12-13-2003

Integrating Algorithmic and Systemic Load Balancing Strategies in Parallel Scientific Applications

Sheikh Khaled Ghafoor

INTEGRATING ALGORITHMIC AND SYSTEMIC LOAD BALANCING

STRATEGIES IN PARALLEL SCIENTIFIC APPLICATIONS

By

Sheikh Khaled Ghafoor

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2003

INTEGRATING ALGORITHMIC AND SYSTEMIC LOAD BALANCING STRATEGIES IN PARALLEL SCIENTIFIC APPLICATIONS

By

Sheikh Khaled Ghafoor

Approved:

Ioana Banicescu Associate Professor of Computer Science and Engineering (Major Professor)	Rayford B. Vaughn Jr. Associate Professor of Computer Science and Engineering (Committee Member)
Anthony Skjellum Professor of Computer and Information Sciences University of Alabama at Birmingham (Committee Member)	Susan M. Bridges Professor of Computer Science and Engineering (Graduate Coordinator)

A.Wayne Bennett
Dean of the Bagley college of Engineering

Name: Sheikh K. Ghafoor

Date of Degree: December 13, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Ioana Banicescu

Title of Study: INTEGRATING ALGORITHMIC AND SYSTEMIC LOAD
BALANCING STRATEGIES IN PARALLEL SCIENTIFIC
APPLICATIONS

Pages in Study: 67

Candidate for Degree of Master of Science

Load imbalance is a major source of performance degradation in parallel scientific applications. Load balancing increases performance of parallel applications in distributed environments. At a coarse level of granularity, advances in runtime systems have been proposed in order to control available resources using task migration. At a finer granularity level, advances in algorithmic strategies for dynamically balancing loads by data redistribution have been proposed. Algorithmic and systemic load balancing strategies have complementary set of advantages. An integration of these two techniques should result in a system, which delivers advantages over each technique used in isolation. This thesis presents a design and implementation of a system that combines an algorithmic load balancing strategy called Fractiling with a systemic load balancing system called Hector. It also reports on experimental results of running N-body simulations under this integrated system. The experimental results indicate that the integrated system provides performance improvement for large applications.

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation for the people who helped me in many ways during the course of this thesis. My earnest gratitude goes to my major professor, Dr. Ioana Banicecu, who supported me generously, guided me with constant encouragement and imperative directions, and had the patience to accommodate my mistakes to make me learn from them. I am very grateful to Dr. Anthony Skjellum, member of my graduate committee, for his suggestions and help during the work on this thesis. I would also like to thank Dr. Rayford B. Vaughn, for serving as a member in my graduate committee, providing me with useful suggestions and comments. I am very grateful to Dr. Samuel H. Russ, former associate professor of Electrical and Computer Engineering, for his guidance and help during the work on this thesis. In addition, I would like to thank Rong Lu, former. student of this department for his assistance. My work on this thesis has been partially supported by the National Science Foundation through the following grants: CAREER # 9984465, ITR/ACS # 0081303,ITR # 0085969, # 973038. I would also like to take the opportunity to acknowledge the support for infrastructure and resources provided by Engineering Research Center of Mississippi State University. I wish to thank all the faculty members here at the Computer Science and Engineering Department from whom I learned a lot. Finally I would like to thank my wife Ambareen Siraj for her encouragement, support, and for her continuous help through the writing of this thesis

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Parallel and distributed computing has become one of the most interesting avenues followed in scientific applications and has become one of the fundamental research areas of computer science. Problems in science and engineering are often large, complex, highly irregular and computationally intensive. These problems can often be decomposed into sub problems that can simultaneously be solved. Thus, parallelization provides a way to solve large computationally intensive problems like ocean modeling, climate modeling fluid turbulence etc., which would otherwise be impossible to solve on a sequential machine. One factor, which typically influences parallel programming, is the type of processor communication used. The way processors communicate depends on the memory architecture, which can be classified as shared memory and distributed memory. In shared memory architectures multiple processors operate in an independent fashion but all share the same memory resources. Shared memory systems are difficult to scale as the number of processors increase. In distributed memory architectures, each processor has its own address space and operates in an independent manner. The processors are connected through the interconnection network and data sharing across the communication network is in general performed through message passing.

In general, we associate high performance with parallel and distributed computing. There are several factors that affect the performance of parallel applications running in a parallel and distributed computing environments. Some of these factors are: the choice of parallel algorithm used, load imbalance, the type of interconnection network, and others. Load imbalance is one of the major performance degradation factors in parallel scientific applications and by balancing the workload, their performance can significantly be improved [3, 27]. Scientific applications are in general data parallel. There are several factors that cause load imbalance in parallel scientific applications running in a distributed computing environment. A few major factors are: non-uniform data distribution, different computational requirements in various data partitions, variations in external workload on different computational nodes, operating system (OS) and network effects.

With the increase in performance of commodity desktop workstations, advancement in high speed networks, and development of architecture independent ways to code parallel programs, such as MPI[20] and PVM[16], Network of Workstations (NOW) or Cluster of Workstations (COW) are becoming a cost effective popular choice for parallel and distributed computing. The operating systems for the workstations were initially developed for interactive sequential jobs with a single processor in mind. Over time, support for multiprocessing and networking has gradually been incorporated into operating systems. However, the commercial operating systems for workstations still do not offer adequate support for a transparent execution of parallel or sequential jobs over a NOW. The workstations in NOW are used by individuals, and the load across the network varies dynamically, as users execute applications or relinquish workstations. This, along with other reasons mentioned earlier cause load imbalance when running applications in parallel and distributed environments. Since the operating systems or message passing libraries (such as MPI) do not provide support for load balancing across workstations in NOW, executing parallel applications on NOW often leads to severe load imbalance and poor resource utilization. This problem can be alleviated by addressing the load balancing problem through migration of tasks (coarse-grain) or data (fine-grain) from the highly loaded workstations to the lightly loaded ones or idle workstations. Therefore, in a NOW environment, load balancing can be performed at both fine and coarse levels of granularity.

Since load imbalance is one of the major performance degradation factors in data parallel scientific applications, providing solution(s) to this problem is an important computer science issue. Finding a generic solution that can dynamically balance load with low overhead could significantly improve the performance of parallel scientific applications. Even if the solution is applicable to only a class of applications, it will have a significant impact on performance of data parallel applications running in distributed computing environments. In the present work, an attempt has been made to find a solution to the load imbalance problem in a complex class of data parallel applications running in distributed computing environments: the N-body simulations.

As there are several factors which cause load imbalance, finding algorithms and methods for addressing this problem in parallel and distributed computing environments is a complex problem. Over time, various techniques to balance load at coarse and fine levels of granularity have been proposed. In general, an individual processor's performance may vary due to external workload, or non-uniform data distribution within an application, as well as other factors. Therefore, methods to maintain an even distribution of work are usually needed in order to obtain good speedup and performance. In a distributed computing environment, coarse-grained strategies have been proposed at the system level, while fine-grained strategies have been proposed at the algorithmic level. By coarse-grained strategies at the system level, we mean that the load balancing is performed by the host operating system or runtime system. No modifications in the applications or algorithms are required by the user or programmer. By fine-grained strategies, we mean that the load balancing algorithm is built into the applications; the host operating system or runtime libraries are unaware of the load balancing performed by the applications.

1.1 Systemic (Coarse-Grained) Load Balancing

In task-parallel applications, load balancing at the coarse-grain level is achieved via task migration. This involves transferring of a program's state from one processor to another during runtime. Task-parallel applications have advantages such as: a natural mapping to the operating system (i.e. the entire process is transferred) and the ability to release resources (such as workstations) back to individual users by moving the work elsewhere, and freeing up both the CPU and the memory.

Systemic load balancing via task migration from heavily to lightly loaded processors is typically coarse-grained and can be supported by two distinct methods.

First, users can write their own state-transfer routines which can be invoked by the runtime system to migrate or checkpoint a job. Systems such as LSF [26, 46] and DQS [13] work in this fashion. The disadvantages of these systems are that they put the burden of checkpointing onto the application developer and therefore, the routines must be actively maintained along with the rest of the source code. The alternative is to provide systemic support for checkpointing and migration. Condor [32, 42], and Hector [33] work in this fashion. However, the Hector distributed runtime environment used in this thesis is unique in the depth and breadth of information gathered about tasks at runtime. Hector runtime system supports the migration of parallel tasks. These are capabilities that can be exploited by data-parallel load balancers. In general, the systemic load balancing is application independent and implemented at the system level (operating system, communications library, or middleware), relieving the application programmer from this responsibility.

1.2 Algorithmic (Fine-Grained) Load Balancing

Algorithmic load balancing via data migration is supported by the applications and is typically fine-grained. Data-parallel programs use data migration (or dynamic data allocation) to maintain balanced loads and therefore are “self-balancing”. This represents a finer grain of control than task migration, because only fractions of a program state have to be moved. Tasks can either negotiate as peers to exchange data from busy tasks to idle ones, or have a central master that allocates data to worker tasks. Systems based on Factoring [22] and Fractiling [5, 6] are examples of the former, and Piranha [11] is an example of the latter.

Fractiling is a dynamic scheduling technique based on a probabilistic analysis that adapts to algorithmic and systemic load imbalances while maximizing data locality. It draws from earlier loop scheduling techniques where iterates are dynamically scheduled in decreasing size chunks to reduce synchronization. It has successfully been implemented in N-body simulations [5, 6]. The early large chunks have relatively little overhead and their uneven finishing times are smoothed over by later smaller chunks. Fractiling uses a tiling technique to optimize chunk shapes such that data locality and reuse are maximized.

1.3 An Integrated Strategy

Advances in runtime systems for parallel programs have been proposed in order to control available resources as efficiently as possible. Simultaneously, advances in algorithmic methods of dynamically balancing computational load have been proposed in order to respond to variations in actual performance. Both, coarse- and fine-grained strategies have advantages and disadvantages. The coarse-grained approach may suffer from load imbalance due to the unequal sizes of tasks, or the total number of tasks that may not always be an integral multiple of the number of workstations in the cluster. On the other hand, in the fine-grained approach, due to the absence of migration capability, the resource utilization is limited to the workstations in use, and no use of new workstations may be acquired or removed during the application execution. Let us consider a scenario where in a sixteen processor cluster (p_0 .. p_{15}), six processors are available, and the cluster uses Hector as runtime system. A user lunches a parallel job with eight tasks. Since Hector works at task level it will assign four tasks to four processors (say p_0 – p_3) and two tasks/processor to the rest of the processors (p_4 and p_5). As a result, tasks running on p_4 and p_5 will finish their computation later than tasks running on p_0 through p_3. In the middle of the execution, if one or more processors become available, Hector can move additional tasks from p_4 and p_5 to newly available processors. Since tasks running on p_4 and p_5 shared the processor before migration they will still finish later than tasks running on p_0 through p_3. If the parallel application would have had incorporated the Fractiling algorithm, it would have balanced the workload among the tasks by using dynamic data redistribution before and after migration. Thus, all the processes would have finished almost at the same time. Let us consider another scenario in which a fractiled scientific application is running on a cluster. While fractiled tasks are running, one or more processors become overloaded due to some additional external load. The Fractiling algorithm will now balance the load by migrating data from tasks running on overloaded processors to lightly loaded processors. Let's suppose that during the execution some other processors become idle. In the absence of Hector, the idle processors cannot be utilized. If Fractiling would have had the capability of task migration in a Hector-like fashion, the fractiled tasks from the overloaded processors could have been migrated to idle processors. In this way, better resource utilization would have been achieved because idle resources would have been utilized. Therefore, in this respect, Hector and Fractiling complement each other.

An ideal runtime system should provide support for both systemic and algorithmic strategies since they have complementary sets of advantages. The systemic coarse-grained strategy considers all tasks from all applications on the system, while the algorithmic fine-grained strategy is confined to individual applications. Once the programmer has expressed the algorithm to be used, the runtime system should execute the program efficiently, taking maximum advantage of available resources. It may have to migrate entire tasks in order to relinquish processors back to "owners". If it does not have to migrate an entire task, it is desirable to move only the amount of data needed to rebalance the load. The essential point is that these load balancing strategies can work in concert to provide additional benefits to one another. The resulting integrated load balancing strategy is systemic in nature, and therefore the burden on the applications programmer is reduced. Moreover, the integration provides an improved performance for parallel applications over the improvements obtained by using either strategy individually.

The present work called Hectiling proposes to combine the load balancing methodology used in Hector, a distributed runtime environment which provides coarse-grained dynamic load balancing for parallel applications on Sun and SGI workstations, with Fractiling, a fine-grained dynamic load balancing technique based on a probabilistic analysis that has been proven to be effective in scientific applications (i.e. N-body simulations). Hectiling should offer load balancing at both levels of granularity and provides a more efficient utilization of resources than either technique used in isolation. This thesis presents the design and implementation of Hectiling, and reports on experimental results of running N-body simulations under this integrated system. The N-body simulations consider N particles, their positions and velocities, and the problem is to compute the forces they exert on each other, and then calculate their new positions. The N-body simulations have been selected as a test application because it requires solutions of multiple algorithms, and is a complex and computationally intensive problem. It has been widely used in a broad class of application areas of science such as astrophysics, molecular dynamics, biophysics, molecular chemistry etc. N-body simulations employ algorithms, which are used in other areas, such as volume visualization. Therefore, if a technique provides performance improvement for N-body simulations it should applicable for a wide range of scientific applications.

1.4 Hypothesis

The hypothesis of this thesis is two fold:

1. The integration of an algorithmic load balancing strategy (Fractiling) with a systemic load balancing strategy (Hector) is possible.
2. For applications, which employ the N-body simulation algorithms, this integration will result in achieving better performance than applying any of these techniques independently. The overhead introduced by the combined (integrated) approach will be small and will be outweighed by the benefit of improved load balancing due to integration. The integrated system will perform no worse than any of the techniques applied in isolation. In other words for the integrated system the following inequality will hold:

$C_{Hectiling} \leq \text{Min} (C_{Fractiling}, C_{HPFMA})$

Where:

$C_{Hectiling}$ is the Parallel execution cost using Hectiling

$C_{Fractiling}$ is the Parallel execution cost using Fractiling

C_{HPFMA} is the Parallel execution cost using Hector

1.5 Approach

The work plan that has been followed in the process of validating the hypothesis is as follows:

1. Survey different algorithmic load balancing techniques and algorithms. Study the Fractiling algorithm in detail and analyze implementation of a parallel application that has employed the Fractiling algorithm for load balancing. For the present work, two parallel implementations of the N-body simulations (one with Fractiling and one without Fractiling) have been selected.
2. Study and analyze the architecture and implementation of Hector.
3. Design an integrated architecture: Hectiling, to combine Fractiling and Hector
4. Implement the integrated architecture.
5. Execute the following experiments and collect timing results
    i. Select a set of data representing different data sizes and data distributions.

ii. Execute following parallel implementations of the N-body simulations on various numbers of processors (up to 32) with each dataset selected at “i.”

1. Straightforward parallelization.
2. Straightforward parallelization under Hector.
3. With Fractiling.
4. With Fractiling under Hector
5. With Hectiling (Fractiling and Hector integrated).

6. Evaluate the overhead of integration experimentally.
7. Select a set of metrics to measure the performance. Provide a qualitative and quantitative analysis of the performance of Hectiling using the experimental results. Validate the hypothesis.

1.6 Expected Contributions

The expected contributions from this thesis are as follows:

1. Provide an integrated strategy to improve the performance of data parallel scientific applications.
2. Provide an implementation of a runtime system (a modified Hector) for easy integration of any data parallel scientific application that incorporates Fractiling algorithm for load balancing.
3. Provide implementation guidelines for integrating data parallel scientific applications with Fractiling into Hector.

