Experiences from Integrating Algorithmic and Systemic Load Balancing Strategies

A Manuscript Submitted for Consideration to Concurrency: Practice and Experience

Ioana Banicescu, Samuel H. Russ, Sheikh Ghafoor, Vijay Velusamy and Mark Bilderback
E-mail: {ioana, russ, ghafoor, vijay, bilderba}@erc.msstate.edu
Mississippi State University
Department of Computer Science
NSF Engineering Research Center for Computational Field Simulation

Abstract – Load balancing increases the efficient use of existing resources for parallel and distributed applications. At a coarse level of granularity, advances in runtime systems for parallel programs have been proposed in order to control available resources as efficiently as possible by utilizing idle resources and using task migration. Simultaneously, at a finer granularity level, advances in algorithmic strategies for dynamically balancing computational loads by data redistribution have been proposed in order to respond to variations in processor performance during the execution of a given parallel application. Combining strategies from each level of granularity can result in a system which delivers advantages of both. The resulting integration is systemic in nature and transfers the responsibility of efficient resource utilization from the application programmer to the runtime system. This paper presents the design and implementation of a system that combines an algorithmic fine-grained data parallel load balancing strategy with a systemic coarse-grained task-parallel load balancing strategy, and reports on recent experimental results of running a computationally intensive scientific application under this integrated system. The experimental results indicate that a distributed runtime environment which combines both task and data migration can provide performance advantages with little overhead. It also presents proposals for performance enhancements of the implementation, as well as future exploration for effective resource management.

Point of Contact:

Name:	Dr. Ioana Banicescu
Phone:	(601) 325-7508
Fax:	(601) 325-8997
email:	ioana@cs.msstate.edu
Web:	http://www.cs.msstate.edu/ ioana
Mail:	Mississippi State University Department of Computer Science Mail Stop 9637 Mississippi State, MS 39762

Experiences from Integrating Algorithmic and Systemic Load Balancing Strategies*

Ioana Banicescu, Samuel H. Russ, Sheikh Ghafoor, Vijay Velusamy and Mark Bilderback
E-mail: {ioana, russ, ghafoor, vijay, bilderba}@erc.msstate.edu
Mississippi State University
Department of Computer Science
NSF Engineering Research Center for Computational Field Simulation

Keywords: Load Balancing; Dynamic Scheduling; Resource Management; Parallel Applications; Distributed Runtime Environment

Abstract – Load balancing increases the efficient use of existing resources for parallel and distributed applications. At a coarse level of granularity, advances in runtime systems for parallel programs have been proposed in order to control available resources as efficiently as possible by utilizing idle resources and using task migration. Simultaneously, at a finer granularity level, advances in algorithmic strategies for dynamically balancing computational loads by data redistribution have been proposed in order to respond to variations in processor performance during the execution of a given parallel application. Combining strategies from each level of granularity can result in a system which delivers advantages of both. The resulting integration is systemic in nature and transfers the responsibility of efficient resource utilization from the application programmer to the runtime system. This paper presents the design and implementation of a system that combines an algorithmic fine-grained data parallel load balancing strategy with a systemic coarse-grained task-parallel load balancing strategy, and reports on recent experimental results of running a computationally intensive scientific application under this integrated system. The experimental results indicate that a distributed runtime environment which combines both task and data migration can provide performance advantages with little overhead. It also presents proposals for performance enhancements of the implementation, as well as future explorations for effective resource management.

1 Introduction

Resource management for applications in distributed computing environments is a complex problem. Over time, various techniques to manage resources at coarse and fine levels of granularity have been proposed. An essential component of resource management at each level is load balancing. In general, individual processors may vary in performance, external workload, or data distribution. 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

*This work was supported by the National Science Foundation Grant EEC-8907070.

have been proposed at system level, while fine-grained strategies have been proposed at algorithmic level.

