upon the load dynamics of the cluster.

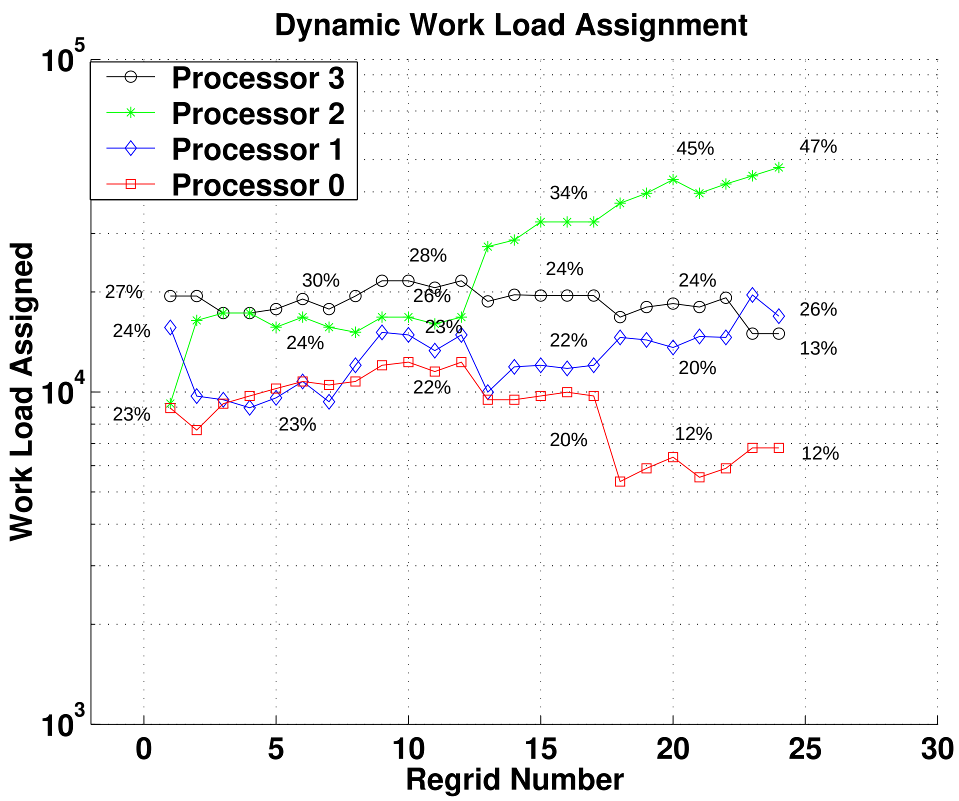

Figure 4.7: *Dynamic load allocation for a system state sensing frequency of 10 iterations.*