4. Provide an estimate about the amount of effort it takes to integrate an application with Fractiling into Hector.
5. Provide an qualitative and quantitative analysis of performance and overhead of Hectiling (see Approach 6 and 7).

1.7 Organization of this Thesis

This thesis is organized as follows. Chapter 2 presents the pertinent background and related work in the areas of systemic and algorithmic load balancing. Chapter 3 describes the design and implementation of Hectiling. Result and analysis are presented in Chapter 4, and finally, Chapter 5 presents conclusion and future work.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Related Work on Systemic (Coarse-Grained) Load Balancing

In the past years, many systems that run sequential and parallel programs on networks of workstations, shared memory processors (i.e., using SMPs), and massively parallel processors (MPP), have been proposed and successfully implemented. Differing in their degree of sophistication and in the methods used to balance the computational load, they offer a variety of features and services. A comprehensive survey of task-based job-scheduling systems has been presented by Baker, Fox and Yau[2]. Features that such systems may contain include: scheduling of sequential and parallel jobs, load balancing, task migration, the nature and complexity of runtime information gathering, and others. Only few of these systems are enhanced to support task migration, and if they do, the migration applies only to sequential jobs. In general, migration could be supported using two distinct methods. First, users can write their own state transfer routines, which can be invoked by the runtime system to migrate or checkpoint, a task. Systems such as LSF [26, 46] work in this manner. The alternative is to provide support for task migration and checkpointing by the runtime system. Systems such as Condor [32] work in this fashion.

All systems mentioned in the survey provide some degree of load balancing at task granularity level. This load balancing is static in nature, in the sense that at the time of launching a job, the entire system load and the scheduling of tasks to achieve load balancing across the entire system are considered. No further action is taken by the runtime system after launching a job if system load varies for any reason such as termination of another job (which could translate into load imbalance of the parallel job at hand). To the best of our knowledge, none of the systems mentioned so far in the literature provides support for migration of parallel tasks or sequential communicating tasks. Therefore, there is a need to design runtime systems with support for task migration that can provide dynamic load balancing during job execution.

One of the clustering systems presented in the survey by Baker, Fox, and Yau [2] is LSF [46]. It is a widely used commercial package for controlling clusters. LSF works by launching utility tasks on each candidate host to monitor usage and to provide remote job-launch capability. The usage monitor reports to a central master, which uses the data to decide which nodes are available for running jobs. It runs parallel jobs, supports task migration through user-level checkpointing, and gathers node usage information. The information is used to control the initial mapping of tasks to hosts. Condor [32], developed at University of Wisconsin, is another clustering system presented in the above-mentioned survey. It is a widely used public-domain cluster management software package. It groups workstations into "flocks", monitors their availability, and only runs parallel jobs if they are designed to tolerate variable numbers of hosts during execution. Workstation load average is used for allocation and the system can either migrate tasks (with system-level checkpointing) or kill them when the workstation becomes busy with external applications. Condor and LSF systems use a distributed architecture design. In this context, by distributed architecture we mean that the components of the clustering system are distributed among its nodes. Both Condor and LSF use relatively coarse load information for initial allocation purposes and for determining if hosts are idle or busy. Both the systems don’t gather information from running tasks and in addition, LSF does not support systemic checkpointing.

Recent work has highlighted the benefits of extracting information from applications during runtime [14]. For example, Nguyen et al. have shown that extracting runtime information can be minimally intrusive and can substantially improve the performance of a parallel job scheduler [39], whereas Gibbons proposed a simpler system to correlate runtimes to different job queues [17]. In either case, information gathered from tasks as they run can support job scheduling and allocation. The Hector distributed runtime environment is intended to support this model [37]. It uses a distributed architecture, provides system-level checkpointing routines, supports execution of unmodified MPI programs, and gathers extensive information during runtime about the performance of hosts and individual tasks. Hector is designed to provide an infrastructure that controls parallel programs during their execution and to monitor their performance. Therefore it combines the benefits of both distributed and centralized processing. The central decision-maker and control process is called a master allocator or “MA”. Running on each candidate platform (where a platform can range from a desktop workstation to a SMP) is a supervisory task called a slave allocator or “SA”. The SA's gather performance information from the tasks (MPI processes) under their control and execute commands issued by the MA. Thus, Hector combines the functions of monitoring and execution contained in LSF's two distributed daemon processes [46].

Hector's instrumentation combines three different mechanisms [33-37]. First, static host information is gathered by the SA when it is launched. Second, dynamic host information is gleaned from a series of system calls to read memory usage and CPU usage. Third, Hector's modified MPI library provides task self-instrumentation that is monitored by the SA. This instrumentation includes a breakdown of time spent communicating and computing, as well as a map of the task's communication topology.

Task migration is supported by the run time system and a specially modified version of MPI to properly handle messages in transit. In this way, applications do not need code changes in order to support task migration [33]. Both Hector and Hectiling use MPICH, an implementation of MPI by the Argonne National Laboratories and Mississippi State University.

2.2 Related Work on Algorithmic (Fine-Grained) Load Balancing

Load balancing at the application level is algorithmic and fine-grained. Therefore load balancing techniques at this level of granularity have to be integrated into a specific application. Selecting a technique that offers best performance and is relatively simple to integrate is essential to the success of the resulting application. While load balancing can be applied to all parallel applications, scientific applications are of particular interest due to their intensive computational requirements. In addition, large classes of scientific applications are irregular in nature, and therefore their performance is severely degraded due to load imbalance. Imbalance over a few time steps of the computation could primarily be caused by changes in data distributions. Furthermore, within one time step, imbalance could be caused by irregularity of data distribution, different processing requirements of interior versus boundary data, and by system effects.

Problems in scientific computing are in general data-parallel and have previously employed various methods to balance processor loads and to exploit locality. For example, in unstructured problems, static partitioning and repetitive static partitioning heuristics have been the only methodology used so far to overcome dynamic load imbalance [9, 10, 23, 38, 40, 41, 45,]. Most of these methods use profiling by gathering information on the workload from a previous time step in the execution of an algorithm in order to estimate the optimal workload distribution at the present time step. "Profiling", in this context, refers to a detailed performance analysis that is only available after the program is finished, or at least after the current program iteration is completed. The cost of these methods increases with the number of processors and problem size [39, 40, 44, 45]. A random assignment of certain sized amounts of work to processors has also been considered to improve the performance of simulations affected by load imbalance [18]. With random assignment, the load imbalances of individual work units mute each other out to some extent. However, performance of these scientific applications is then severely degraded by loss of locality.

Another important observation is that the above methods employ a static assignment of workload to processors during a time step, due to an assumption that the data distribution changes slowly between time steps. These assumptions are not valid in the entire spectrum of scientific applications and therefore these methods are not robust, especially in the case of applications where none of the existing load balancing strategies accommodates the unpredictable behavior of simulations (i.e. plastic deformations, nonisothermal multiphase flow, etc.). Therefore, there is a need for developing new techniques that address load imbalances between time steps, as well as during a time step.

Dynamic scheduling schemes attempt to maintain balanced loads by assigning work to idle processors at runtime. Thus, they accommodate systemic as well as algorithmic variances. In general, there is a tension between exploiting data locality and dynamic load balancing as the re-assignment of work may necessitate access to remote data. The cost of dynamic schemes is loss of locality, which translates into increased overhead. Another potential shortcoming involves the amount of data exchanged among tasks to balance the load. If the amount of data is too large, the resulting corrections might be too coarse. If the amount of data is too small, the process of exchanging data might incur much overhead. Thus, in master/worker parallelism if the increment of workload that the master distributes is too small or too large, this might lead to either inefficiency or imbalance.

Since loops are the most prevalent source of parallelism in scientific applications, their scheduling on parallel machines has received considerable attention. The fundamental tradeoff when scheduling parallel loops is processor load imbalance versus overhead due to synchronization and communication. Parallel loop scheduling schemes have been widely analyzed and measured [25, 28, 31, 43].

Factoring, a scheduling scheme that evolves from earlier loop scheduling techniques, balances processor loads while reducing the overhead of synchronization [22]. Loop iterates are dynamically scheduled in decreasing size chunks such that early larger chunks have relatively little overhead, and their uneven finishing times are smoothed over by later smaller chunks. The technique minimizes the cumulative contributions of load imbalances and scheduling synchronization. A technique for reducing communication, called Tilling, statically partitions the iteration space into tiles whose shape is chosen to maximize data reuse and locality. Factoring selects the optimal chunk sizes, (i.e. how many iterates to group together), while Tiling selects optimal chunk shapes (i.e. which iterates to group together).

Another technique, Fractiling, combines the load balancing advantages of Factoring with the data reuse properties of Tiling [3, 21]. In this combined scheme, chunk sizes are determined globally according to a Factoring rule, while chunk shapes are determined locally according to a Tiling rule. The Fractiling method was developed in response to the shortcomings of other methods and has successfully been applied to N-body simulations [4, 6]. It is based on a probabilistic analysis, and therefore accommodates load imbalances caused by predictable events (such as irregular data) and unpredictable events (such as data access latency). Fractiling adapts to algorithmic and system induced load imbalances while maximizing data locality. In Fractiling, the computation space is initially placed to processors in tiles, to maximize locality. Processors that finish early "borrow" decreasing size subtiles of work units from slower processors to balance loads. The sizes of these subtiles are chosen so that they have a high probability of finishing before the optimal time. Subtile assignments are computed in an efficient way by exploiting the self-similarity property of fractals. These decreasing size chunks are represented by multidimensional subtiles of the same shape selected to maximize data reuse. The subtiles are combined in Morton order in larger subtiles, thus preserving the self-similarity property [4, 6]. Early in the program run, large performance variations can be accommodated by exchanging large subtiles. As the computation progresses, the subtiles shrink so that smaller variations can be corrected. By having subtile sizes based on a uniform size ratio, a complex history of executed subtiles does not need to be maintained. Each task simply keeps track of the size of its currently executing subtile, and in this way, the unit of data exchange among tasks is the largest subtile currently being executed by any task. Thus the algorithm inherently minimizes the global "bookkeeping" overhead.

This technique allows negotiations by idle resources to replace profiling. The load balancing actions are a function of performance, in the sense that idle processors have performed well, but are not a function of a direct performance measurement. Rather, they simply exchange work from "busy" processors to "idle" ones. This reduces overhead, as detailed data collection is not needed, and increases responsiveness, as load balancing can occur during an iteration step. The bulk of load balancing work is performed by idle tasks and therefore little negative effect on runtime is expected. Additionally, Fractiling does not take into account the source of load imbalance in order to spur useful performance gains. Even applications where the amount of computation per data element varies dynamically can benefit, because it would simply have to search for idle and busy resources.

In the implementation of Fractiling in a distributed environment, one of the processors selected as master and called Fractiling Master controls and maintains the entire data exchange information. In addition, it performs computation as all the other processors do, called Fractiling Tasks. When computation starts, the Fractiling Master divides the computation space into P tiles, one per processor. Each Fractiling Task starts by working first on half of its tile. When this subtile is finished, the Fractiling Task sends a Fract_Ask message to the Fractiling Master to request additional work. The Fractiling Master updates its information and assigns a new subtile size to the requesting Fractiling Task. If a Fractiling Task completes its own tile, and there is still work left in other Fractiling Task's tile, the Fractiling Master sends a request to another Fractiling Task to send data to the idle Fractiling Task. The data is then forwarded to the idle Fractiling Task, which works on the received data and sends the result back to the owner. The above process is repeated until there is no more work left in any Fractiling Task's tile. When assigning subtiles to the Fractiling Tasks, the Fractiling Master always observes the following rules: (i) a task will have to have all the work completed in its own tile before starting to help another Fractiling Task; (ii) after completing its own tile, a Fractiling Task will always work on a tile with the largest available unfinished subtile size.

Experimentation on both a distributed memory shared-address space and a message passing environment with Fractiling schemes applied to N-body simulations have been presented in [3, 4, 6]. The distributed memory shared-address space implementation was run on a KSR-1 at the Cornell Theory Center and the message passing environment implementation was run on an IBM SP2 at the Maui High Performance Computing Center. In experiments involving both uniform and nonuniform data distributions, performance of N-body simulation codes was improved by as much as 53% by Fractiling. The corresponding coefficient of variation of processor finishing times among the simulation tasks was extremely small, indicating a very good load balance was obtained. Performance improvements were obtained even on uniform data distributions, underscoring the need for a scheduling scheme that accommodates system-induced variance in addition to the algorithmic one.

CHAPTER III

DESIGN AND IMPLEMENTATION

Hector achieves better resource utilization by migrating tasks from highly loaded workstations to idle or lightly loaded workstations. Since task sizes are unequal, an application using this coarse-grained load balancing strategy only will continue to suffer from load imbalance. On the other hand, applications employing fine-grained data parallel load balancing strategies, such as Fractiling, ensure a high degree of load balancing by migrating data from one task to another. However, in a distributed computing environment an application using Fractiling may suffer from poor resource utilization, because task migration is not supported. One or more of the processors executing Fractiling tasks may become heavily loaded by other applications, thereby significantly degrading the performance of the Fractiling application. Having the capability to migrate a Fractiling task from a heavily loaded to an idle or lightly loaded processor would enable the Fractiling application to utilize resources more efficiently.

To take advantage of the benefits offered by Hector and Fractiling, a new system integrating both has been designed and implemented. This system, Hectiling, combines systemic information gathering and task migration capabilities of Hector with fine-grained algorithmic load balancing advantages of Fractiling. Before describing the integrated architecture, the following two sections present the architecture of Hector and centralized management implementation of Fractiling.

3.1 Hector Architecture

Hector is designed around a master-slave hierarchy. Figure 1 shows the architecture of Hector. There is a single task called the Master Allocator (MA) that performs all of the decision-making functions. This task doesn’t control MPI programs directly, but communicates with tasks called Slave Allocators (SA). There is one slave allocator per node. Each slave allocator controls all MPI tasks running on its machine, and monitor their performance characteristics. It reports the performance information back to the MA, which makes decision about allocation and migration. The MA periodically collects information from every node on the network. If required, it then sends a command to migrate a targeted task to the slave allocator that launched the tasks.

The slave allocators are directly involved in the process of migrating an MPI task. They notify a task that needs to migrate, track the status of migration, and notify the master that migration has completed. The SAs communicate with the MPI tasks under its control by maintaining a permanent UNIX socket at a predetermined port number, which allows the tasks to send information about their current status. The communication mechanisms and protocols used by the SAs to pass control information is an important part of Hector design and it is done through \a listener process attached to each MPI task.