1.1 Systemic (Coarse-Grained) Load Balancing

Task-parallel applications that use task migration represent a coarse degree of load balancing. This involves the 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 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 [17] and DQS [29] 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 [30], and Hector [21] work in this fashion. However, the Hector distributed runtime environment used in this paper is unique in the depth and breadth of information gathered about tasks at runtime. Hector is, to the best of our knowledge, the only runtime system which 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 application and is typically fine-grained. Data-parallel programs use data migration (or dynamic data allocation) to maintain a balanced load 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 either can negotiate as peers to exchange data from busy tasks to idle ones or can have a central master that allocates data to worker tasks. Systems based on Factoring [14] and Fractiling [4][3][13] are examples of the former, and Piranha [8] 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 and has been successfully implemented in N-body simulations [4][3]. 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. The 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. The resulting integrated load balancing strategy is systemic in nature, and therefore the burden on the applications programmer is reduced.

Hectiling [22][5] combines 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). It manages resources at both levels of granularity and provides a more efficient utilization of resources than either technique could provide individually. This paper reports on the design and implementation of Hectiling, and presents recent experimental results of running N-body simulations in the Hectiling environment. Ongoing work on Hectiling indicates that a runtime system, combining both task and data migration, can provide performance advantages with little overhead. The paper presents the successes and limitations of this implementation, as well as proposes future directions for effective resource management.

1.4 Organization of this Paper

This paper is organized as follows. Section §2 presents the pertinent background and related work in the areas of systemic and algorithmic load balancing. Section §3 describes the design and implementation of Hectiling and presents experimental results and performance enhancement techniques. It also discusses the successes and limitations of this integrated system. Finally, conclusions and future directions of this ongoing research are presented in section §4.

2 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[1]. 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 [34] work in this manner. The alternative is to provide support for task migration and checkpointing by the runtime system. Systems such as Condor [30] work in this fashion.

All systems mentioned in the survey provide some degree of load balancing at task 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 provide 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 above mentioned survey is LSF [34]. 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. Another clustering system presented in the above mentioned survey and developed at University of Wisconsin is Condor. It is a widely used public-domain cluster management software package [30]. 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 addition, both use relatively coarse load information for initial allocation purposes and for determining if hosts are idle or busy. Neither system gathers 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 [9]. 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 [19], whereas Gibbons proposed a simpler system to correlate runtimes to different job queues [10]. 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 [24]. 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 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 an 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, it combines the functions of monitoring and execution contained in LSF's two distributed daemon processes [34].

Hector's instrumentation combines three different mechanisms [24]. 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 [21]. Both Hector and Hectiling use the 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 the 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, a large class 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 [15][28] [27][33][25][6][7]. Most of these methods use profiling by gathering information on the work load from a previous time step in the execution of the algorithm in order to estimate the optimal work load distribution at the present time step. "Profiling", in this context, refers to 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 [32][26][27] [33]. Randomly assigning certain size amount of work to processors has also been considered to improve the performance of simulations due to load imbalance [11]. 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 work load 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 accommodate 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 [20][31][18][16].

Factoring, a scheduling scheme that evolves from earlier loop scheduling techniques, balances processor loads while reducing the overhead of synchronization [14]. 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 [2][13]. 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 [3][4]. 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 [3][4]. 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 [2][3][4]. 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 in processor finishing time among the simulation tasks was extremely small, indicating a very good load balance was obtained. Since performance improvements were obtained even on uniform data distributions, underscoring the need for a scheduling scheme that accommodates system-induced variance.

Section §3. describes our integration of Fractiling into the Hector environment and presents some of the experimental results obtained from running N-body simulations.

3 Hectiling

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 a Fractiling application may suffer from poor resource utilization, because it does not support task migration. 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 [22][5], combines systemic information gathering and task migration capabilities of Hector with fine-grained algorithmic load balancing advantages of Fractiling. Since Fractiling requires communication to control exchanges of data between tasks and Hector has the proper information gathering facility, their combination results in a system which can provide a more efficient resource utilization.

