Hectiling: An Integration of Fine and Coarse–Grained Load–Balancing Strategies [1]

Samuel H. Russ, Ioana Banicescu, Sheikh Ghafoor, Bharathi Janapareddi, Jonathan Robinson, and Rong Lu
{russ,ioana,sheikh,jana,rong}@erc.msstate.edu, jon@aue.com
Mississippi State University
NSF Engineering Research Center for Computational Field Simulation

Abstract

Abstract — General–purpose programmers have come to expect a high degree of portability among widely varying architectures. Advances in run–time systems for parallel programs have been proposed in order to harness available resources as efficiently as possible. Simultaneously, advances in algorithmic ways of dynamically balancing computational load have been proposed in order to respond to variations in actual performance and therefore in runtime. The primary mechanism for harnessing idle resources effectively, task migration, can be used alongside the primary mechanism for dynamic load balancing, data redistribution. Besides the fact that the two methods can be used simultaneously to spur further increases in performance, the run–time information–gathering infrastructure necessary to detect and use idle resources can also benefit dynamically load–balanced applications. This paper describes an architecture for and preliminary implementation of a system that combines data–parallel load–balancing with task–parallel load–balancing. Performance test results are included as well.

1: Introduction and Survey of Existing Systems

One of the responsibilities of a parallel program and/or run–time system is that of load–balancing. Individual processors may vary in performance, external workload, or data distribution, and so methods to maintain an even distribution of work are usually needed to obtain good performance and speedup.

One can consider a parallel program as consisting of p threads of execution and q data partitions. In general, p does not have to equal q (although this is usually the case). To maintain a balanced load, the threads can be moved, the data in the partitions can be redistributed, or the two (matched pairs of threads and data) can be moved together. These three cases can be called "thread migration", "data migration", and "task migration", respectively.

This paper considers the network–of–workstation environment, and in the NOW environment there is an additional reason to migrate work. It is often desirable to return a workstation back to its "owner" and release the CPU and memory resources. Thus "release of resources" is an additional reason to provide and use task migration. This is an important distinction, as task–migration–based load–balancing systems can perform task migrations for reasons unrelated to load–balancing, and data–migration and thread–migration–based systems may also provide task migration services. The three styles of migration (and of programming) are characterized below.

1.1: Task Migration

Task–parallel applications tend to be coarse–grained, and task migration, involving transfer of the program's state to another computer during run–time, represents a coarse degree of load–balancing. It has the advantage of a natural mapping to the operating system (the entire process is transferred) but the disadvantage of being relatively cumbersome. It also has the advantage of being able to release resources (such as workstations) back to indiviual users by moving the work elsewhere and freeing up both the CPU and memory.

1.2: Data Migration

Data migration is typically supported by the application itself. That is, data–parallel programs tend to use data migration (or dynamic data allocation) to maintain a balanced load, and therefore tend to be self–balancing.

[1] This work was funded in part by NSF Grant No. EEC–8907070 Amendment 021 and by ONR Grant No. N00014–97–1–0116.

This represents a finer grain of control than task migration, as only fractions of a program state have to be moved.

1.3: Thread Migration

Thread–parallel applications take advantage of shared–memory multiprocessors to operate over a large data set. As the case with data–parallel applications, they are typically supported at the application/algorithm level and may offer a fine degree of control due to the relatively small state size of a single thread.

1.4: A Run–Time System for All Three

The ideal run–time system should provide support for all of these strategies, as they have complementary sets of advantages. Once the programmer has expressed the algorithm to be used, the run–time 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", but 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 key point is that these load–balancing strategies can actually work in concert to provide additional benefits.

It is the desire to support multiple load–balancing strategies that provides impetus for this work. This paper demonstrates that a prototype run–time system, combining both task and data migration, can provide performance advantages with little overhead, and outlines an architecture for even tighter coupling between the task–based and data–based balancing mechanisms. After a review of both data–based and task–based load balancing, a joint architecture is outlined and some preliminary testing and performance results are presented.

2: Data–Parallel, Fine–Grained Load Balancing

2.1: Background and Related Work

Although many scientific application algorithms are amenable to parallelization, performance gains from execution on parallel machines are difficult to obtain due to load imbalances caused by irregular distributions of data, by the different processing requirements of data in the interior versus those near the boundary of the space, and by system effects (such as data access latency and operating system interference) . The distribution of data may need to change at each time step in the algorithm, for example.