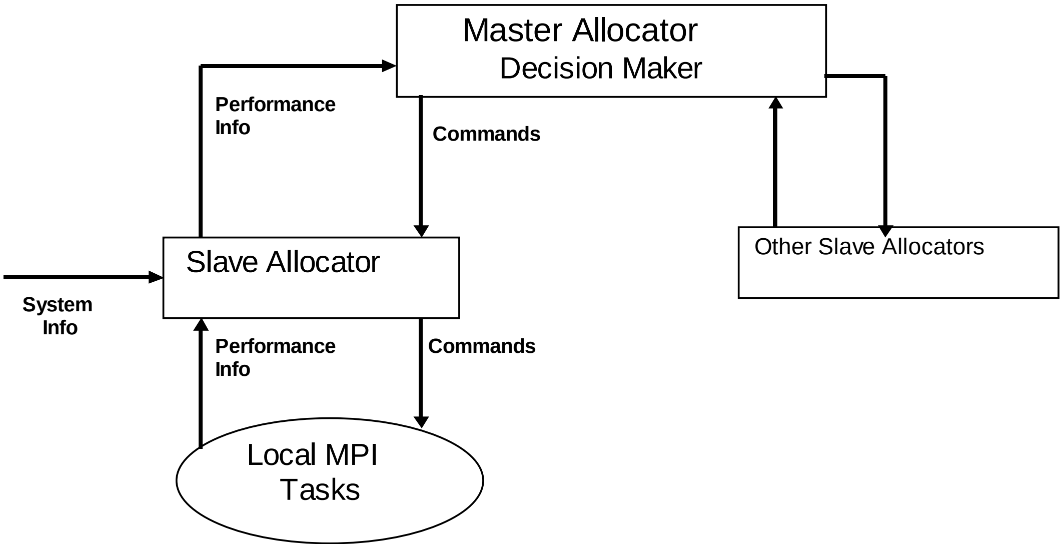

Figure 3.1 Hector Architecture

Task migration is the most important feature of Hector. There are three aspects to task migration. First, it is necessary to encapsulate a program’s state completely. Second, the state must be transferred to the destination as efficiently as possible. Third, the state must be reconstructed correctly and in such a way as not to corrupt the MPI environment. The process of task migration is shown in Figure 2 and the steps are as follows:

1. When the MA decides to migrate a task, it sends a message to the appropriate SA, which in turn sends migration message to that task’s listeners.
2. The listener finishes handling any other events such as establishing a connection, and sends a control signal to the tasks.
3. The task sends a notification about its pending migration to all other tasks’ listeners and begins waiting for End Of Channel (EOC) messages from other tasks.

4. After all EOC messages have been received, the task closes all active connections.
5. The MA informs the SA on the destination node and the task is spawned with the arguments to read in the program state.
6. After the task has restarted, it sends its new location information to all other tasks' listeners.
7. The task sends a message back to the SA that the migration is complete and it is now available for migration again. Further details of Hector architecture and task migration can be found in [33-37].

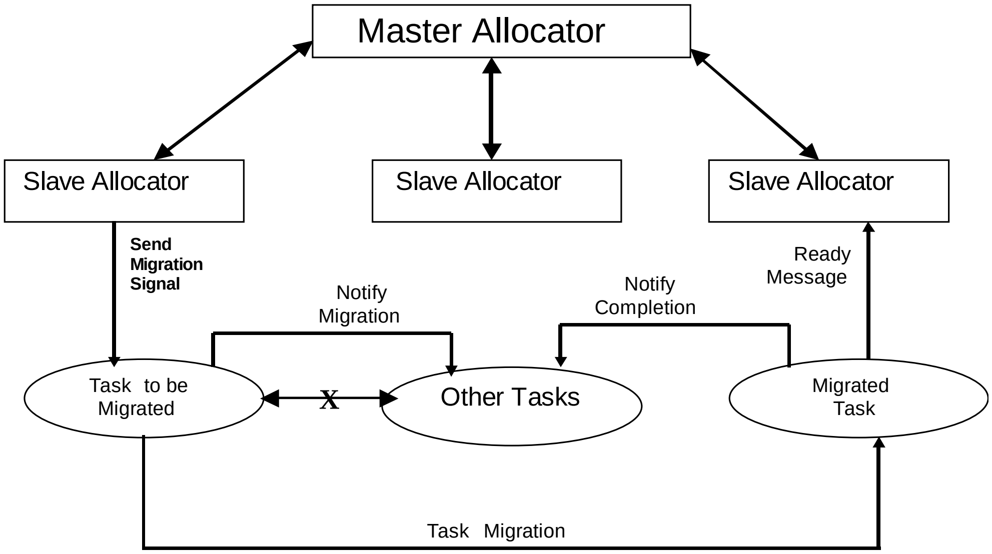

Figure 3.2 Migration of Task under Hector

3.2 Fractiling Implementation

Fractiling adapts to algorithmic and system induced load imbalances while maximizing data locality. In Fractiling, the computation space is initially placed to processors in tiles, to maximize locality. Processors that finish early "borrow" decreasing size subtiles of work units from slower processors to balance loads. The sizes of these subtiles are chosen so that they have a high probability of finishing before the optimal time. Subtile assignments are computed in an efficient way by exploiting the self-similarity property of fractals. Early in the program run, large performance variations can be accommodated by exchanging large subtiles. As the computation progresses, the subtiles shrink so that smaller variations can be corrected. By having subtile sizes based on a uniform size ratio, a complex history of executed subtiles does not need to be maintained. Each task simply keeps track of the size of its currently executing subtile, and in this way, the unit of data exchange among tasks is the largest subtile currently being executed by any task. Thus the algorithm inherently minimizes the global "bookkeeping" overhead.

In a centralized management implementation of Fractiling scheme, one processor is selected as master, which manages the global variable and schedule data among other processors. Thus, Fractiling also works around a master/slave hierarchy. The Fractiling communication pattern is shown in Figure 3. Fractiling divides the computation space into P tiles, one tile per processor. At the beginning each processor works on the half in its own tile. If a processor finishes its first half, it sends a FRACTILE_ASK message to the master. The master receives the message looks up the global variables, and then it assigns a job (subtile) to the requesting processor with FRACT_REPLY mesaage. The requesting processor receives the answer and continues to work. If the requesting processor completes its own tile and there is work available in other processor’s tile, the master will assign a subtile size in a neighboring processor, and then sends a

FRACT_COMM message to tell the neighboring processor to send its data to the helper (requesting processor). Meanwhile, the master sends FRACT_REPLY to the requesting processor indicating which processor is to be helped. The neighbor receives the message, and sends its data to the helping processor using FRACT_ORG_DATA. The helper receives the FRACT_ORG_DATA and works on the data. After completion, it sends a FRACT_ASK to the master to request a new job, and also sends the result to the processor (FRACT_FIN_DATA) that owns the data. The owner receives the data and stores it. The above steps are repeated until no subtiles are left.

When assigning subtiles, the master processor always observes the following rules:

- After completing its own tile a processor will help another processor to complete its tile.
- After completing its own tile, a processor will always work on the largest subtile available.
- At any time, the processor will finish its own tile first, then help other processors.

With the combination of these features, Fractiling improves data locality and reduces load imbalance.

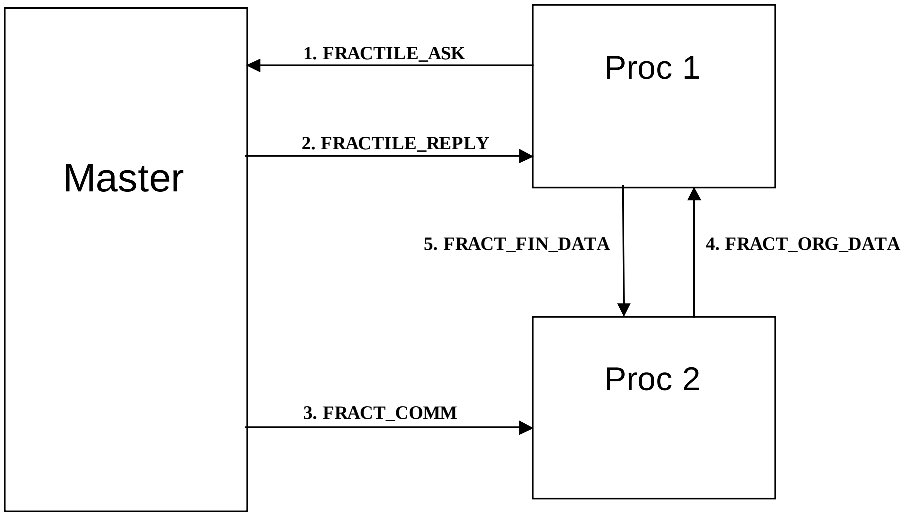

Figure 3.3 Master/Slave communication in Fractiling

3.3 Hectiling Design and Implementation

The architecture of Hectiling is shown in Figure 4. Since Fractiling requires communications to control exchanges of data between tasks, and Hector has a built in information gathering infrastructure, it was decided in the first phase of this design to a re-routing of “Fractile_Ask messages” from Fractiling Tasks to the Fractiling Master via the MA. This requires a communication channel from Fractiling Tasks to the MA. The integration imposes several challenges. In the Hector paradigm, the MPI tasks do not communicate with the MA. Thus, a communication mechanism has to be devised from a task to the MA, and care has to be taken so that non-Fractiling tasks, where task-to-MA communication is not required, could also run under the same integrated system. To accomplish this, the location and port number of the MA must first be conveyed to all

Fractiling Tasks. Once the Fractiling Master receives this information, it "registers" with the MA by opening a socket and sending its port number and host name to the MA. As a result, the MA is able to recognize which of the tasks is the Fractiling Master and where to forward the Fractile_Ask messages. During the execution of the Fractiling application, when the MA receives a Fractile_Ask message, it first checks to see if the Fractiling Master has been "registered". If so, the message is forwarded to the Fractiling Master. If not, the message is put into a queue which, has already been created at the beginning of the execution of the Fractiling application. This queue is being maintained by the MA throughout the execution of the application. Once the Fractiling Master registers with the MA, all pending messages are forwarded to it. At the same time, the MA sends a message to the Fractiling Master's SA, which in turn interrupts the Fractiling Master allowing it to read the associated message from its socket (see Figure 5). This mechanism was designed to address the fact that UNIX does not allow task interrupts on remote machines.

The integration also imposes another challenge on Hector migration mechanism. In Hector, all the MPI tasks are treated equally, and the migration process is the same for all the tasks. However, in Hectiling the migration of the Fractiling Master is different from the ones of Fractiling Tasks. This is due to the fact that the MA needs to forward the Fractile_Ask message to the Fractiling Master. Thus, the MA has to have the information about the location of the Fractiling Master, and this is achieved by the registration process of Fractiling Master presented above.

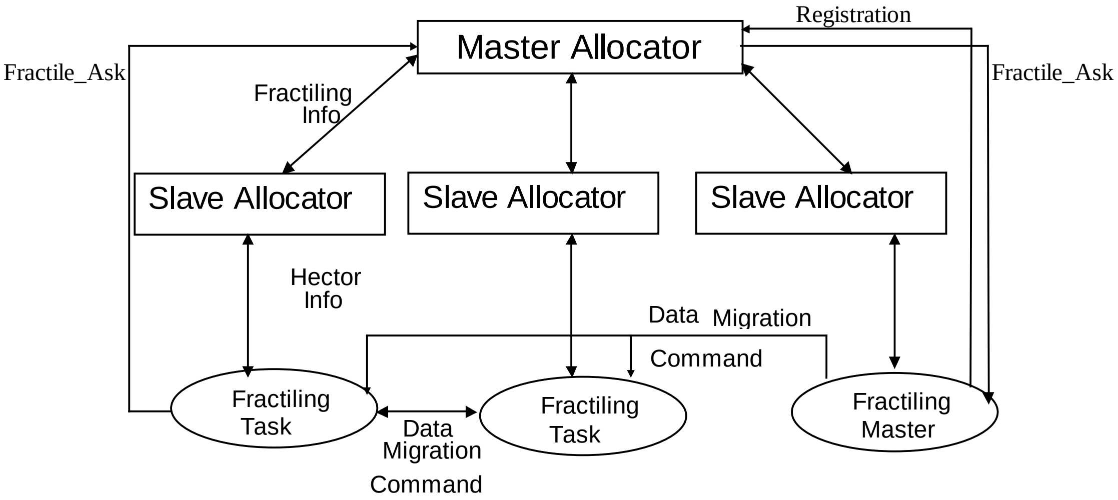

Figure 3.4 Hectiling Architechture

In case of migration, the Fractiling Master first un-registers itself with the MA, and upon completing the migration, it re-registers itself again with the MA. The un-registration process consists of two steps. First, when the MA decides to migrate the Fractiling Master, it sends an End-of-Channel message to the Fractiling Master, and stops forwarding any Fractile_Ask message to it. If the MA receives any Fractile_Ask messages from the Fractiling Tasks before the migration is complete, it queues these messages. This process ensures that no Fractile_Ask message is lost during the migration of the Fractiling Master. In the second step, the Fractiling Master closes its socket as soon as it receives the End-of-Channel message, and only then the migration could start. The re-registration process involves the opening of a new socket and sending of the associated port number and the new host name to the MA. After re-registration, the MA sends any messages queued during the migration to the Fractiling Master.

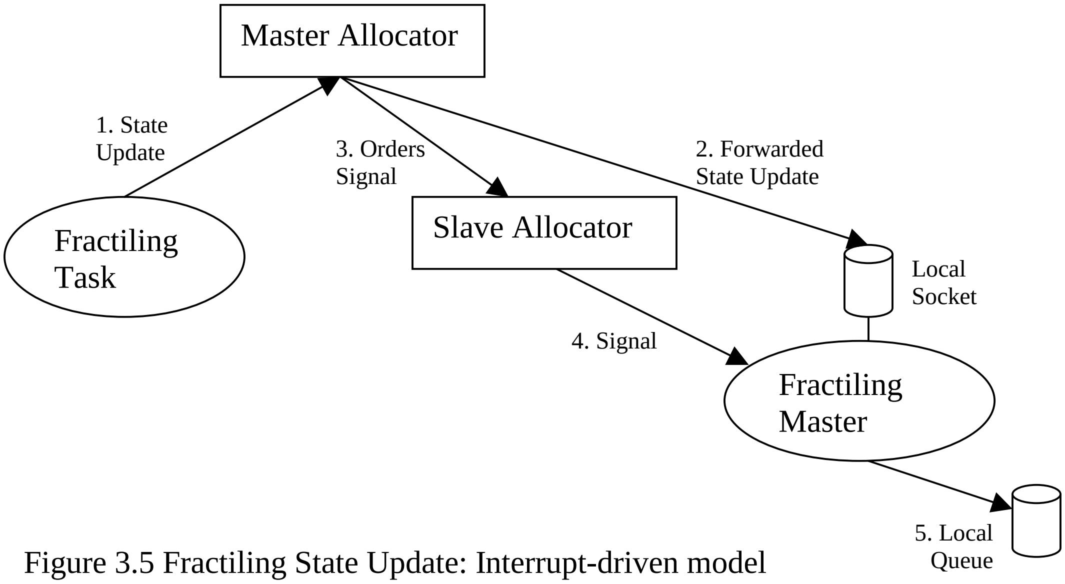

Figure 3.5 Fractiling State Update: Interrupt-driven model

CHAPTER IV

EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS

The experiments with the integrated system were conducted in two phases. In the first phase, Hectiling experiments were conducted without process migration. The results are described in section 4.1. Section 4.2 describes the results of experiments with Hectiling using process migration. Experiments were conducted on a system which consists of thirty-two 90 MHz Ross HyperSPARC processors arranged in a cluster of eight 4-processor machines. Each of the machines is a SMP running Solaris 2.6. The machines are connected by three interconnection technologies: (i) 155 Mbits/sec ATM switches, (ii) Myrinet, (iii) 10 Mbits/sec Ethernet. Any of them could be used for communication between machines. The ATM interconnection has been used in the experiments presented here. The experiments were conducted with three different data distributions: a uniform distribution ("Uniform"), a nonuniform Gaussian distribution ("Gaussian"), and a nonuniform Gaussian distribution with the center shifted to the center of one of the octants of the computation space ("Corner"). Each distribution has four different data sizes: 10K particles, 20k particles, 50k particles and 100k particles. In total we conducted the experiments with 12 different data sets. All the executions were carried out three times and the result of the three executions were averaged. The metrics that has been chosen to measure the performance of different techniques are the parallel cost and the coefficient of variation (C.O.V) of processors finishing times. They are defines as follows:

Cost = P X T_P

P = Number of processor used

T_P = Execution time of the processor which finishes last

$$c.o.v = (\sqrt{\sum_{i=1}^{n} \frac{(x_i - \mu)^2}{n-1}}) / \mu$$

x_i = Execution time of an individual processor

n = Number of processors

μ = Mean of x_i s

For each experiment individual processor finishing time was measured, from this parallel cost and coefficient of variation of individual processor finishing time was calculated.