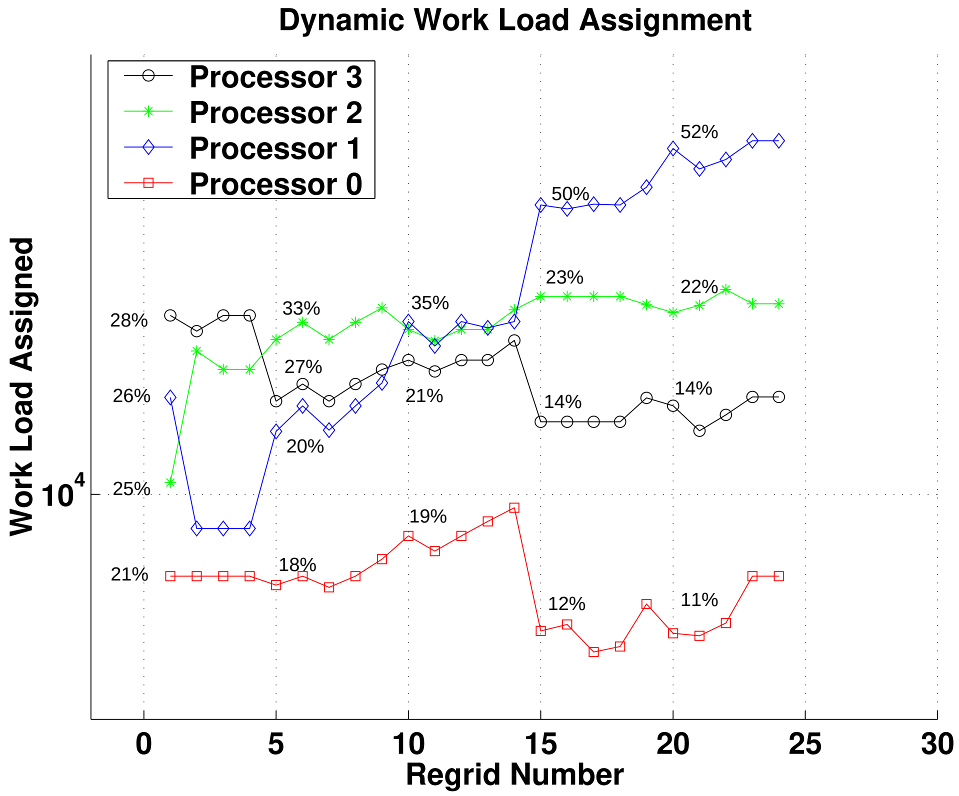

Figure 4.8: *Dynamic load allocation for a system state sensing frequency of 20 iterations.*

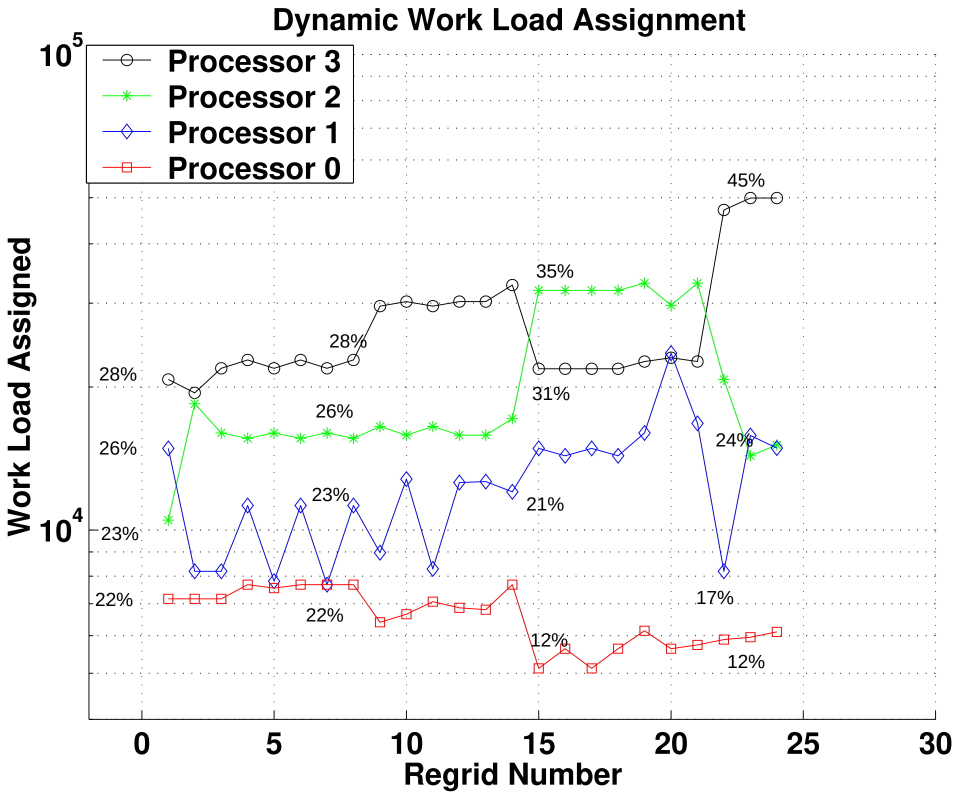

Figure 4.9: *Dynamic load allocation for a system state sensing frequency of 30 iterations.*