Currently, most data–parallel scientific applications use static scheduling methods to address performance degradation due to load imbalance. Some of them use repetitive static methods to adapt to variable work loads from one step to another. However, this requires profiling which results in increased overhead. "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.

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 will be too coarse. If the amount of data is too small, the process of exchanging data will occur much overhead. The Fractiling method was developed in response to the shortcomings of these other methods [2],[7]. It draws on earlier schemes that schedule loop iterations in decreasing–size chunks: the early larger chunks have relatively little overhead, while the later smaller chunks smooth over their unevenness [10], [8].

2.2: Current Work and Recent Results

Fractiling simultaneously balances processor loads and maintains locality by exploiting the self–similarity properties of fractals. It is based on a probabilistic analysis, and thus, accommodates load imbalances caused by predictable phenomena, such as irregular data, and unpredictable phenomena, such as data–access latencies and variations in processor performance.

In fractiling, work and the associated data are initially placed to processors in tiles so as to maximize locality, and processors that finish early are allowed to "borrow'' decreasing–sized subtiles of work from slower processors to balance loads. Each successive subtile is one–half the size of the previous subtile. Early in the program run, large performance variations can be accommodated by exchanging large subtiles. As the program runs, 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 tracks the size of its currently executing subtile, and 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" data requirement.

In experiments on a KSR1 and a IBM SP2 the performance of N–body simulation codes were improved by as much as 53% by fractiling [3],[4]. The corresponding coefficient of variation in processor finishing time among the simulation tasks was extremely small, indicating a very good load–balance was obtained. Performance improvements were obtained on uniform and nonuniform distributions of bodies, underscoring the need for a scheduling scheme that accommodates system–induced variance.

In fractiling, negotiations by idle resources for more work replaces profiling. The load–balancing actions taken by fractiling 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 in mid–iteration. Note that the bulk of load–balancing work is done by idle tasks, and so little negative effect on run–time is expected. Additionally, note that fractiling does not have to be aware of the source of imbalance in order to spur useful performance gains. Even applications where the amount of computation per grid point (or data element) varies dynamically can benefit, as it will simply look for idle and busy resources.

Section 4 describes our integration of Fractiling into the Hector environment (coarse–grained load balancing environment) and presents some of our experimental results obtained on running N–body simulations.

3: Task–Parallel, Coarse–Grained Load Balancing

3.1: Background and Related Work

Many systems exist to run sequential and parallel programs on networked workstations, SMP's, and MPP's. 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 good survey of task–based job–scheduling systems may be found in [1]. Recent work has highlighted the benefits of extracting information from applications as they run [5]. For example, Nguyen et al. have shown that extracting run–time information can be minimally intrusive and can substantially improve the performance of a parallel job scheduler [9]. Gibbons proposed a simpler system to correlate run–times to different job queues [6].

These approaches have shown the ability of detailed performance information to improve job scheduling. However, to summarize, these approaches have several shortcomings. First, some of them require special–purpose hardware. Second, some systems require user modifications to the applications program in order to keep track of relevant run–time performance information. Third, the information that is gathered is relatively coarse. The Hector environment is designed to address these shortcomings.

3.2: Hector: A Task–Parallel Load–Balancing Environment

Hector is designed to provide features transparently to MPI programs [11],[12]. Hector's MPI library provides a complete MPI implementation as well as interfaces to a self–migration facility, to Hector's command and control structure, and to an instrumentation facility. (The MPI implementation is based on the MPICH implementation developed at Argonne National Laboratory and Mississippi State University.) Thus unmodified MPI programs can be linked with this library and obtain access to its services. These interfaces are diagrammed in Figure 1.

It is also designed to provide the infrastructure to control parallel programs during their execution and monitor their performance. It does this by running in a distributed manner, as shown in Figure 2. 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'' (pieces of MPI programs) under their control and execute commands issued by the MA.