4.1 Hectiling without Migration

For testing in phase one, five implementations of the N-body simulations based on the Parallel Fast Multipole Algorithm (PFMA) by Greengard [19] have been used: (i.) without Fractiling (PFMA); (ii.) with Fractiling (Fractiling); (iii.) under the Hector environment and without Fractiling (HPFMA); (iv.) with Fractiling under Hector environment (HFractiling); and (v.) with Hectiling (Hectiling).

All distributions were run on 4, 8, 16 and 32 processors while the system was exclusively used for these experiments, to exclude the effects of any external loads. The costs of runs using the "Uniform", "Gaussian", and "Corner" distributions for data size of

100k particles are shown in Figures 6-8. The costs of runs for data sizes 10K, 20K and 50 K particles are shown in Appendix – A. From these results, it can be seen that in almost all cases the costs of Fractiling, HFractiling, and Hectiling are lower than those of PFMA and HPFMA. When HFractiling is compared to Hectiling, it can be seen that the cost of Hectiling is in general lower. However, for 32 processors, the cost of Hectiling becomes higher than that of HFractiling.

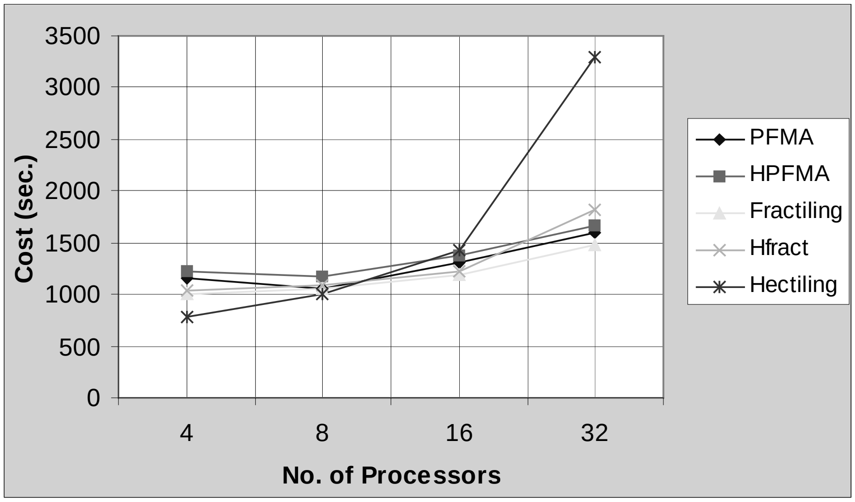

Figure 4.1 Cost for Uniform Distribution (100 K particle)

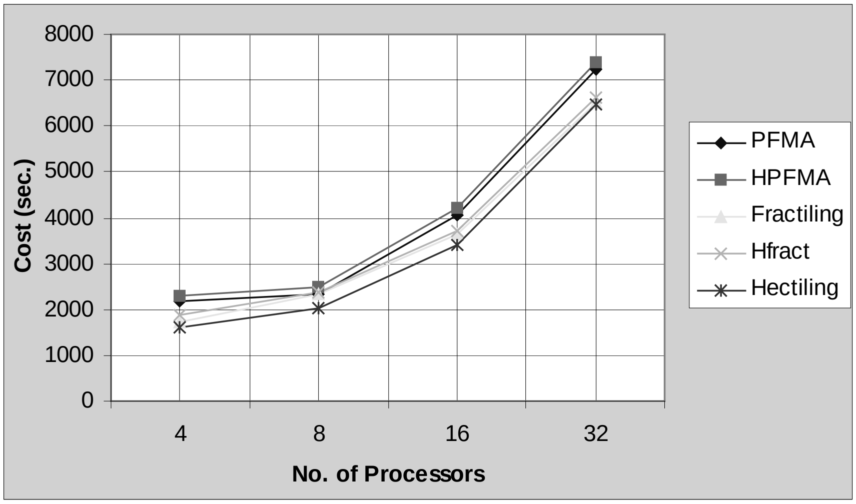

Figure 4.2 Cost for Gaussian Distribution (100 K particle)

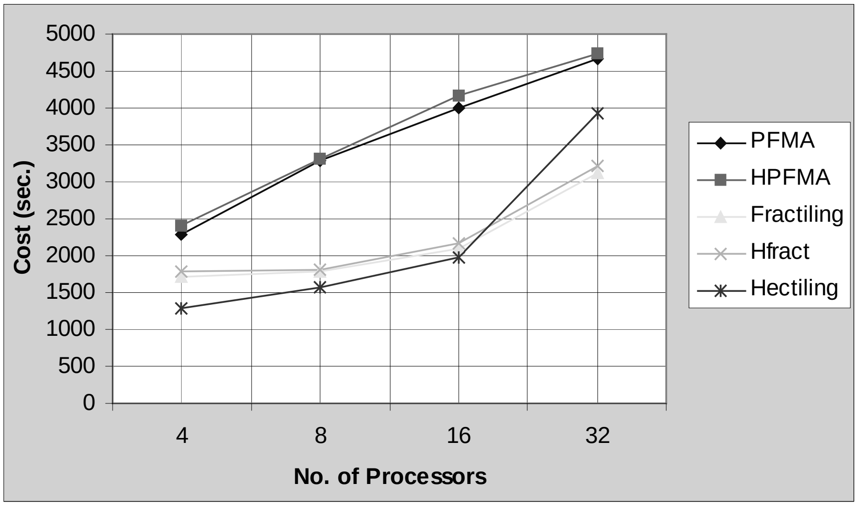

Figure 4.3 Cost for Corner Distribution (100 K particle)

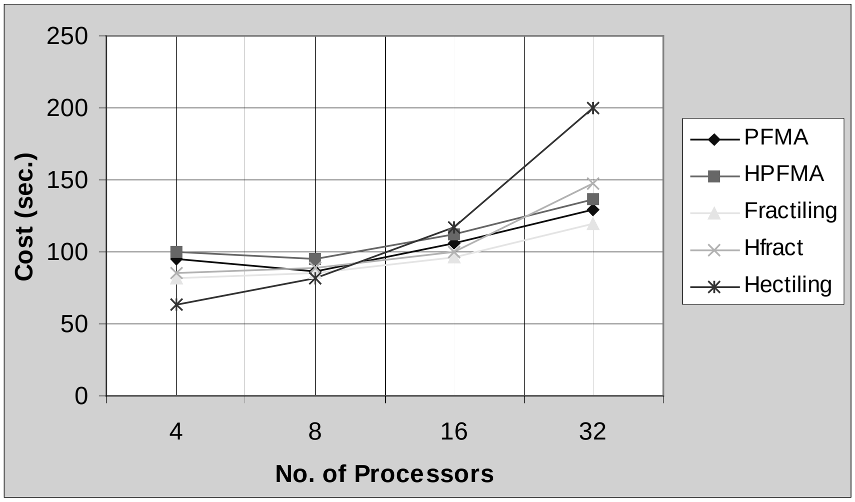

Figure 4.4 Cost for Uniform Distribution (10 K particle)

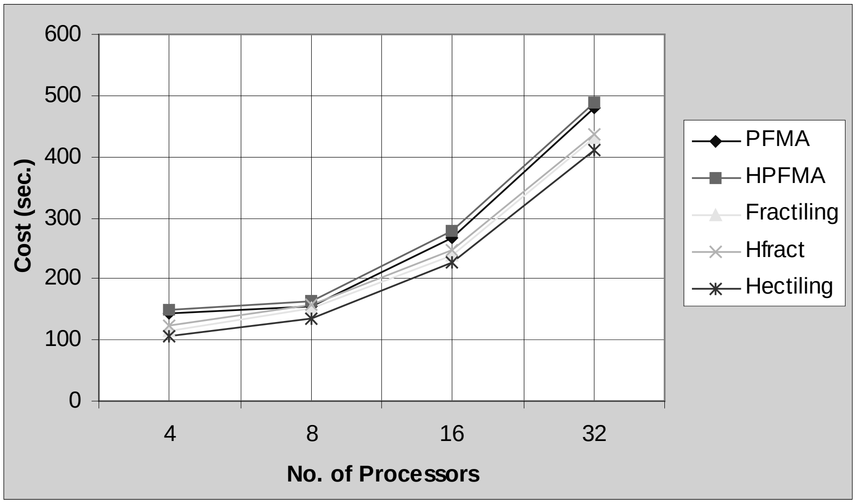

Figure 4.5 Cost for Gaussian Distribution (10 K particle)

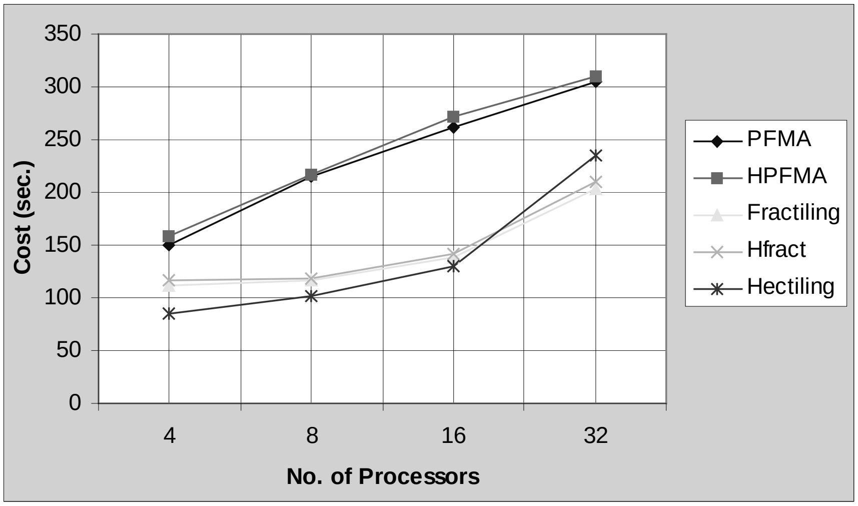

Figure 4.6 Cost for Corner Distribution (10 K particle)

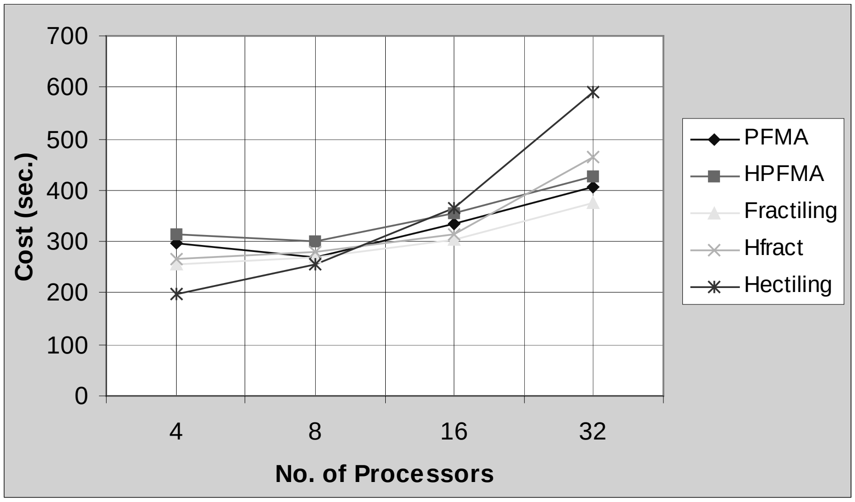

Figure 4.7 Cost for Uniform Distribution (20 K particle)

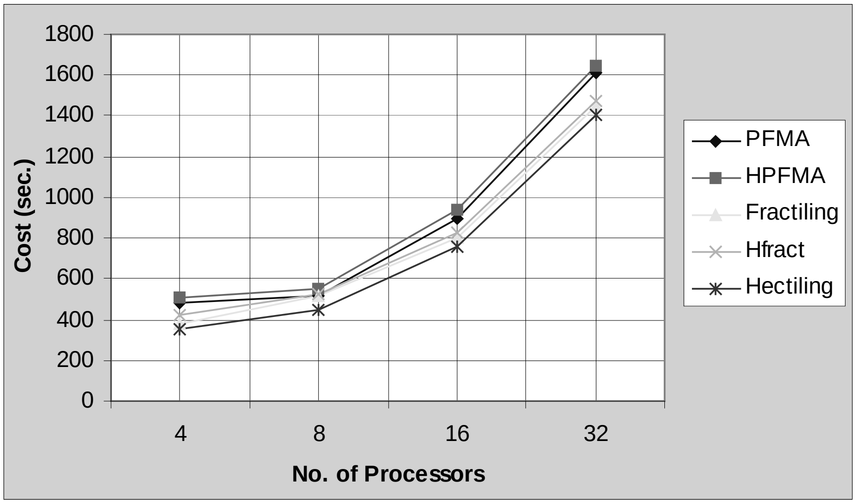

Figure 4.8 Cost for Gaussian Distribution (20 K particle)

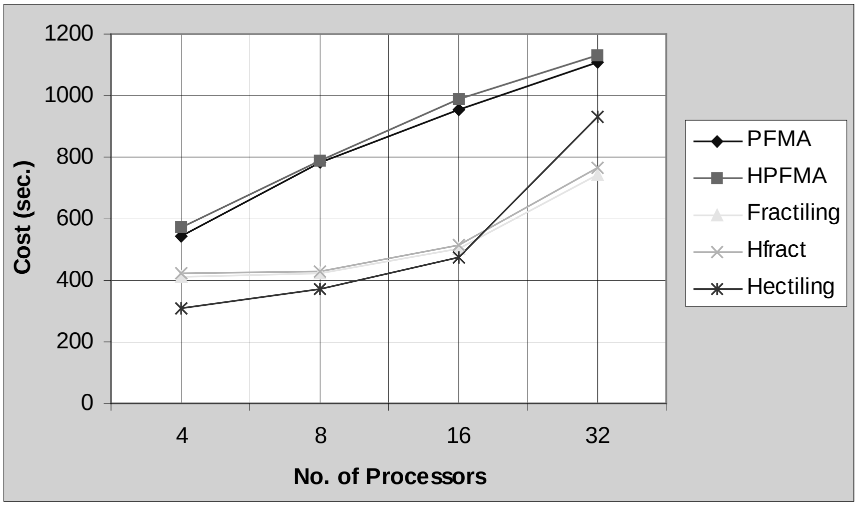

Figure 4.9 Cost for Corner Distribution (20 K particle)

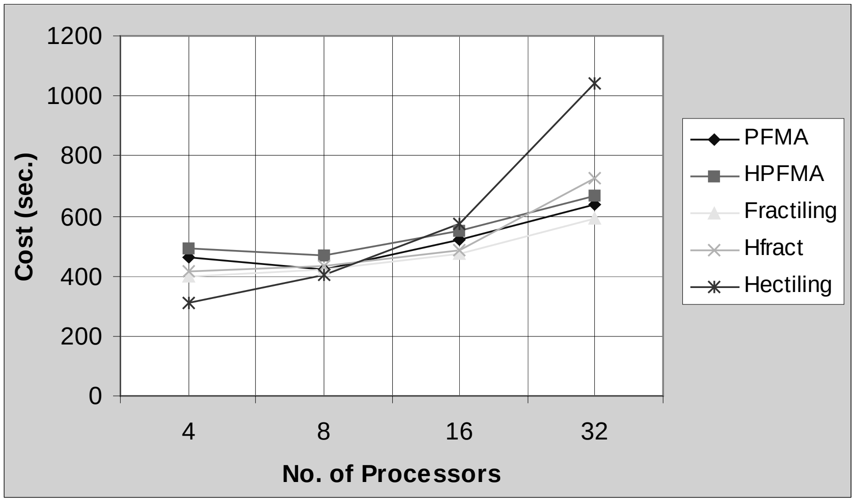

Figure 4.10 Cost for Uniform Distribution (50 K particle)