3.1 Design and Implementation

An architecture for Hectiling is described in [22] and shown in Figure 1. The first phase of this design involves 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 Hector paradigm, the MPI tasks don't communicate with the MA. Thus, a communication mechanism has to be devised from the 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

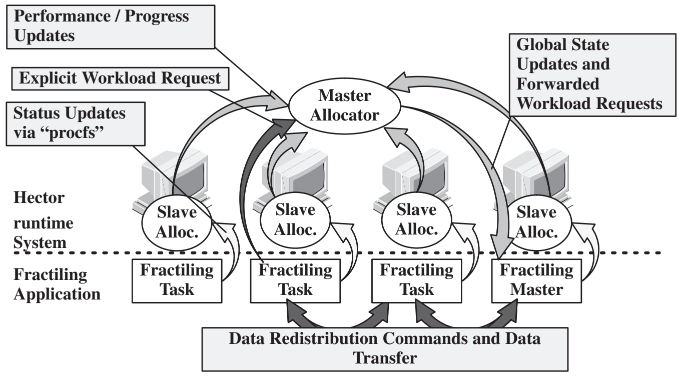

Figure 1: A Complete Hector-Fractiling Interface

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 2). 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 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. 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

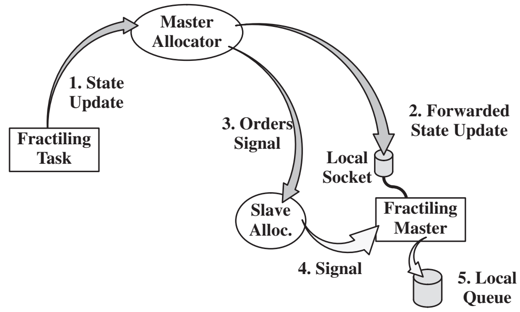

Figure 2: Fractiling State Update: Interrupt-driven Model

migration to the Fractiling Master.

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.

3.2 Experimental Results and Performance Enhancements

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 §3.2.1. Section §3.2.2 describes the results of experiments with Hectiling using process migration. These implementations, written in C and using MPICH implementation of MPI, were run on a dataset of 100k particles. 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. Three different data distribution were used: a uniform distribution ("Uniform"), a nonuniform

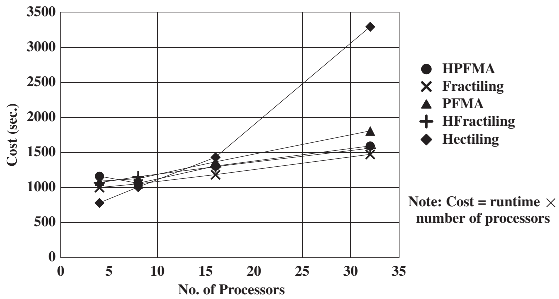

Figure 3: Costs for Uniform Distribution

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").

3.2.1 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 [12] 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) [23].

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 the "Uniform", "Gaussian", and "Corner" distributions are shown in Figures 3, 4, and 5. 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 generally lower. However, as the number of processors increases, the cost of Hectiling becomes higher than that of HFractiling. The coefficients of variation (c.o.v.) of processors finishing times are shown in Figures 6, 7 and 8. They are significantly lower for Hectiling and HFractiling when compared to PFMA.