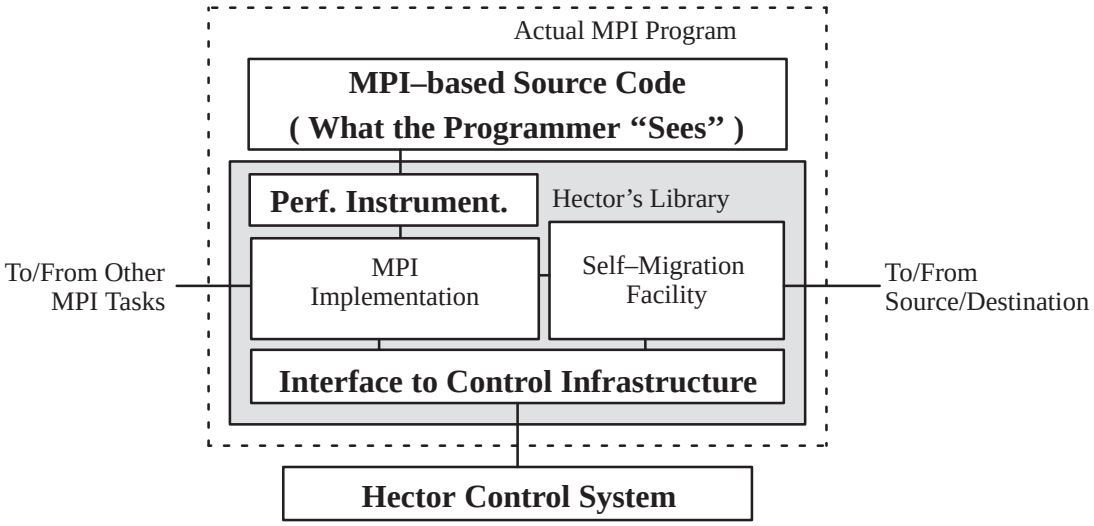

Figure 1: The Hector Library and Its Interfaces

Many of Hector's features are very important for running large MPI jobs, but have been described before in the literature. For example, migration of tasks during run–time is accomplished by the combination of a method that permits a running UNIX task to transfer its own state a specially designed communications protocol that maintains message consistency [11]. The ability to migrate tasks naturally supports the ability to checkpoint them for "safe" storage and can be used to restart the program if a failure occurs. Hector collects run–time information about the CPU loading and memory usage of every candidate platform and determines a machine's status (busy or idle), its relative performance, the amount of memory available, and the total CPU usage of each MPI task under its control [12]. Task self–instrumentation provides more detailed information as well; the SA's can read the task's address space in order to gather this information.

3.3: Current Work and Recent Results

Testing was performed on specially instrumented 90 MHz Sparcstation 10's in order to make detailed timing measurements [12]. Note that the measurements below are for wall–clock time. First, every call to MPI activates local instrumentation and adds about 4.2 μs latency per MPI call. Second, information about every running task and the machine itself is gathered by the local slave allocator. As an example, the Sparc 10 took an average of 346 μs per estimate with 12 tasks running. Since this is only done once every five seconds, this represents a negligible amount of CPU time. Third, the master allocator must process status messages from every slave allocator. The average status message took about 6NT + 18 μs, where NT is the number of tasks. Again, the processing time is extremely small and implies that the practical limit for the number of slaves that a single master can monitor is limited more by network bandwidth than by processing time. More details about the overhead, along with its accuracy, may be found in [12].

4: Integration and Architecture of Combined System

4.1: Run–Time Needs of Data–Parallel Jobs

Data–parallel jobs, such as fractiling applications, exchange requests for additional workload from idle tasks with extra work from busy tasks. The process of data–parallel load–balancing therefore requires knowledge of both busy and idle tasks. Detection of idle tasks can be very simple—since they are idle, they can send requests for additional work. Detection of busy tasks requires some ongoing knowledge of the status of all tasks.

(An alternative strategy, master/worker parallelism, requires instead that the master retains the entire workload and distributes it in response to requests. This reduces the need to know about task progress, but if the increment of workload that the master distributes is too small or too large, inefficiency and/or imbalance results.)