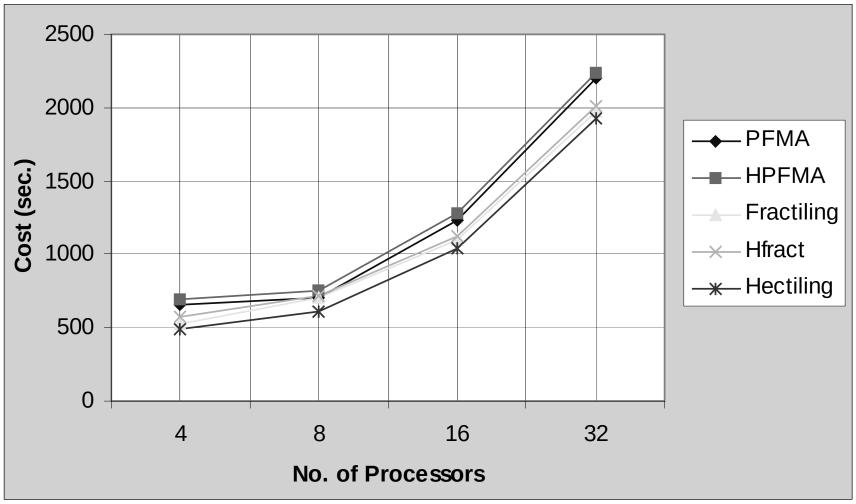

Figure 4.11 Cost for Gaussian Distribution (50 K particle)

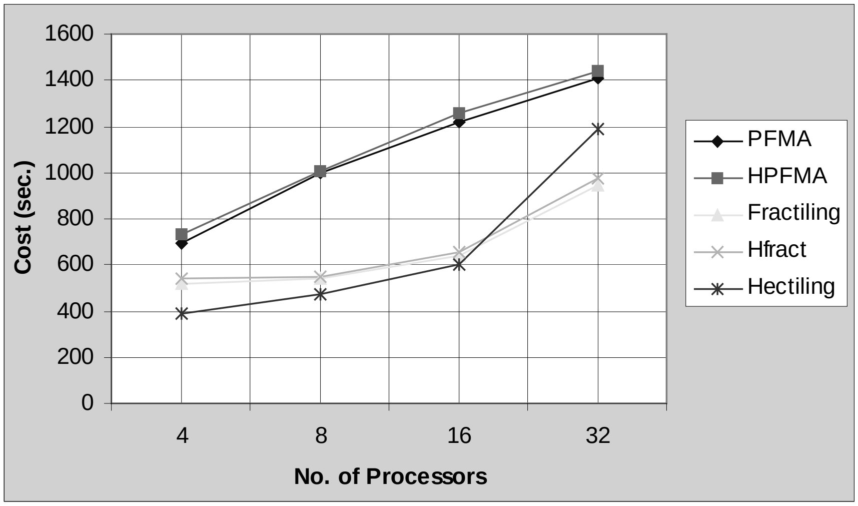

Figure 4.12 Cost for Corner Distribution (50 K particle)

The coefficients of variation (C.O.V.) of processors finishing times for data sizes 100K are shown in Figures 9-11. They are significantly lower for Hectiling, HFractiling and Fractiling when compared to PFMA and HPFMA. From the results presented in this section, it can be seen that the cost of Hectiling is slightly lower than those of HFractiling and Fractiling when a lower number of processors is used. However, when a higher number of processors is used, the cost of Hectiling is higher. The underlying communication structure and the nature of the Fractiling algorithm are responsible for these differences in costs. Hectiling uses UNIX sockets to implement this communication. The MA maintains a single socket for receiving Fractile_Ask and Hector update messages, whereas Fractiling routes Fractile_Ask messages directly from the Fractiling task to the Fractiling master by using the MPI infrastructure. Eventhough Hectiling adds an additional hop to the route taken by the Fractile_Ask messages, the socket implementation is faster. As a result, the overall cost of Hectiling is lower than that of HFractiling. However, as the number of processors increases, the number of Fractile_Ask messages also increases due to a larger number of Fractiling chunks. As the running application proceeds, the chunks sizes become smaller and require less time to complete. This translates into an increased communication overhead, due to an increase in frequency of Fractile_Ask messages. Therefore, at a higher number of processors, this creates a bottleneck in the MA and the cost of Hectiling increases disproportionately. This problem can be alleviated by two techniques, which could be simultaneously applied. One technique is to reduce the number of Fractiling chunks by increasing the minimum chunk size. The other is to create separate sockets, one for Fractile_Ask messages and another for Hector update messages.

Increasing the minimum chunk size would reduce the total number of Fractiling scheduled chunks. As a result, the number of Fractile_Ask messages would be reduced. However, with the increasing of the minimum chunk size, the probability of an increased load imbalance is higher.

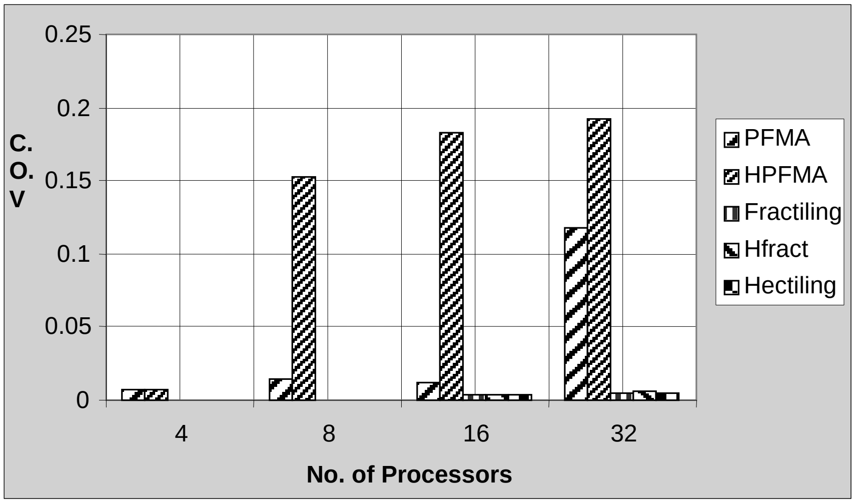

Figure 4.13 C.O.V for Uniform Distribution (100 K particle)

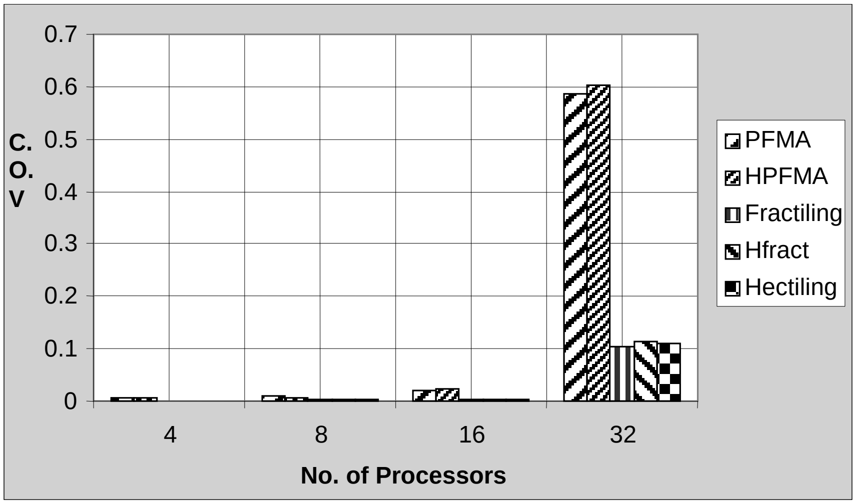

Figure 4.14 C.O.V for Gaussian Distribution (100 K particle)

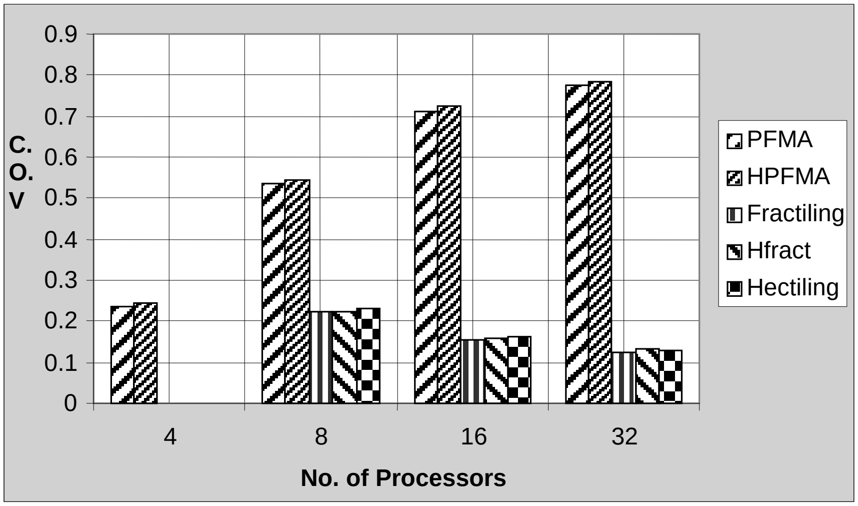

Figure 4.15 C.O.V for Corner Distribution (100 K particle)

A careful tuning of the minimum chunk size should reduce the impact of the increased communication overhead. Experiments using 32 processors for a uniform data distribution with various minimum chunk sizes were conducted. The experimental results show that increasing the minimum chunk size from one to two iteration units, increases the performance by 8% for HFractiling and 12% for Hectiling, while increasing the chunk size from one to four iteration units increases the performance by only 5% for HFractiling and 10% for Hectiling. With a minimum chunk size of one iteration unit versus two iteration units, the increase in communication overhead is larger than the gain obtained by load balancing. When the minimum chunk size is four iteration units versus two iteration units, the benefit of reducing the communication overhead is outweighed by the increase in load imbalance. Therefore, these experiments establish an optimal minimum chunk size of two iteration units for best performance. In general, optimal minimum chunk size may vary depending on the use of a specific architecture, application, data distribution, etc. These results support the theory on which Fractiling is based. In addition, these results show that the amount of performance improvement is larger for Hectiling than for HFractiling. More experiments using different minimum chunk sizes, data distributions, and problem sizes are required to determine the optimum chunk size for best performance.

The other technique for improving performance requires a separate dedicated socket for Fractile_Ask messages. Presently, the MA processes all messages it receives in order of their arrival. As a result, towards the end of the computation when the frequency of messages increases, Fractile_Ask messages stall at the MA before being forwarded to the Fractiling Master. To reduce the average stalling time the MA can use two separate sockets, one for the Fractile_Ask messages and another one for Hector update messages. Messages at the Fractile_Ask message socket should be given priority in such a way that the stalling time is reduced and that the Hector update messages do not suffer from starvation.

4.2 Hectiling with Migration

In this phase of testing five implementations of N-Body Simulations, using PFMA, HPFMA, Fractiling, HFractiling and Hectiling were studied. Since maximum of 32 processors were available and for task migration idle processors are required, experiments could not be executed on 32 processors. The experiments were executed on 2, 4, 8 and 16 processors. To determine the optimum chunk size, we conducted a limited number of experiments with all the distributions on 16 processors with minimum chunk sizes of one, two and four iteration units. The results show that the cost was least when the chunk size was two iteration units. As a result, a minimum chunk size of two iteration units was chosen for all the experiments in this phase. There were two sets of experiments in this phase. The first set of experiments was conducted with no external load. The costs of runs on all distributions without external load for data sizes 100K and 50K particles are shown in Figures 12-17. The second set of experiments was conducted with controlled external load to measure the performance of migration. A specially developed external application which takes about 50% of the processor cycles was launched on half the processors about 10 seconds after the execution started. The execution costs for all the distributions for data sizes 100k and 50K particles are shown in Figures 18-23.

From these figures it can be seen that when there is no external load, the cost of HFractiling is slightly higher than that of Fractiling, and the cost of Hectiling is always lower than that of Fractiling. The reason for this behaviour has been discussed in subsection 4.1.

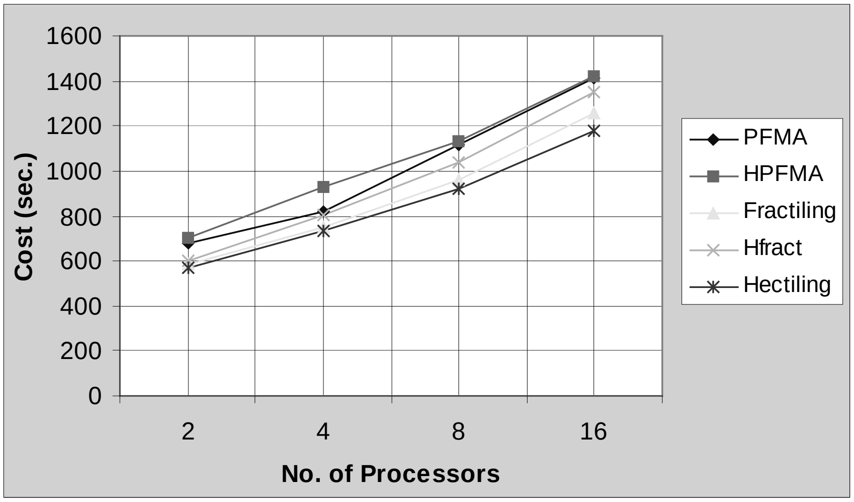

Figure 4.16 Cost for Uniform Distribution without Load (100 K particle)

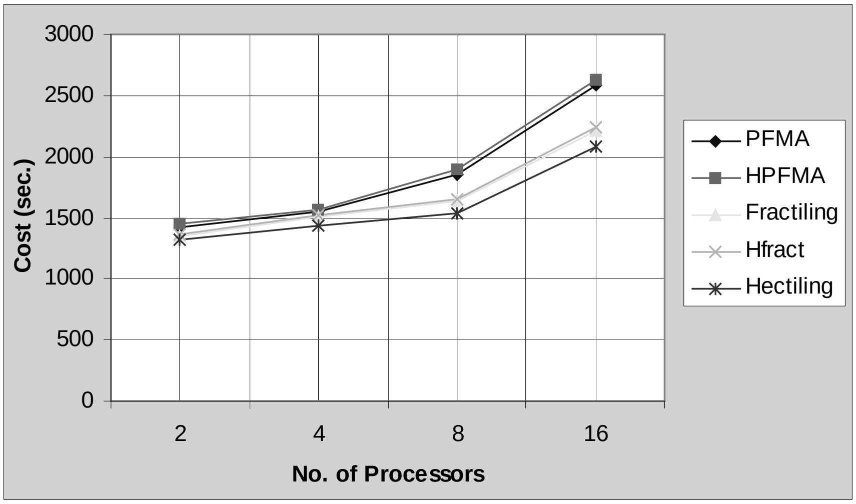

Figure 4.17 Cost for Gaussian Distribution without Load(100 K particle)