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

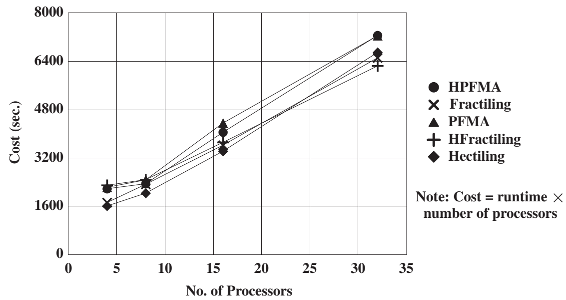

Figure 4: Costs for Gaussian Distribution

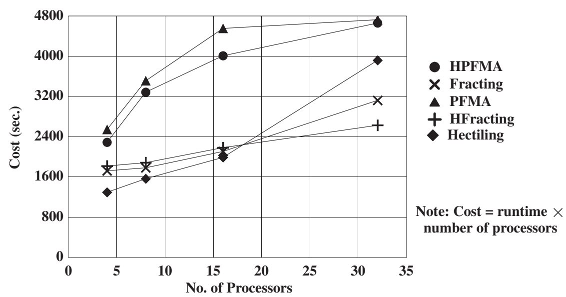

Figure 5: Costs for Corner Distribution

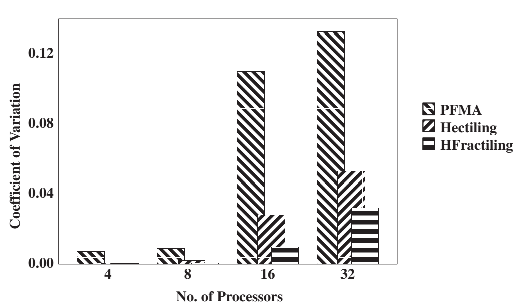

Figure 6: Coefficients of Variation for Uniform Distribution

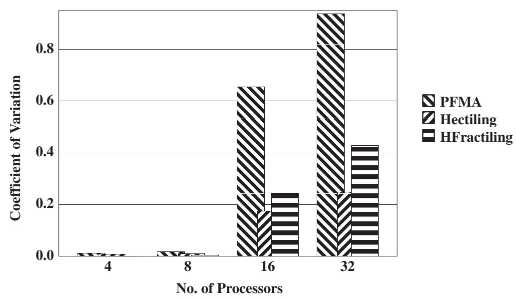

Figure 7: Coefficients of Variation for Gaussian Distribution

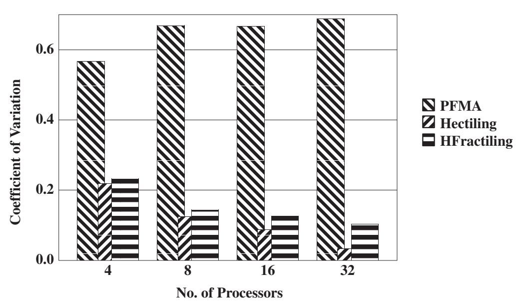

Figure 8: Coefficients of Variation for Corner Distribution

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 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. 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

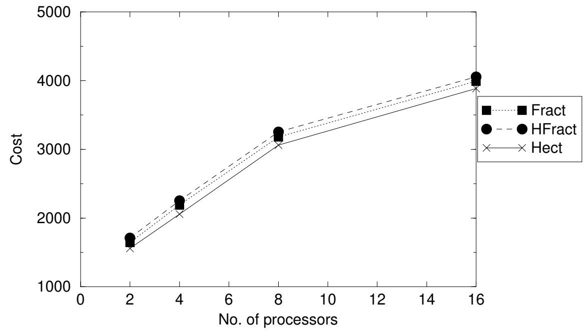

Figure 9: Costs for Uniform Distribution without load

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. Creating separate sockets will require major modification of the MA code. This work is in progress and results will be reported in the future.

3.2.2 Hectiling with migration

In this phase of testing three implementations of N-Body Simulations, using Fractiling, HFractiling and Hectiling were studied. Unfortunately due to system problems, 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 limited 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 were conducted with no external load. The costs of runs on all distributions without external load are shown in Figures 9, 10, and 11. The second set of experiments were conducted with controlled external load to measure the performance of migration. A specially developed external application which takes 50% of the processor cycles

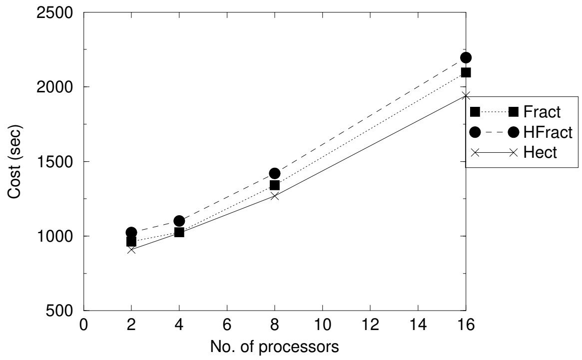

Figure 10: Costs for Gaussian Distribution without load

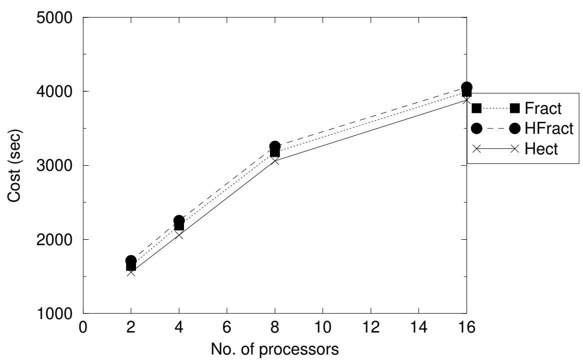

Figure 11: Costs for Corner Distribution without load

was launched on half the processors. The execution costs for all the distributions of the second set of experiments are shown in Figures 12, 13 and 14.