4.2: An Architecture for Cooperative Execution

The Hector environment is equipped to support data–parallel jobs. For example, data–parallel load balancers can benefit from more detailed information about

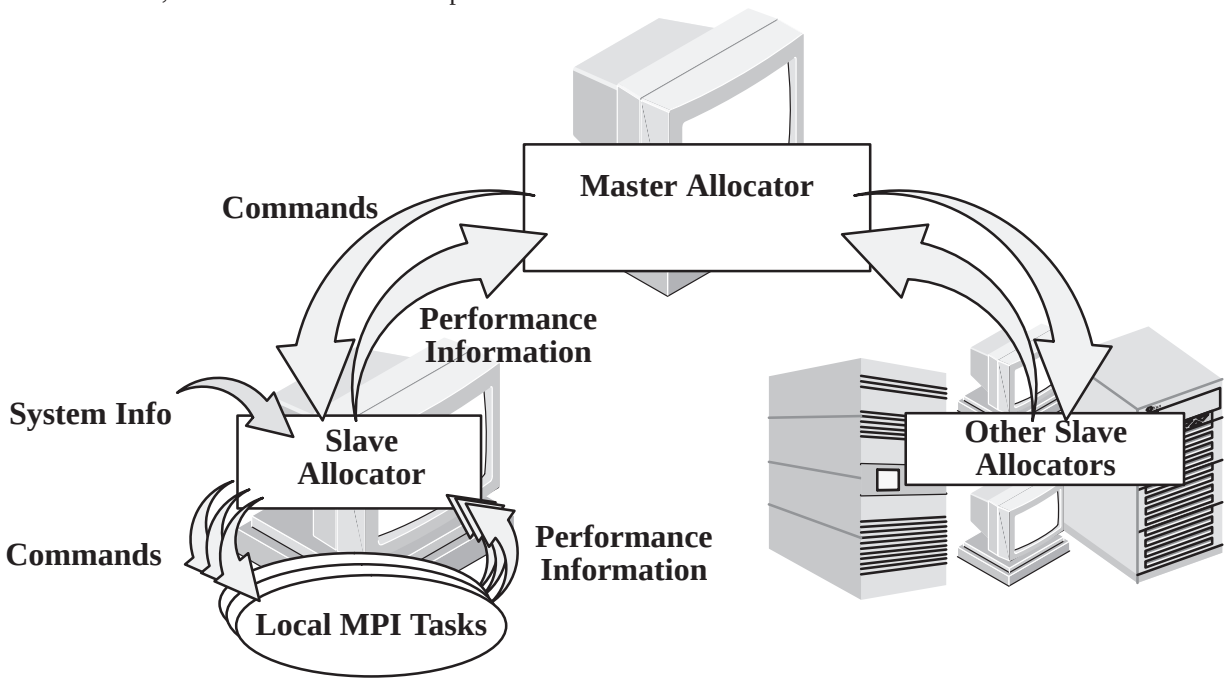

Figure 2: Hector's Run–Time Structure

system loads and task completion. Since much of this information is already gathered by the Hector run–time system, it makes sense to draw on Hector's information repository in order to make "better informed" data migration decisions. Likewise, requests for additional work can be funnelled through Hector's information–gathering infrastructure.

An architecture for running a fractiling (or any data–parallel load–balancing) application under Hector is shown in Figure 3. Note that the data–parallel load–balancer is assumed to have some sort of "master" task to broker workload requests. This master may be an explicitly declared task or it may simply be an idle one.

The Slave Allocators continuously monitor the tasks and their nearness to completion via shared–memory. Nearness to completion can be monitored by checking an iteration loop counter, for example. (The Slave Allocators have demonstrated the ability to read the address space of individual tasks and send the information to the Master Allocator [12].) This is sent to the master allocator as part of the normal "machine update" process, and so is continuously updated. Individual tasks, once they have completed their current work assignment, ask the fractile master for additional workload. These requests are also sent to the master allocator.

The master allocator forwards the global state information and the explicit workload requests to the fractiling master (or masters, if multiple masters are used for efficiency). The fractiling master can use each task's status to estimate its nearness to completion. Workload can then be transferred from relatively "busy" tasks to idle ones. Since the state of each task is being monitored by the slaves, busy tasks can be identified. Since idle tasks post explicit requests for additional workload, they are already known.

The first phase of implementation involves routing the explicit requests for additional work through the master, and is nearing completion. The second phase will be to add the instrumentation of individual tasks and the forwarding of this information to the appropriate fractile master. That is, in the second phase state information such as progress indicators and/or iteration loop counters will be forwarded to the fractile master.