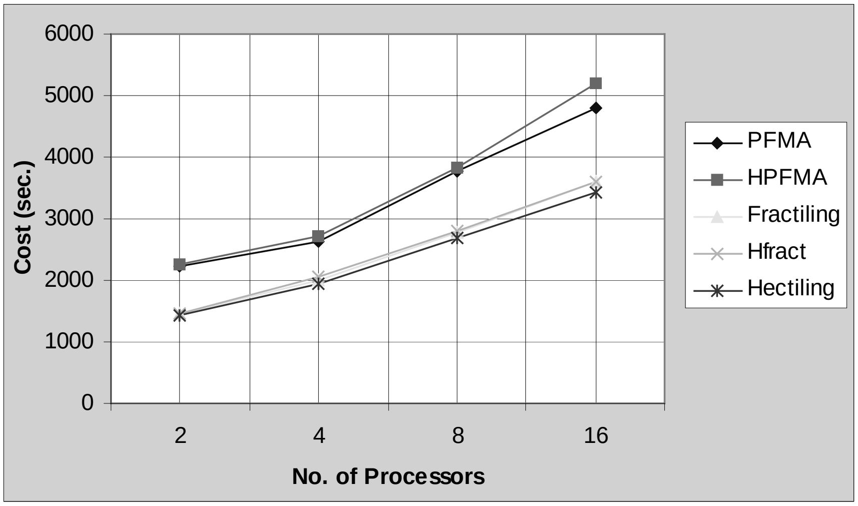

Figure 4.18 Cost for Corner Distribution without Load (100 K particle)

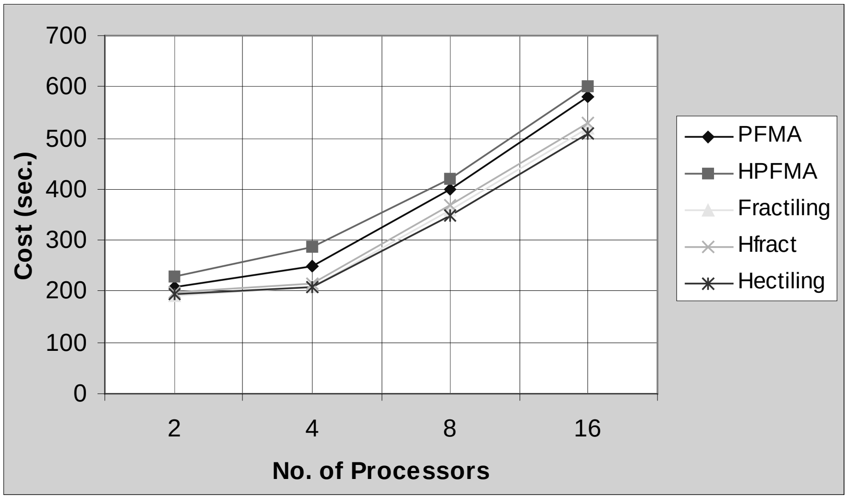

Figure 4.19 Cost for Uniform Distribution without Load (50 K particle)

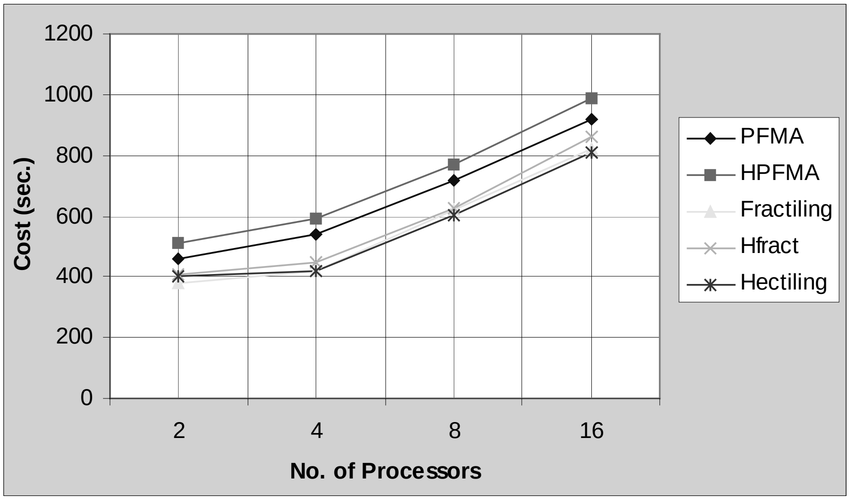

Figure 4.20 Cost for Gaussian Distribution without Load (50 K particle)

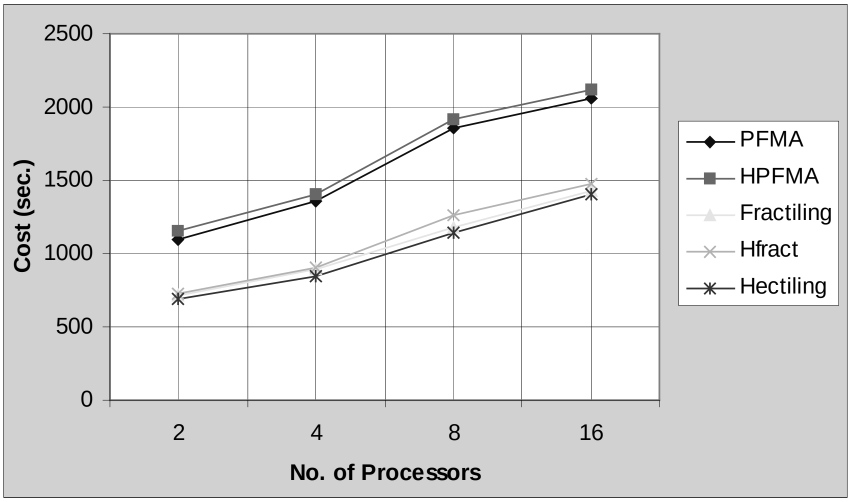

Figure 4.21 Cost for Corner Distribution without Load (50 K particle)

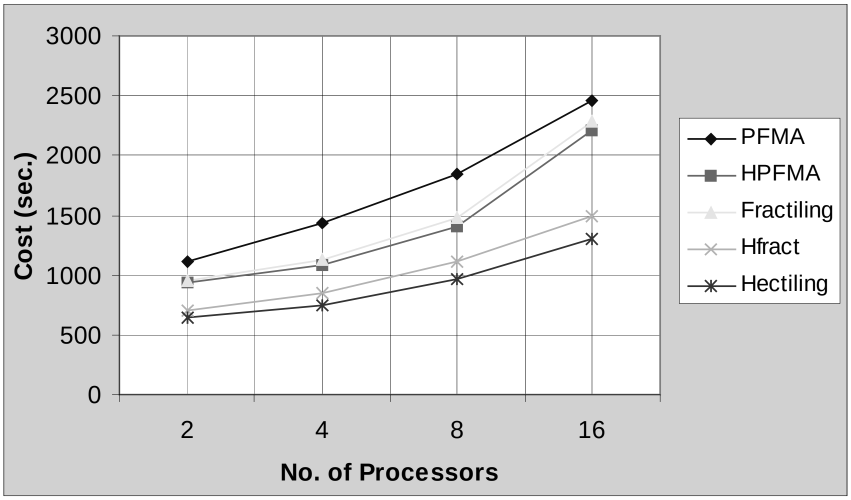

Figure 4.22 Cost for Uniform Distribution with Load (100 K particle)

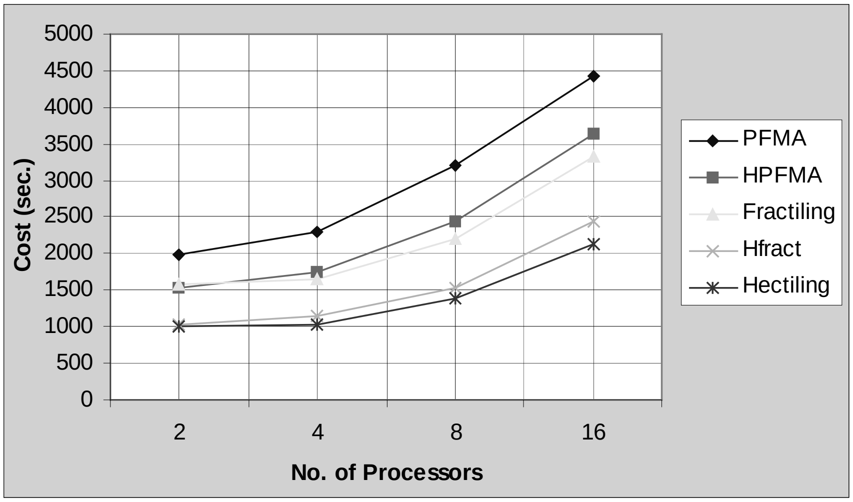

Figure 4.23 Cost for Gaussian Distribution with Load(100 K particle)

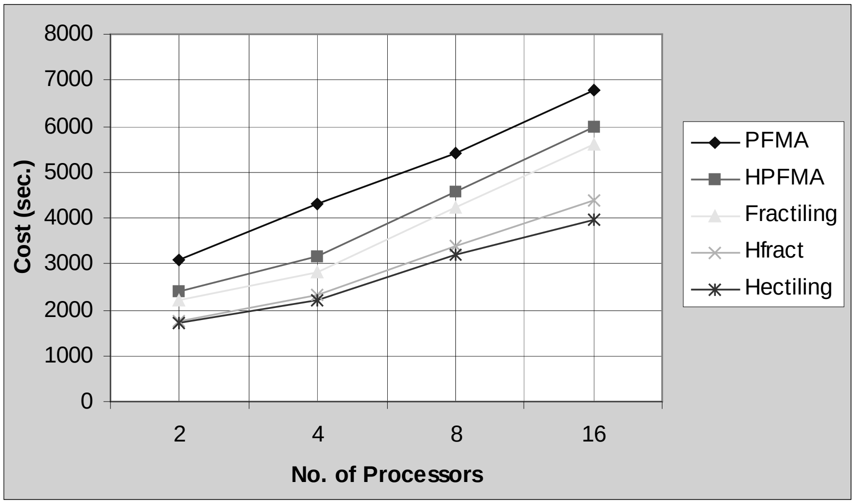

Figure 4.24 Cost for Corner Distribution with Load (100 K particle)

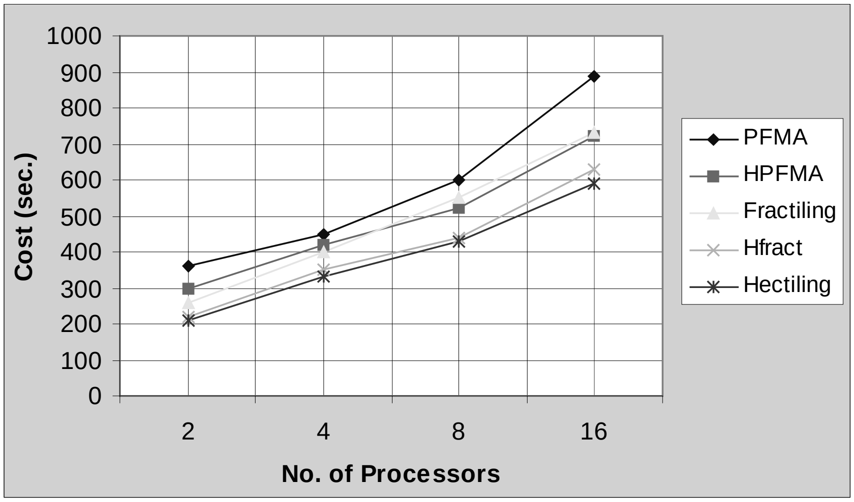

Figure 4.25 Cost for Uniform Distribution with Load (50 K particle)

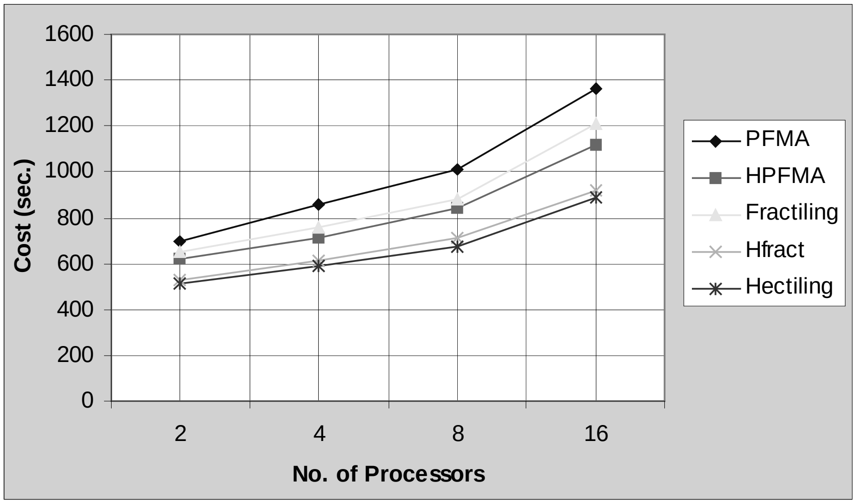

Figure 4.26 Cost for Gaussian Distribution with Load (50 K particle)

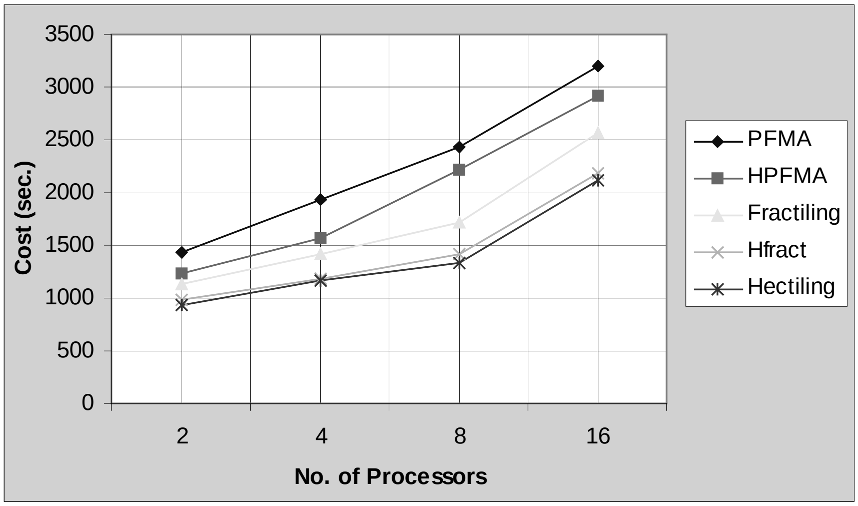

Figure 4.27 Cost for Corner Distribution with Load (50 K particle)

However, when there is external load, the cost of Fractiling is found to be always higher than that of HFractiling or Hectiling, and is also found to be considerably higher than that of Fractiling with no external load. This can be attributed to the external load, which takes away CPU cycles, resulting in an increase of Fractiling cost. In the case of HFractiling or Hectiling, the external load causes the process to migrate to an idle processor where it can use the CPU exclusively. As a result, the introduction of an external load does not result in a cost increase. Due to migration overhead, the costs of HFractiling and Hectiling with external loads are slightly higher than those of Fractiling with no external loads. The results show that because of its capability to migrate tasks from busy workstations to idle ones, Hectiling performs much better than Fractiling when external workloads are present. The results also show that Hectiling performs better than HFractiling. In addition, under no load conditions, Hectiling slightly outperforms both Fractiling and HFractiling, which indicates that the overhead of Hectiling is lower than that of Fractiling and HFractiling. The coefficients of variation (C.O.V.) of processors finishing times for data sizes 100K are shown in Figures 24-26. They are similar for Hectiling, HFractiling and Fractiling, and significantly lower when compared to PFMA and HPFMA. The C.O.V.s of PFMA and HPFMA are 6 to 2000 times larger than those of Hectiling