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 §3.2.1. However, when there is external load, the cost of Fractiling is found to be always higher than that of HFractiling or Hectiling, and also is found considerably higher than that of Fractiling with no external load. This can 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 external load does not result in 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 work loads 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 considerably low.

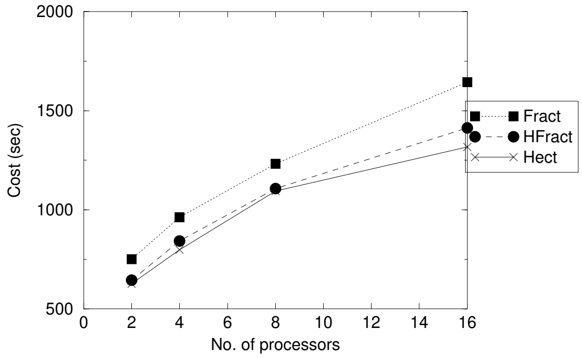

Figure 12: Costs for Uniform Distribution with load (migration)

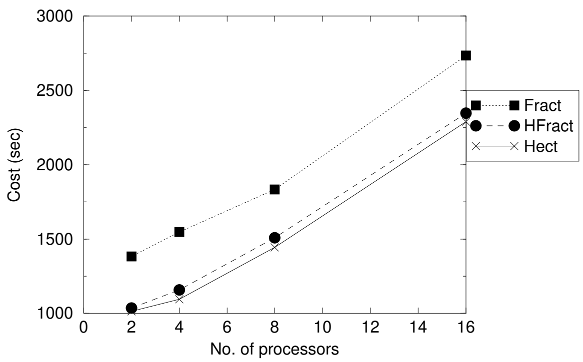

Figure 13: Costs for Gaussian Distribution with load (migration)

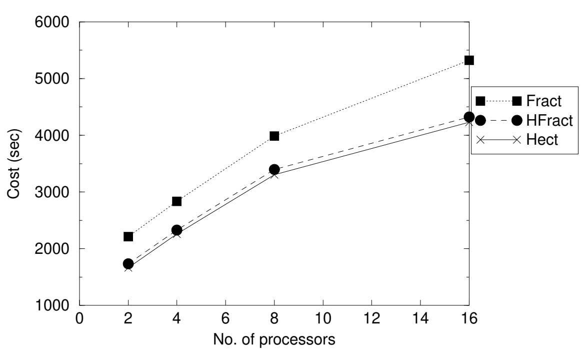

Figure 14: Costs for Corner Distribution with load (migration)

4 Conclusions 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 paper described 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 have been 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 redistributing the data assignments among tasks.

After observing that Fractiling could benefit by accessing the run-time information gathered by Hector, it was decided to develop an interface between them. In the complete interface, the Fractiling Master draws on performance information gathered at runtime by Hector to make better informed reallocation decisions. This allows techniques such as prefetching and fine-grained data redistribution. A partially completed interface was tested in order to measure the overhead of passing state-update messages through Hector's Master Allocator. The performance of the interfaced version was better than that of Fractiling alone or Fractiling under Hector, in the presence of external load as well as 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.

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.

References

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

[2] I. Banicescu. *Load Balancing and Data Locality in the Parallelization of the Fast Multipole Algorithm.* PhD thesis, Polytechnic University, January 1996.

[3] I. Banicescu and S. F. Hummel. Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations. In *Proceedings of Supercomputing'95 Conference*, 1995.

[4] I. Banicescu and R. Lu. Experiences with Fractiling in N-Body Simulations. In *Proceedings of High Performance Computing'98 Symposium*, pages 121–126, 1998.

[5] I. Banicescu, S. Russ, M. Bilderback, and S. Ghafoor. Competitive Resource Management in Distributed Computing Environment with Hectiling. In *Proceedings of High Performance Computing'99 Symposium*, pages 337–343, 1999.