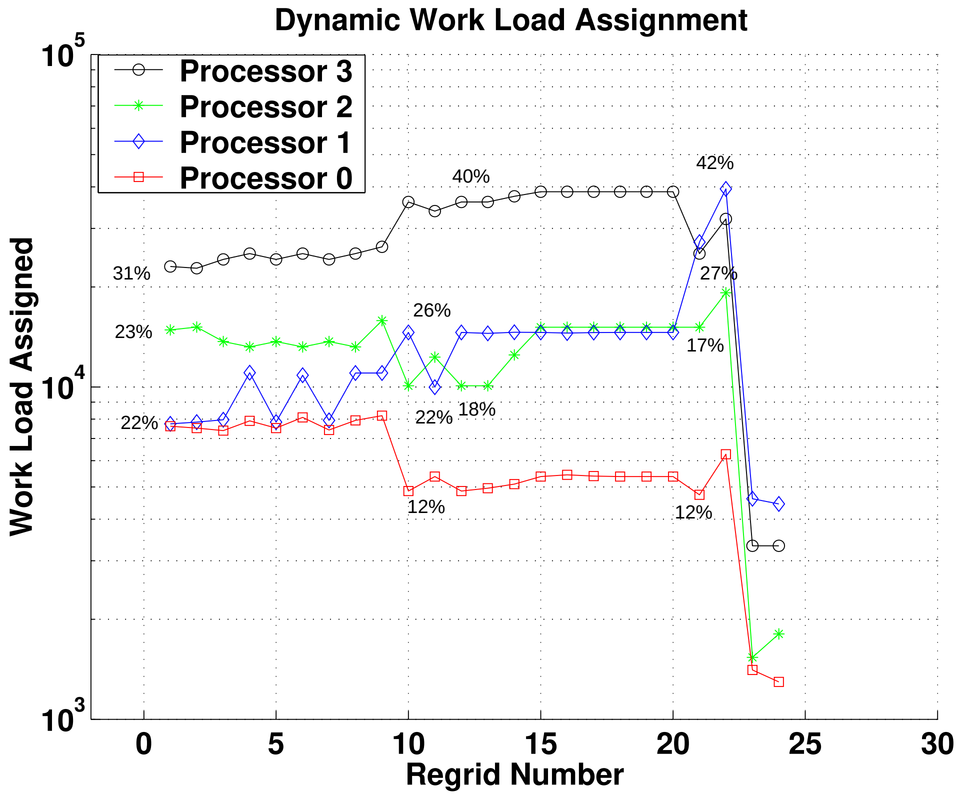

Figure 4.10: *Dynamic load allocation for a system state sensing frequency of 40 iterations.*

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis presented a system sensitive partitioning and load-balancing framework for distributed AMR applications in heterogeneous cluster environments. The framework uses system capabilities and current system state to appropriately distribute the application and balance load. The framework have been implemented within the GrACE runtime system. The system state is dynamically monitored using NWS, and is used to compute current relative capacities of the computing nodes. These capacities are then used for partitioning and load balancing. An experimental evaluation of the framework was presented using a 3-D AMR CFD kernel executing on a Linux based cluster. A synthetic load generator was used to load the nodes in the cluster for the experiments. The experiments showed that dynamic system-sensitive partitioning can significantly improve the load balance achieved and overall application performance. Furthermore, application performance is sensitive to the frequency of sensing of the system state. This frequency has to be chosen based on the sensing overheads and the cluster load dynamics.

5.2 Future Work

It was shown through simulation that the system-sensitive partitioning scheme not only reduces the total execution time of the application but also reduces the load imbalance as compared to a scheme that does not take the relative capacities of the computing nodes into account. We can extend this work to accommodate a more careful choice of weights w_p, w_m, and w_b which will adequately reflect the computational needs of the application. Recall that in our scheme, w_p, w_m, and w_b are the weights associated with the relative CPU, memory, and link bandwidth availabilities, respectively, subject to the constraint that $w_p + w_m + w_b = 1$. In our experiments, although we have chosen the weights to be equal, they do not have to be so and can in fact be chosen more carefully according to the computational needs of a particular application. For example, if the application is memory intensive, then a larger value can be assigned to w_m in comparison to w_p and w_b.

Although the ACEHeterogeneous partitioning scheme exhibits a lower load-imbalance compared to the ACEComposite scheme as shown in Figure 4.5, the load-imbalance of the former can be reduced even further. A primary cause of load-imbalance in the ACEHeterogeneous scheme can be attributed to the fact that the bounding box is cut only along the longest axis. If the box is instead cut along more axes, it could lead to finer partitioning granularity, and hence better work assignments, which would in turn reduce the load-imbalance.

NWS also has the capability to forecast the state of the resources. This predictive ability of NWS could be used and evaluated within the framework provided in this thesis.

Also, in our cost model we have taken into account the state of CPU, memory and link bandwidth but other system parameters could be used as well.