The third phase of work will draw on the capabilities of the instrumentation to guide prefetching of data. That is, the instrumentation will be used to detect tasks that are busy and tasks that are *about to be* idle. Additional data can be sent to nearly–idle tasks before becoming idle, so that it is available immediately. This approach should reduce actual idle time in tasks and should reduce the workload request message traffic. Idle time is reduced because under the original scheme a task must wait until the fractile master receives and processes the workload request before receiving additional data to process.

Finally, the amount of data sent to idle processes can be adjusted by the relative performance imbalance. If task A is idle and has historically been twice as fast as task B, then two–thirds of task B's remaining workload should be sent to task A.

4.3: Benefits of the Approach

There are several advantages of running self–balancing applications under a load–balancing run–time system. First, the ability of the system to relinquish workstations back to their "owners" is retained. Thus self–balancing

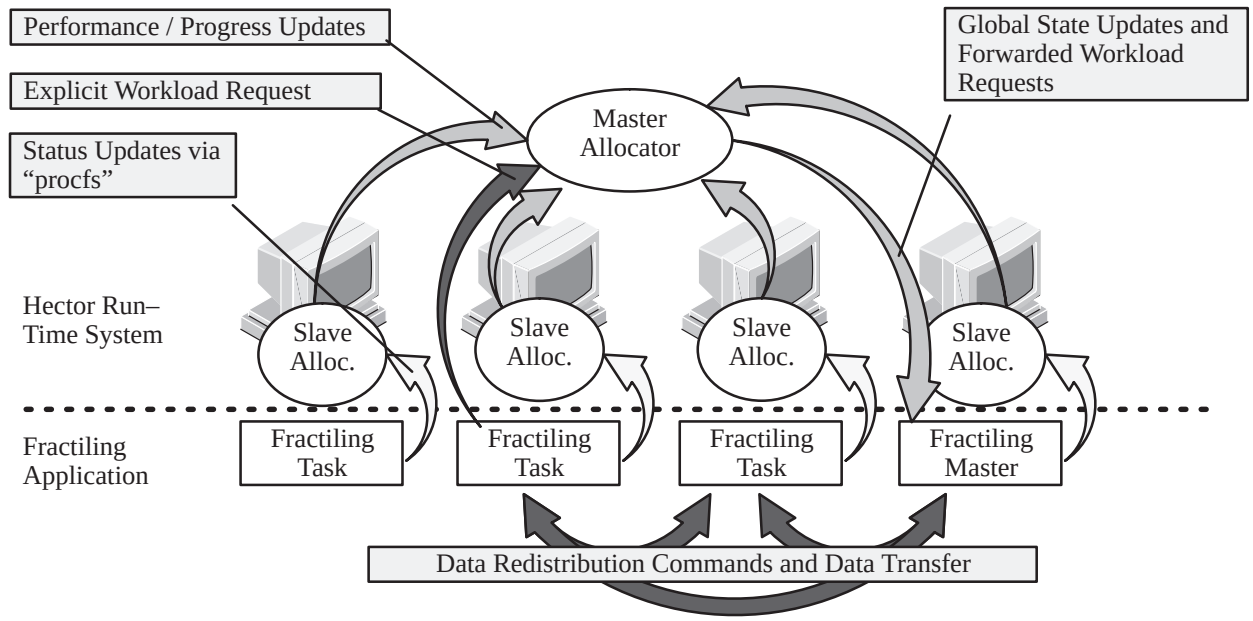

Figure 3: Running a Fractiling Data–Parallel Application under Hector

applications can completely abandon busy resources and rapidly reclaim idle ones. Second, construction of data–parallel applications is simplified because the run–time system can take over the information–gathering responsibilities. Additionally, the depth of information is greater because the run–time system already extracts detailed information. Third, data prefetching becomes possible. As pointed out above, this can reduce idle time and messaging traffic. Fourth, jobs with fundamentally different load–balancing strategies can be run alongside one another and be scheduled cooperatively.

4.4: Testing an Initial Implementation

For the purposes of initial testing, fractiling–based algorithms were run under Hector without any special–purpose interface between them. The purpose of this testing was to demonstrate the advantage of the fractiling–based algorithm over the same algorithm without fractiling and to show that running under Hector adds very little overhead.