[6] J. A. Board, J. Causey, J. F. Leathrum Jr., et al. Accelerated Molecular Dynamic Simulations with the Parallel Fast Multipole Algorithm. *Chemical Physics Letters*, 198:23–34, 1992.

[7] J. A. Board, Z. S. Hakura, W. D. Elliot, et al. Scalable Variants of Multipole-based Algorithms for Molecular Dynamics Applications. In *the Proceeding of Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 295–300, Philadelphia, February 1995. SIAM.

[8] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *Computer*, 28(1):40–49, 1995.

[9] 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.

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

[11] A. Y. Grama, V. Kumar, and A. Sameh. Scalable Parallel Formulations of Barnes-Hut Method for N-Body Simulations. In *Proc. of Supercomputing'94*, pages 439–448, November 1994.

[12] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulation. *Journal of Computational Physics*, 73:325–48, May 1987.

[13] S. F. Hummel. Fractiling: A Method for Scheduling Parallel Loops on NUMA Machines. Technical report, IBM RC18958, 1993.

[14] S. F. Hummel, E. Schonberg, and L. E. Flynn. A Practical and Robust Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[15] M.T. Jones and P.E. Plassman. Parallel Algorithms for Adaptive Mesh Refinement. *SIAM Journal on Scientific Computing*, 18:686–708, 1997.

[16] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and Loop Scheduling on NUMA Machines. *Proceedings of Int. Conf. on Parallel Processing*, pages II140–II147, 1993.

[17] LSF. Product Reviews: Platform Computing Corp. Load Sharing Facility (LSF). *SunExpert*, 8(8):62–64, 1997.

[18] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, 1992.

[19] 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.

[20] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[21] J. Robinson, S. Russ, B. Flachs, and B. Heckel. A Task Migration Implementation of the Message-Passing Interface. In *5th High Performance Distributed Computing Conference (HPDC-5)*, 1996.

[22] S. Russ, I. Banicescu, S. Ghafoor, B. Janapareddi, J. Robinson, and R. Lu. Hectiling: An Integration of Fine and Coarse-Grained Load-Balancing Strategies. In *Proceeding of the IEEE International Symposium on High Performance Distributed Computing '98*, pages 106–113, 1998.

[23] S. Russ, I. Banicescu, S. Ghafoor, B. Janapareddi, J. Robinson, and R. Lu. Integrating Fine and Coarse-Grained Load-Balancing Strategies. *Submitted to Parallel and Distributed Computing Practices (PDCP) Special Issue on High Performance Computing on Clusters*, 1998.

[24] 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*, 7(2):47–55, 1999.

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

[26] J. Singh. *Parallel Hierarchical N-Body Methods and Their Implications for Multiprocessors.* PhD thesis, Stanford University, 1993.

[27] J. Singh, C. Holt, T. Totsuka, et al. A Parallel Adaptive Fast Multipole Algorithm. In *Proc. of Supercomputing'93*, pages 54–65, 1993.

[28] A. Sohn, 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*, pages 189–192, 1997.

[29] Supercomputer Computations Research Institute, Florida State University. *DQS User Manual – DQS Version 3.1.2.3*, 1995.

[30] T. Tannenbaum and M. Litzkow. *The Condor Distributed Processing System.* Dr. Dobbs' Journal of Software Tools for Professional Programmer, 1995.

[31] T. H. Tzen and L. M. Ni. Dynamic Loop Scheduling for Shared-Memory Multiprocessors. *Proc. Int. Conf. on Parallel Processing*, II:247–250, 1991.

[32] M. Warren and J. Salmon. Astrophysical N-Body Simulation Using Hierarchical Tree Structures. In *Proc. of Supercomputing'92*, 1992.

[33] M. Warren and J. Salmon. A Parallel Hashed Oct Tree N-Body Algorithm. In *Proceeding of Supercomputing'93*, pages 12–21. IEEE Computer Society, 1993.

[34] S. Zhou. *LSF: Load Sharing and Batch Queueing Software.* Platform Computing Corporation, North York, Canada, 1996.