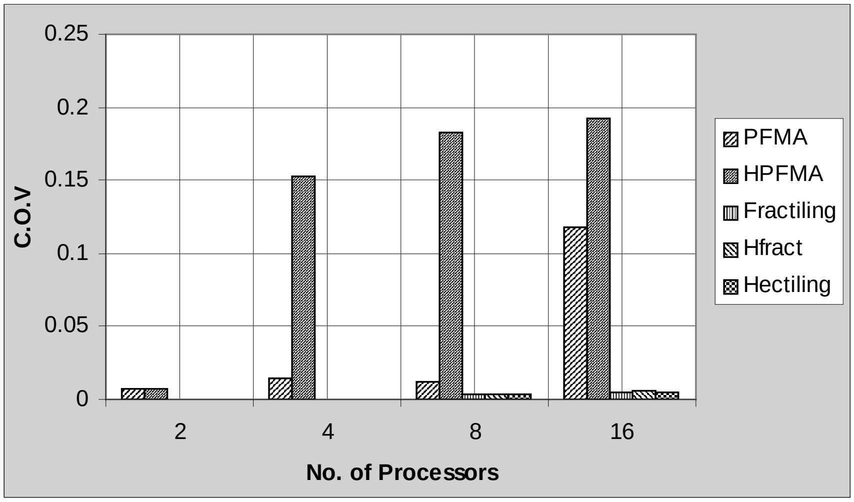

Figure 4.28 C.O.V for Uniform Distribution (100 K particle)

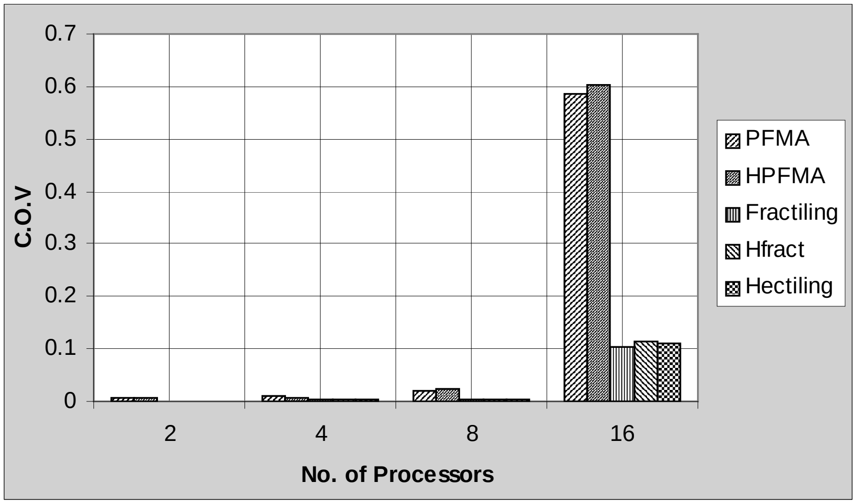

Figure 4.29 C.O.V for Gaussian Distribution (100 K particle)

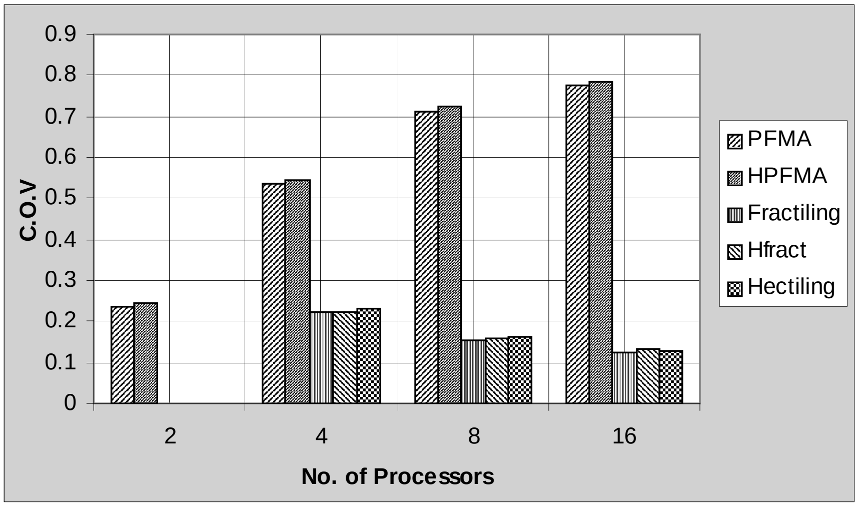

Figure 4.30 C.O.V for Corner Distribution (100 K particle)

4.3 Analysis

Figure 27-29 show the percentage of improvement of Hectiling in cost over HPFMA, Fractiling and HFractiling without load for all Distribution for data size100K. Figure 30-32 shows the percentage of improvement with load for data sizes 100K. From these result it can be seen that Hectiling always achive better performance than HPFMA,Fractiling or Hfractiling. In general as number of processor increases for a particular data size the percentage improvement also increases slighly. This is because as the number of processor increases the load imbalance also increases and Hectiling does a better load blanacing than HPFMA, Fractiling or Hfractiling. More over the percentage of improvement over Fractiling with load is more than that of without load. That is because Hectiling migrates tasks from nodes with exaternal load to idle nodes, which Fractiling cannot do.

Table 1-3 shows speed up for all distributions and data sizes without external load. The speed up is similar for Hectiling, Hfractiling and Fractiling. The speed up increases as the number of processors increases. This indicates that all these methods scale well as the number of processor increases. Moreover, for particular number of processor as the problem size increases the speed up increases, which indicates that Hectiling, Hfractiling and Hectiling scale well as the problem size increases.

For every method we have conducted 48 experiments (12 data sets on 4 different number of processors) in the first phase, and 96 experiments (48 without external load and 48 with external load) in the second phase. Out of 144 experiments only in 9 experiments Hectiling performs worse than Fractiling and in all cases Hectiling performs better than HPFMA (PFMA under Hector). In experiments where external load is used (48 experiments), Hectiling always performed better than all other methods. Since in normal operating environment in network of workstations it is reasonable to assume that external loads will be present, the experimental results underscore the importance of running scientific applications using Hectiling.

In all experiments of up to sixteen processors Hectiling always performed better than Fractiling or HPFMA. In the first phase of experimentation, in eight experiments out of forty eight experiments, Hectiling performed worse than Fractiling or HPFMA; these results occurred when the experiments were conducted on thirty two processors. There are two explanations for these behaviors. First, task migration, one of the major components of Hectiling could not be activated while running experiments on thirty two processors because a maximum of thirty two processors were available, and there were no idle processors available for task migration. The second explanation is that the problem sizes were not big enough to get a performance improvement. More experimentation would be conducted in the future on higher number of processors and larger problem sizes.

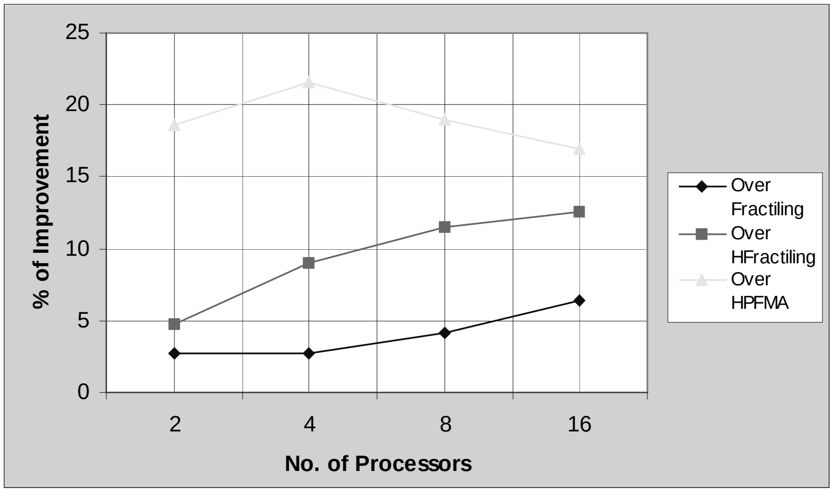

Figure 4.31 Hectiling Cost Improvement for Uniform Distribution without Load (100 K particles)

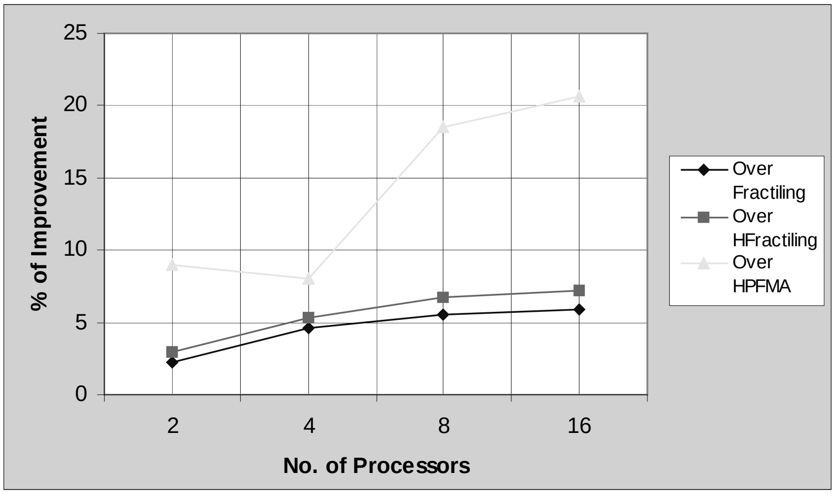

Figure 4.32 Hectiling Cost Improvement for Gaussian Distribution without Load(100 K particles)

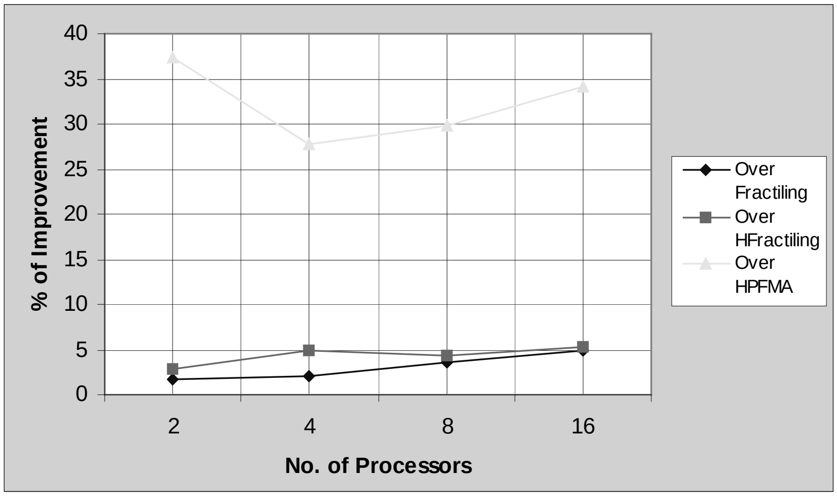

Figure 4.33 Hectiling Cost Improvement for Corner Distribution without Load (100 K particles)

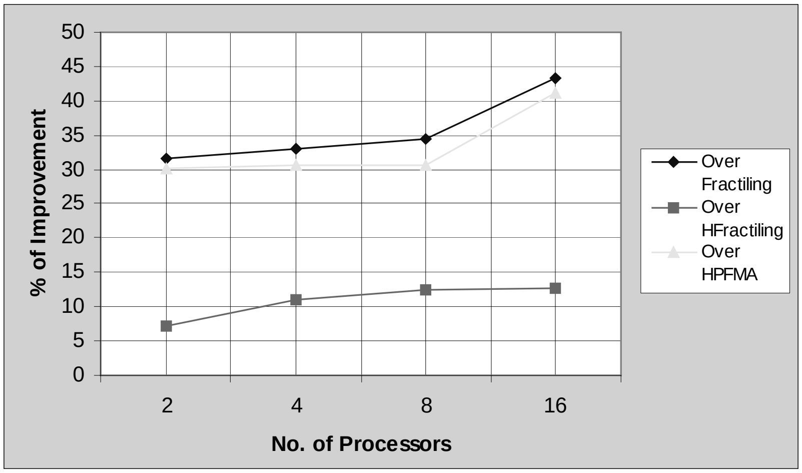

Figure 4.34 Hectiling Cost Improvement for Uniform Distribution with Load (100 K particles)

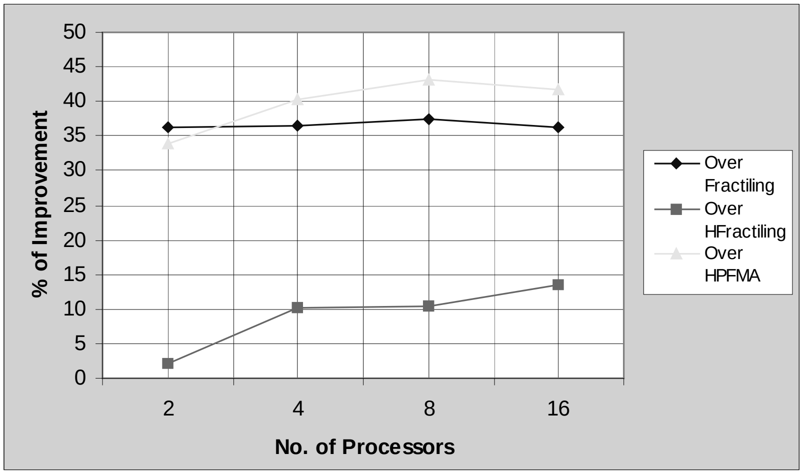

Figure 4.35 Hectiling Cost Improvement for Gaussian Distribution with Load(100 K particles)

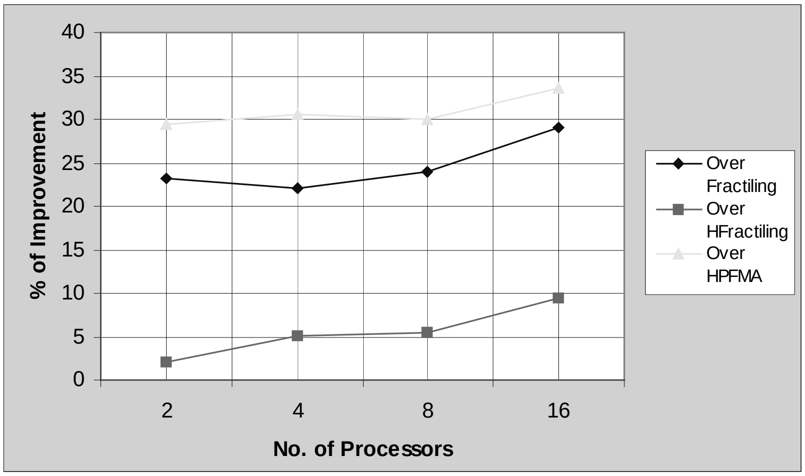

Figure 4.36 Hectiling Cost Improvement for Corner Distribution with Load (100 K particles)

Table 4.1 Speedup for Uniform Distribution

Problem Size (Particles)	**# Processors**	**2**	**4**	**8**	**16**
	Method				
10 K	Hectiling	1.84	3.21	5.67	6.89
	Hfractiling	1.82	3.14	5.41	6.65
	Fractiling	1.78	2.99	5.01	6.09
20 K	Hectiling	1.89	3.55	6.02	9.76
	Hfractiling	1.86	3.48	5.96	9.44
	Fractiling	1.81	3.25	5.76	8.90
50 K	Hectiling	1.91	3.76	6.97	10.79
	Hfractiling	1.89	3.67	6.88	10.67
	Fractiling	1.86	3.54	6.55	10.41
100 K	Hectiling	1.94	3.92	6.89	12.52
	Hfractiling	1.93	3.83	6.78	12.34
	Fractiling	1.91	3.64	6.76	12.02

Table 4.2 Speedup for Gaussian Distribution

Problem Size (Particles)	**# Processors**	**2**	**4**	**8**	**16**
	Method				
10 K	Hectiling	1.73	2.88	4.80	5.98
	Hfractiling	1.64	2.73	4.61	5.78
	Fractiling	1.66	2.76	4.62	5.81
20 K	Hectiling	1.72	2.79	5.12	7.45
	Hfractiling	1.63	2.71	4.95	7.18
	Fractiling	1.67	2.73	4.99	7.21
50 K	Hectiling	1.92	3.61	6.28	8.28
	Hfractiling	1.81	3.38	6.02	8.02
	Fractiling	1.84	3.41	6.06	8.05
100 K	Hectiling	1.73	3.21	6.02	8.03
	Hfractiling	1.81	3.30	6.11	8.17
	Fractiling	1.82	3.31	6.13	8.18

Table 4.3 Speedup for Corner Distribution

Problem Size (Particles)	**# Processors**	**2**	**4**	**8**	**16**
	Method				
10 K	Hectiling	1.79	2.87	4.88	6.87
	Hfractiling	1.72	2.49	4.67	6.53
	Fractiling	1.75	2.51	4.68	6.55
20 K	Hectiling	1.82	2.94	5.08	8.32
	Hfractiling	1.95	2.48	4.81	7.97
	Fractiling	1.93	2.52	4.84	8.00
50 K	Hectiling	1.94	2.99	5.57	9.58
	Hfractiling	1.88	2.84	5.45	9.22
	Fractiling	1.90	2.86	5.44	9.27
100 K	Hectiling	1.93	2.89	5.65	9.88
	Hfractiling	1.90	2.73	5.22	9.47
	Fractiling	1.91	2.72	5.25	9.49

The implementation of Hectiling and succecsfull run of experiments on different data sizes and processors validates the first part of the hypothesis, which state that: “The integration of an algorithmic load balancing strategy (Fractiling) with a systemic load balancing strategy (Hector) is possible.”