A 100,000–point parallel fast multipole algorithm (i.e. N–body) simulation was coded twice, with and without fractiling. It was run on a cluster of 8 quad–processor 90 MHz Sparcstation 10's with three different data distributions—a uniform distribution of points ("Uniform"), a Gaussian distribution of points centered on the grid space ("Gaussian"), and a Gaussian distribution of points centered near one corner of the solution space ("Corner"). The non–fractile and fractiling cases were run with and without Hector over 4 to 32 processors on an otherwise unloaded cluster. The results are shown in Figure 4, Figure 5, and Figure 6 below for the uniform, Gaussian and corner cases respectively. The graphs show cost (run time times number of processors) on the vertical axis and number of processors horizontally, and so "lower" is "better". Each graph charts four cases. The N–body simulation without fractiling is called "PFMA" and N–body simulation with fractiling is labelled "Fractiling". The simulations run under Hector are also labelled "(Hector)".

The results confirmed that the addition of fractiling improved performance dramatically, and that these performance improvements were maintained with the addition of Hector. Hector added very little overhead, and even ran some jobs faster. Hector's MPI implementation is able to exploit shared–memory more easily than the non–Hector MPI implementation, and so the speedup from shared–memory usage offset the increased overhead of Hector's run–time instrumentation. The most dramatic improvements from Fractiling occurred when the data distribution was the most uneven, as would be expected, because Fractiling is able to redistribute the uneven workload.

Further testing with the addition of external load on the cluster at run–time is underway. External load will cause the fractiling to redistribute data and will cause Hector to move tasks.

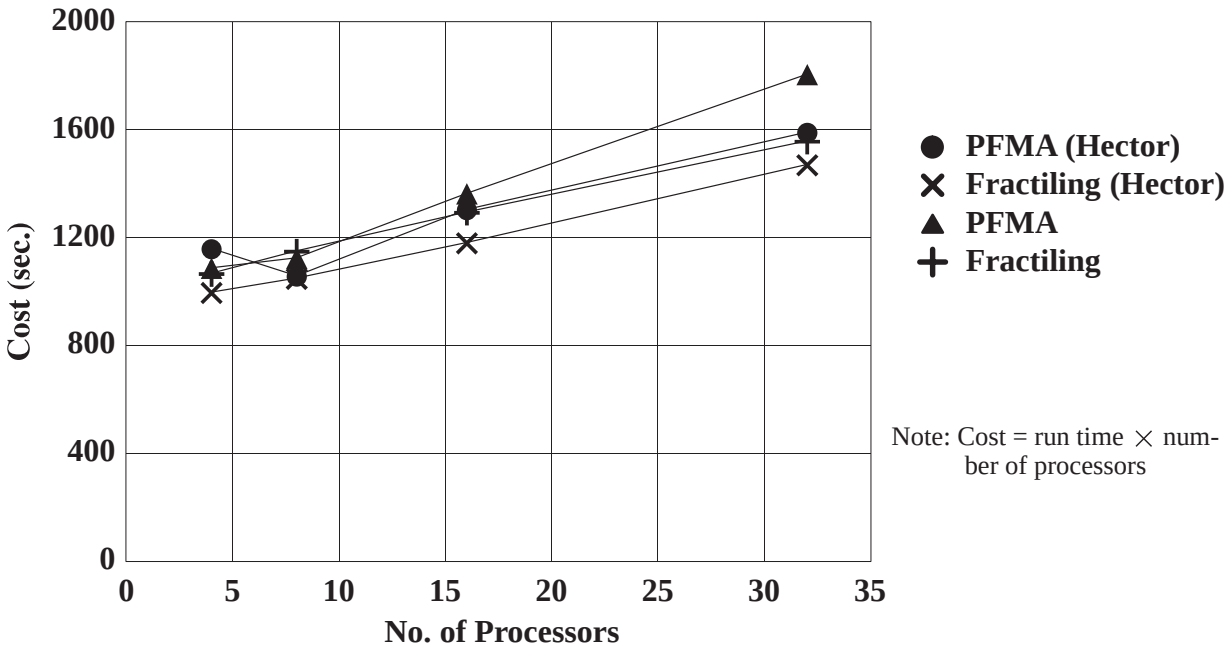

Figure 4: Costs for Uniform Distribution

5: Conclusions and Future Work

5.1: Support for Thread–Parallel Applications

The information gathered by the run–time system can benefit thread–parallel applications as well. One remaining obstacle is relatively poor support for DSM, which is important because most distributed thread systems are DSM–based. Once DSM support is obtained, the final system will be able to run data–parallel, thread–parallel, and task–parallel programs efficiently.