Appendix A

A.1 Network Weather Services (NWS)

The NWS has been constructed using four different component processes [3]:

- **Persistent State** process: It stores and retrieves measurements from persistent storage.
- **Name Server** process: implements a directory capability used to bind process and data names with a low-level contact information
- **Sensor** process: gathers performance measurements from a specified resource.
- **Forecaster** process: produces a predicted value of deliverable performance during a specified time frame for a specified resource.

A.1.1 NWS Sensors

The function of an NWS Sensor is to gather and store time stamp-performance measurement pairs for a specific resource. Each sensor process may measure several different performance characteristics of the resource it is sensing. The TCP/IP Network sensor, for example, provides both bandwidth and end-to -end round trip latency measurements, but each set of measurements is named and stored separately.

A.1.2 CPU Sensor

The NWS CPU Sensor combines information from Unix system utilities uptime and vmstat with periodic active CPU occupancy tests to provide measurements of CPU availability on timeshared Unix systems. CPU availability is measured as a fraction of CPU occupancy time a full-priority standard user process can obtain. Typically, the uptime utility reports load average as the average number of processes in the run queue over the past one, five and fifteen minutes. The CPU sensor uses the one-minute measurement to calculate the fraction of the CPU occupancy time that a process would get if it were to run at the moment the uptime measurement were taken. From vmstat output, the CPU sensor uses a combination of idle time, user time, and system time measurements to generate an estimate of the available CPU occupancy fraction.

These utilities generate their reports using internal Unix system variables and hence invoking them does not generate significant load. That is, they are fairly non-intrusive monitoring utilities. However, both may leave out considerable information that can affect measurement of CPU availability. In order to obtain more accurate measurements, the CPU sensor incorporates active probes into its calculations. It periodically runs an artificial, compute-intensive probe program and calculates the CPU availability as the ratio of its observed CPU occupancy time to the wall-clock time of its execution. The sensor then compares the results of these probes to the measurements taken using uptime and uses the utility that is reporting the most accurate information.

A.1.3 Network Sensor

Because end-to-end network performance data between arbitrary machines is not consistently available, NWS network sensors rely on active network probes exclusively when determining network load. Each probe consists of a timed network operation, such as the movement of a fixed amount of data, or, in the case of TCP, the establishment and dissolution of a network connection. At regular intervals, each network sensor connects to a set of peer sensors running on machines of interest and conducts one or more probes of different types to gather its measurements. To gather a set of end-to-end performance measurements of any type from N sensors would require $N^2 - N$ probes. To avoid introducing this much load, Sensors are organized hierarchically so that an end-to-end measurement can be made for a representable subset of the total Sensor population. These representative measures can then be used to describe the network performance between an arbitrary Sensor pair.

Currently, the NWS network sensor is capable of measuring three network performance characteristics: small message round-trip time, large-message throughput, and TCP socket connect-disconnect time. The small message probe consists of a 4-byte TCP socket transfer that is timed as it is sent from a source Sensor to a destination Sensor and back. The socket connection used to facilitate the transfer is already established before the probe is conducted. Large message throughput is calculated by timing the transfer of a message using TCP and the acknowledgment of its receipt by the receiving sensor.

To make experiment results available, the Sensors contact Persistent State processes to store the information. The location of the Persistent sate process that a Sensor will use for each of the measurements it gathers is specified when the Sensor is configured. When it is initialized, each Sensor registers the location of the Persistent state process that stores its measurement data with the Name Service so that measurement data may be located by name.

The daemons that run continuously on the nodes are *nws_memory*, *network_sensor*, and *nws_forecast*. Only *nws_memory* requires disk storage. They listen and communicate with the outside world using TCP ports 8050, 8060, and 8070 respectively. *network_sensor* and *nws_forecast* both may fork, but neither executes an *exec* and all run with standard user privileges. *nws_extract* and *ctrl_network_sensor* are utility programs that extract data/forecasts from the system and provide administrative control respectively [3].

References

[1] M. Parashar and J. C. Browne, "*System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement,*" in Volume 117: Structured Adaptive Mesh Refinement Grid Methods, IMA Volumes in Mathematics and its Applications. Springer-Verlag, pp. 1-18, 2000.

[2] M. Parashar and J. C. Browne, "*On Partitioning Dynamic Adaptive Grid Hierarchies*," Proceedings of the 29th Annual Hawaii International Conference on System Sciences, Wailea, HI, pp. 604-613, January 1996.