When no external load is present in 92% (88 out of 96) experiments, Hectiling performs better than all other techniques. If we consider all the experiments in 94% (136 out of 144) experiments, Hectiling performs better than Fractiling and in all case it performs better than HPFMA. In experiments with external load Hectiling always performs better than Fractiling or HPFMA. From these experiments it can be said that the following inequality has been proven for all cases up to sixteen processors and in 92% cases up to thirty-two processors.

$C_{\text{Hectiling}} \leq \text{Min} (C_{\text{Fractiling}}, C_{\text{HPFMA}})$

Where:

$C_{\text{Hectiling} =}$ Parallel execution cost in Hectiling

$C_{\text{Fractiling} =}$ Parallel execution cost in Fractiling

$C_{\text{HPFMA} =}$ Parallel execution cost in Hector

Hence the second part of the hypothesis has also been proven.

CHAPTER V

CONCLUSION AND FUTURE WORK

Load balancing improves the efficient use of resources and therefore the performance of parallel and distributed applications. Over time, systemic techniques have improved the performance of runtime systems at coarse-grained levels, while algorithmic techniques have improved the performance of applications at fine-grained levels. Combining strategies from both levels of granularity can result in methods, which deliver advantages of both. This thesis describes lessons learned from the successes and limitations of Hectiling, a system that combines an algorithmic strategy for data-parallel load balancing with a systemic strategy for task-parallel load balancing. In addition, avenues for performance enhancement are explored.

Earlier experiments with algorithmic and systemic load balancing strategies showed their ability to improve performance. A systemic coarse-grained load balancing was supported in Hector by monitoring and re-balancing loads via task migration. Algorithmic, fine-grained load balancing was supported using Fractiling by a dynamic redistribution of data assignments among tasks.

After realizing that Fractiling could benefit by accessing the run-time information gathered by Hector, it was decided to develop an interface between them. The integrated system was tested in order to measure the overhead of passing state-update messages through Hector's Master Allocator. The performance of the integrated version was better than that of Fractiling alone or Fractiling under Hector, in the presence of external load as well as in its absence. This performance improvement is due to the fact that the overhead of Hectiling is considerably low while allowing dynamic process migration.

For larger number of processors, the Hectiling cost could be reduced in a few ways. One way to improve performance is through tuning of the minimum chunk size. Experiments with different minimum chunk sizes show that performance improvements can be obtained simply by tuning of the Fractiling scheme. In addition, redesigning the Master Allocator with multiple sockets may overcome the performance bottlenecks.

The integrated system was tested for N-body simulations. N-body simulations have been widely used in a broad class of application areas of science such as astrophysics, molecular dynamics, biophysics, molecular chemistry etc. Hectiling will improve performance of any application that employs N-body simulations in a distributed computing environment. Parallel N-body simulations are a data parallel application. It is also reasonable to assume for this data parallel application, Hectiling will perform better than applying Fractiling or Hector independently.

Extensions to both Hector and Fractiling may also prove fruitful. For example, support for a distributed shared memory environment would enable thread-migration-based load balancing, and the combination of Hector and Fractiling would then support the three ways that computational load can be redistributed (task, data, and thread migration). In addition, enhancements to Fractiling that are currently being pursued, may in turn improve the functionality of the resulting integrated system.

In cases where low-overhead measurements of performance can be made, some improvements in Fractiling performance are possible. For example, measurements of nearness to completion and of relative performance can allow the amount of data exchange to be proportional to the actual performance. In general, the measurements required are less expensive than the ones used in profiling, and can be immediately used, instead of waiting until a subtile execution is completed. An advantage of the integration of Fractiling and Hector into a single framework is that it specifically facilitates this performance improvement. Since the MA periodically gathers information from the SAs about the tasks running under them, the nearness to completion of subtiles can be collected and forwarded to the Fractiling Master without any extra overhead. This enables the Fractiling Master to transfer data from a slow Fractile Task to a Fractiling Task, which is about to finish. As a result, the Fractiling Tasks would not run out of data, and thus would not have to request the Fractiling Master to transfer data. This results in minimizing communication and better resource utilization. Another advantage of this integrated design is the re-routing of the Fractile_Ask message via the MA. Since the re-routing is implemented using sockets, it is faster than a direct MPI based communication between Fractiling Master and Fractiling Tasks. In general, the MPI communications use lower level communication primitives (i.e., sockets), which involve at least one extra level of interface. A third advantage of this integrated design is that the controlling and the decision making component of the Fractiling Master could be moved as a module inside the MA, and this would reduce some of the communication overhead.

Hectiling can also be implemented on heterogeneous platforms. In such cases, Hectiling migrates tasks between pairs of homogeneous workstations, as for example, between pairs of Sun workstations, or pairs of SGI workstations, as opposed to between Sun and SGI workstations. The migration cost between two Sun SPARCstations connected by 10 Mbits/sec Ethernet was observed to be 0.6 Mbytes/sec[18]. If the workstations are connected by various bandwidth interconnection networks, the migration cost between different pairs of workstations will vary. In Hectiling, network information, such as bandwidth, latency, and congestion of interconnects, is presently not taken into account when making migration decisions. This may lead to reduced performance in some situations where, for instance, a very large task is migrated between workstations connected by a very slow connection. For such cases, the cost of migration may be higher than the increase in cost of running the task on the busy workstation. Further work to improve Hectiling can be pursued by incorporating network information into task migration decisions.

The Hectiling paradigm can be generalized with little effort, to be applied to any scientific application that is data parallel. Even more, any algorithmic load balancing technique that works around a master slave strategy could be integrated into Hector with minor modifications. By careful planning and design, it is possible to develop a set of well-defined Hectiling APIs, which, in turn, can be used by scientific applications to incorporate Hectiling.

REFERENCES

1. C. R. Anderson, An Implementation of the Fast Multipole Method SIAM J. Sci. Stat. Comput.,1992, 923-947.

2. M. Baker and G. Fox and H. Yau. Cluster Computing Review, Northeast Parallel Architecture Center, Syracuse www.npac.syr.edu/techreports/hypertext/sccs-0748/cluster-review.html, 1995.

3. I. Banicescu. Load Balancing and Data Locality in the Parallelization of the Fast Multipole Algorithm, Ph.D. Dissertation, Polytechnic University, 1996 January.

4. I. Banicescu and S. F. Hummel. Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations, Proceedings of Supercomputing'95 conference,1995 (on CD-ROM).

5. I. Banicescu and S. F. Hummel. Balancing Processor Loads and Exploiting Data Locality in Irregular Computations, IBM Research Report, 1995, RC19934.

6. I. Banicescu and R. Lu. Experiences with Fractiling in N-Body Simulations, Proceedings of High Performance Computing'98 Symposium, 121--126, 1998.

7. I. Banicescu and S. Russ and M. Bilderback and S. Ghafoor. Competitive Resource Management in Distributed Computing Environment with Hectiling Proceedings of High Performance Computing'99 Symposium, 337-343, 1999.

8. I. Banicescu and S. Ghafoor and M. Bilderback. Efficient Resource Management for Scientific Applications in Distributed Computing Environment, Proceedings of the Workshop on Distributed Computing on the Web (DCW'98), 45--54,1998.

9. J. A. Board and J. Causey and J. F. Leathrum Jr. and Accelerated Molecular Dynamic Simulations with the Parallel Fast Multipole Algorithm, Chemical Physics Letters, 1992, 198, 23-34.

10. J. A. Board and Z. S. Hakura and W. D. Elliot and others. Scalable Variants of Multipole-based Algorithms for Molecular Dynamics Applications, In the Proceeding of Seventh SIAM Conference on Parallel Processing for Scientific Computing, 1995, SIAM, Philadelphia, 295--300, February.

11. N. Carriero and E. Freeman and D. Gelernter and D. Kaminsky. Adaptive Parallelism and Piranha, Computer, 28, 1, 40-49, 1995.

12. J. Casas and D. Clark and R. Konuru and S. W. Otto MPVM: A Migration Transparent Version of PVM, Usenix Computing Systems Journal, 171--216, 8, 2,1995.

13. DQS User Manual - DQS Version 3.1.2.3Supercomputer Computations Research Institute, Florida State University, 1995.

14. D.G. Feitelson, L. Rudolph, U. Schwiegelshohn,K.C. Sevcik and P. Wong". Theory and Practice in Parallel Job Scheduling. IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing, 1997.

15. I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, 1998.

16. Al Geist, Adam Beguelin, Jack Dongara, Weiching Jiang, Robert Manchek, and Vaidy Sundaram. PVM: Parallel Virtual Machine. MIT Press, Cambridge, 1994.

17. R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers, IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing, 1997.

18. Y. Grama, V. Kumar and A. Sameh. Scalable Parallel Formulations of Barnes-Hut Method for N-Body Simulations, Proc. of Supercomputing'94, 439--448, November, 1994 .

19. L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulation, Journal of Computational Physics, 1987, May, 325--48, 73.

20. Willium Gropp, Edwing Lusk and Anthony Skjellum. Using MPI. MIT Press, Cambridhge 1994.

21. S. F. Hummel. Fractiling: A Method for Scheduling Parallel Loops on NUMA Machines, IBM RC18958, 1993.

22. S. F. Hummel and E. Schonberg and L. E. Flynn, A Practical and Robust Method for Scheduling Parallel Loops, Communications of the ACM, 1992, 358, August, 90—101.

23. M.T. Jones and P.E. Plassman. Parallel Algorithms for Adaptive Mesh Refinement, SIAM Journal on Scientific Computing", Vol.18, pp 686-708, 1997.

24. J. F. Leathrum and J. A. Board, The Parallel Fast Multipole Algorithm in Three Dimensions, Duke University, Department of Electrical Engineering, 1992, TR92-001, April.

25. H. Li, S. Tandri, M. Stumm and K. C. Sevcik. Locality and Loop Scheduling on NUMA Machines, Proceedings of Int. Conf. on Parallel Processing, pp II140-II147, 1993.

26. LSF. Product Reviews: Platform Computing Corp. Load SunExpert, 8, 8, 62--64, 1997

27. R. Lu, Parallelization of the Fast Multipole Algorithm with Fractiling in Distributed Memory Architectures, Mississippi State University, 1997.

28. E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 4, pp 379-400, 1992.

29. B. Neuman and S. Rao, The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed System, Concurrency: Practice and Experience, 339--355, 1994.

30. T.D. Nguyen, R. Vaswani and J. Zahorjan. Using Run-Time Measured Workload Characteristics in Parallel Processing Scheduling, IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing, 1996.

31. C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers, IEEE Transactions on Computers, Vol. C-36, No. 12, pp1425-1439, 1987.

32. J. Pruyne and M. Livney. Providing Resource Management Services to Parallel Applications, Workshop on Job Scheduling Strategies for Parallel Processing, Proceedings of the International Parallel Processing Symposium (IPPS 1995),1995.

33. J. Robinson and S. Russ and B. Flachs and B. Heckel. A Task Migration Implementation of the Message-Passing Interface, 5th High Performance Distributed Computing Conference (HPDC-5), 61--68, 1996.
34. S. Russ and B. Flachs and J. Robinson and B. Heckel. Hector: Automated Task Allocation for MPI, 10th International Parallel Processing Symposium, 344--348,1996.

35. S. Russ and M. Gleeson and B. Meyers and L. Rajagopalan and C. Tan, Using Hector to run MPI Programs over Networked Workstation, Concurrency: Practice and Experience, Accepted for publication.
36. S. Russ and B. Meyers and M. Gleeson and J. Robinson and L. Rajagopalan and C. Tan and B. Heckel. User Transparent Run-Time Performance Optimization, The 2nd International Workshop on Embedded HPC and Applications at the 11th IEEE International Parallel Processing Symposium, 1997.

37. S. H. Russ, K. Reece, J. Robinson, B. Meyers, L. Rajagopalan and C.-H. Tan. An Agent Based Architecture for Dynamic Resource Management, IEEE Concurrency, Vol. 7, No. 2, pp 47-55, 1999.

38. J. Salmon and M. S. Warren, Parallel, Out-of-core Methods for N-Body Simulation, Proceeding of 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997, SIAM.

39. J. Singh, Parallel Hierarchical N-Body Methods and Their Implications for Multiprocessors, Stanford University, 1993 .

40. J. Singh and C. Holt and T. Totsuka and others, A Parallel Adaptive Fast Multipole Algorithm, Proc. of Supercomputing'93, 54--65, 1993.

41. A. Sohn and R. Biswas and H. Simon. Dynamic Load Balancing Framework for Unstructured Adaptive Computations on Distributed-Memory Multiprocessors, Proceedings of Symposium on Parallel Algorithms and Architectures, 189-192, 1997.

42. T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System, Dr. Dobbs' Journal of Software Tools for 20, 2, 40--48, 1995.

43. T. H. Tzen and L. M. Ni. Dynamic Loop Scheduling for Shared-Memory Multiprocessors, Proc. Int. Conf. on Parallel Processing, II, 247-250, 1991.

44. M. Warren and J. Salmon. Astrophysical N-Body Simulation Using Hierarchical Tree Structures, Proc. of Supercomputing'92, 1992.

45. M. Warren and J. Salmon. A Parallel Hashed Oct Tree N-Body Algorithm, Proceeding of Supercomputing'93, 1993, 12--21, IEEE Computer Society.

46. S. Zhou. LSF: Load Sharing and Batch Queueing Software, Platform Computing Corporation, 1996, North York, Canada.