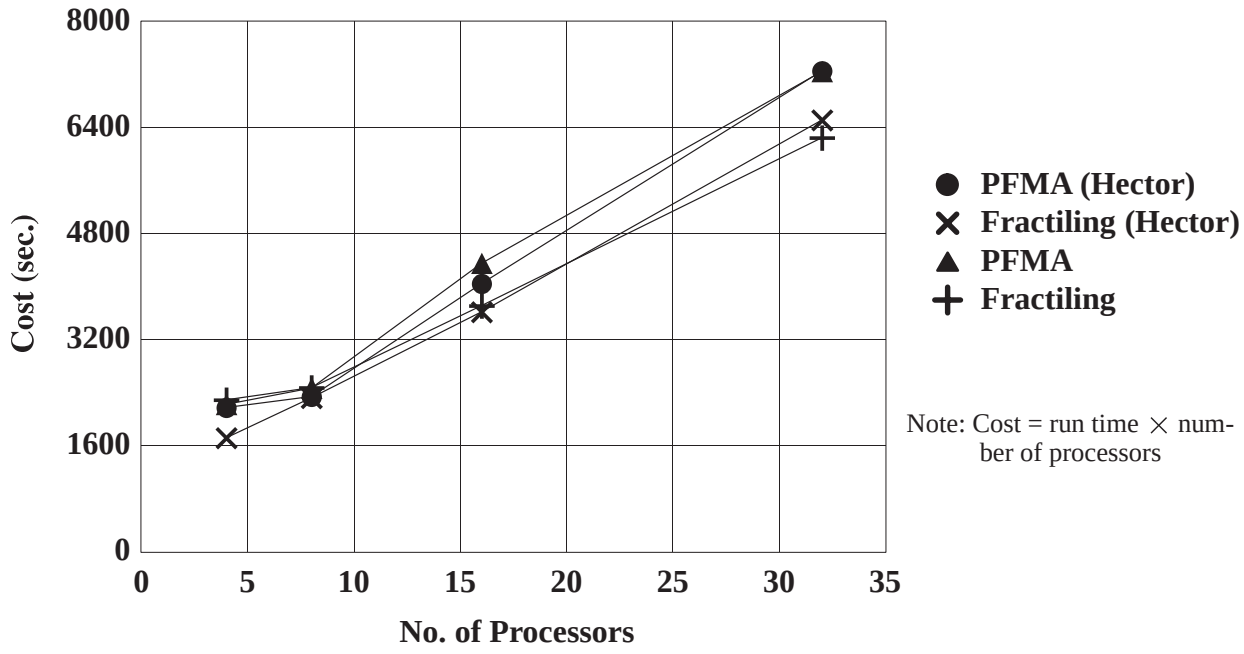

Figure 5: Costs for Gaussian Distribution

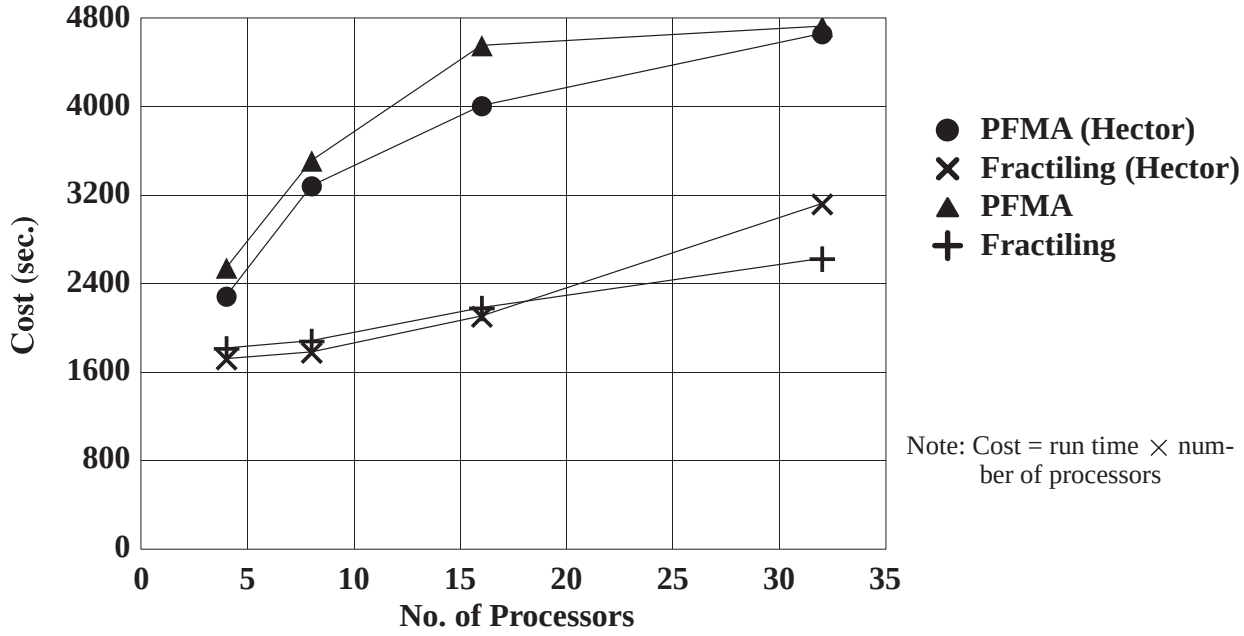

Figure 6: Costs for Corner Distribution

5.2: Real–Time (Time–Constrained) Applications

Communications events can be tagged and timestamped transparently, laying the groundwork for a run–time system with transparent support for measuring compliance with real–time deadlines and having the ability to alter the allocation of work to compensate.

5.3: Fault Tolerance

Task migration can be used to duplicate tasks. When accompanied with additions to the communications library to duplicate (and decimate) messages, the system could support rapid fault recovery via task duplication. Each task would have a "hot standby" task running on a physically separate machine, able to continue functioning in the presence of single–node failures.

5.4: Improvements to Fractiling

The current fractiling method uses fixed sub–tile ratios and involves exchanges of integer numbers of sub–tiles. Future work could use a finer–grained unit of data exchange, drawing on Hector's run–time information to provide an estimate of an optimal amount of data to exchange.

6: References

[1] M. A. Baker, G. C. Fox, and H. W. Yau, ''Cluster Computing Review'', Northeast Parallel Architectures Center, Syracuse University, 16 November 1995. Available via *http://www.npac.syr.edu/techreports/hypertext/sccs–0748/cluster–review.html*.

[2] I. Banicescu, *Load Balancing and Data Locality in the Parallelization of the Fast Multipole Algorithm*, Ph.D. Thesis, Polytechnic University, Department of Computer Science, 1996.