[3] R. Wolski, N. T. Spring and J. Hayes, "*The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*," Future Generation Computing Systems, Vol. 15, pp. 757-768, October 1999, www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html.

[4] M. J. Berger and J. Oliger, "*Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*," Journal of Computational Physics, Vol.53, pp. 484-512, March 1984.

[5] S. M. Figueira and F. Berman, "*Mapping Parallel Applications to Distributed Heterogeneous Systems*, " UCSD CS Tech Report # CS96-484, June 1996.

[6] J. Watts, M. Rieffel and S. Taylor, "*Dynamic Management of Heterogeneous Resources*," High Performance Computing: Grand Challenges in Computer Simulation, pp. 151-156, April 1998.

[7] M. Maheswaran and H. J. Seigel, " *A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems*," 7th IEEE Heterogeneous Computing Workshop (HCW '98), Orlando, pp. 57-69, March 1998.

[8] M. Shee, "*Evaluation and optimization of load balancing/distribution techniques for dynamic adaptive grid hierarchies*," M.S. Thesis, Department of Electrical and Computer Engineering, Rutgers University, May 2000.

[9] M. Ripeanu, "*Issues of Running Large Scientific Applications in a Distributed Environment*," M.S. Thesis, Computer Science Department, University of Chicago, November 2000.

[10] G.C. Fox, M. Johnson, D. Walker, "*Solving problems on Concurrent processors*," Prentice Hall 1988.

[11] H. Sagan, "*Space-filling curves*," Springer-Verlag, 1994.

[12] R. Fagin, "*Extendible Hashing - A Fast Access Mechanism for Dynamic Files*," ACM Transactions on Database Systems, pp. 315–344, 1979.

[13] P. H. Andersen, J. K. Antonio, "*Implementation and Utilization of a Heterogeneous Multicomputer Cluster for the Study of Load Balancing Strategies*," Proceedings of the Seventh Institute of Electrical and Electronics Engineers, Inc. (IEEE) International Symposium on High Performance Distributed Computings (HPDC-7), Chicago, pp. 362-363, July 1998.

[14] W. Cai, F. L. Bu-Sung, and A. Heng, "*A Simulation Study of Dynamic Load Balancing for Network-based Parallel Processing*," in Proceedings of 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97), IEEE Computer Society Press, Taipei, pp. 383-389, 14-16 Dec. 1997.

[15] T. Gross, P. Steenkite, and J. Subhlok, "*Adaptive Distributed Applications on Heterogeneous Networks*," in proceedings of 8th IEEE Heterogeneous Computing Workshop (HCW '99), Puerto Rico, April 1999.

[16] W. Leinberg, G. Karypis, V. Kumar, and R. Biswas, "*Load Balancing Across Near-Homogeneous Multi-Resource Servers*," in proceedings of the 9th IEEE Heterogeneous Computing Workshop (HCW 2000), Mexico, May 2000.

[17] S. Ramanathan, " Performance optimization for Dynamic Adaptive Grid Hierarchies," M.S. thesis, Department of Electrical and Computer Engineering, Rutgers University, May 2001.

[18] J. Steensland, S. Chandra, M. Thune, and M. Parashar, " Characterization of domain-based partitioners for parallel SAMR applications," in Proceedings of parallel and Distributed Computing and Systems(PDCS 2000), Las Vegas, volume 2, pp. 425-430, November 6-9, 2000.

[19] G. Karypis, K. Schloegel, and V. Kumar, "ParMetiS: Parallel graph partitioning and sparse matrix ordering library," Technical report 97-060, Dept. of Computer Science, University of Minnesota, Minneapolis, 1997.

[20] S. Sinha, M. Parashar, "*System Sensitive Runtime Management of Adaptive Applications*, " accepted by the 10th IEEE Heterogeneous Computing Workshop (HCW 2001), San Francisco, April 23rd 2001.

[21] B. Hendrickson and K. Devine, "*Design of Dynamic Load-Balancing Tools for Parallel Applications*," accepted by the International Conference on Supercomputing, Santa Fe, NM, pp. 110-118, May 2000.

[22] H. Topcuoglu, S. Hariri, and M. Wu, "*Task Scheduling Algorithms for Heterogeneous Processors*," in proceedings of 8th IEEE Heterogeneous Computing Workshop, Puerto Rico, pp. 3-14, April 1999.