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

[4] I. Banicescu and R. Lu, "Experiences with Fractiling in N–body Simulations", Accepted to the 1998 High Performance Computing Conference (HPC '98).

[5] 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*, Geneva, April 1997.

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

[7] S.F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops", *Communications of the ACM*, Volume 35, No. 8, pp. 90–101, Aug. 1992.

[8] 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, Apr. 1992.

[9] T.D. Nguyen, R. Vaswani, and J. Zahorjan, "Using Runtime Measured Workload Characteristics in Parallel Processing Scheduling", *IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, Honolulu, HI, April 1996.

[10] C. Polychronopoulos and D. Kuck, "Guided Self–Scheduling: A Practical Scheduling Scheme for Parallel Computers", *IEEE Transactions on Computers*, Vol. C–36, No. 12, pp. 1425–1439, 1987.

[11] J. Robinson, S. H. Russ, B. Flachs, and B. Heckel, ''A Task Migration Implementation for the Message–Passing Interface'', *Proceedings of the IEEE 5th High Performance Distributed Computing Conference (HPDC–5)*, Syracuse, NY, August 1996, pp. 61–68.

[12] S. H. Russ, B. Meyers, C.–H. Tan, and B. Heckel, ''User–Transparent Run–Time Performance Optimization'', *Proceedings of the 2nd International Workshop on Embedded High Performance Computing*, Associated with the 11th International Parallel Processing Symposium (IPPS 97), Geneva, April 1997.