RC 19934 (02/15/95)
Computer Science 13 pages

IBM Research Report

Balancing Processor Loads and Exploiting Data Locality in Irregular Computations

Ioana Banicescu

Polytechnic University
Six MetroTech Center
Brooklyn NY 11201

Susan Flynn Hummel

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Balancing Processor Loads and Exploiting Data Locality in Irregular Computations [1]

Ioana Banicescu [2] and Susan Flynn Hummel [3]

Abstract

Fractiling is a scheduling scheme that simultaneously balances processor loads and exploits locality. Because it is based on a probabilistic analysis, fractiling accommodates load imbalances caused by both predictable phenomena, such as irregular data and conditional statements, and also unpredictable phenomena, such as data access latency and operating system interference. Fractiling exploits both temporal locality, which is often profitable for computations on regular data, and spatial locality, which is often profitable for computations on irregular data. Here, we report on a case study involving the application of fractiling to computations on irregular data, namely N-body simulations. In experiments on a KSR1, performance was improved by as much as 43% by fractiling. Performance improvements were obtained on nonuniform and uniform distributions of bodies, underscoring the need for a scheduling scheme that accommodates application as well as system induced execution time variance.

1 Introduction

On parallel machines, processor load imbalances are induced by application features, such as irregular data and conditional statements, and system effects, such as data access latency and operating system interference [10, 11, 9, 7]. There is a tension between exploiting data locality and load balancing, as the (re)assignment of work may necessitate access to remote data. *Fractiling* is a scheduling scheme that simultaneously achieves good load balancing and exploits data locality [12]. In fractiling, work and their associated data are initially

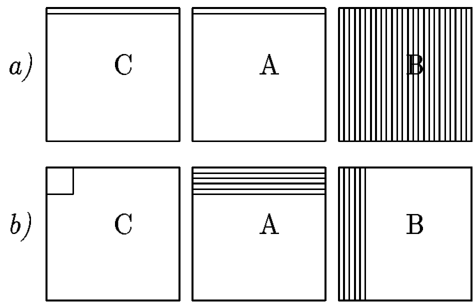

Figure 1: Rows of A and columns of B needed to calculate *a)* a row and *b)* a block tile of matrix multiplication $C = A \times B$.

assigned to processors so as to maximize data reuse (see figure 1), and processors that finish

[1]This work supported in part by NSF Grant CCR-9321424. The research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation and New York State; additional funding comes from the Advanced Research Projects Agency, the National Institute of Health, IBM Corporation and other members of the center's Corporate Research Institute.

[2]Polytechnic University, Six MetroTech Center, Brooklyn NY 11201.

[3]Polytechnic University and IBM T. J. Watson Research Center, email: hummel@mono.poly.edu

early are allowed to "borrow" decreasing-sized chunks of work from slower processors to balance loads. The sizes of the borrowed chunks are chosen so that they have a high probability of finishing before the optimal time. Another novel feature of fractiling is that its dynamic work assignment is self-similar, which improves locality and simplifies the implementation of work borrowing.

Fractiling exploits both temporal and spatial locality. Exploiting temporal locality is often profitable for computations on regular data, such as matrix multiplication [12, 13], and spatial locality for computations on irregular data, such as N-body simulations and finite element analysis. To apply fractiling to computations to regular data, the data/iteration space is tiled and assigned to processors. Irregular and sparse data can be represented as a dense data structure upon whose elements auxiliary variable-sized data structures are defined. Fractiling can then be applied to this dense data structure.

To demonstrate the effectiveness of applying fractiling to computations on irregular data, we chose as a case study algorithms for N-body simulations, which model the interaction of bodies over time. The irregularity of the data depends on the distribution of bodies, which changes per time step. N-body simulations arise in many areas of science, ranging from astrophysics to molecular biology. Having specified the initial positions and velocities of the N bodies, the problem is to find their positions after a number of time steps. The naive serial N-body algorithm has complexity $O(N^2)$. Recently $O(N log N)$ [2] and $O(N)$ [1, 16] algorithms that approximate the interaction of bodies within a specified accuracy have been developed. These later algorithms recursively partition space into a hierarchy of finer grained boxes, resulting in either quad-trees (2-dimensions) or oct-trees (3-dimensions) (see figure 2). At the lowest level, leaf boxes contain bodies, and at higher levels, they

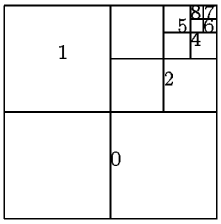

Figure 2: Recursive Geometric Bisection into 16×16 cells.

contain the summary effect of the bodies in their rooted subtree.

The calculation of forces on each body during a time step is, for the most part, independent. Thus, N-body algorithms are amenable to parallel execution [17, 33, 38]; however, parallel implementations of N-body simulations pose load balancing problems due to the irregularity of the distribution of bodies and, to a lesser extent, different processing requirements of bodies in the interior and of those near the boundary of the space [39, 34, 14]. To balance processor loads, various methods have been used (see §2). These methods primarily focus on balancing work loads by using information about the work load of the previous time step. Their cost increases with the number of bodies and processors.

Our approach to load balancing N-body simulations is to incorporate fractiling into the simpler quad- or oct-tree data structure. This method dynamically balances processor loads within each time step and does not rely on information on previous work loads. We compare implementations of a parallel and a fractiled N-body simulation on uniform and nonuniform distributions of bodies on from 4 to 32 processors of a KSR1. The N-body algorithm used in the experiments is due to Greengard [16]; we note, however, that our techniques can be applied to the other algorithms involving computations on irregular data, such as finite element analysis. The fractilied N-body simulation consistently improved performance for both uniform and nonuniform distribution of bodies.

In the next section, we review scheduling schemes for parallel machines. The fractiling method of scheduling is described in detail in §3. In §4, the parallel and the fractiled implementations of the N-body simulation algorithm on the KSR1 are outlined. We discuss experimental results and draw some conclusions in §4 and §5 (respectively).

2 Background and Related Work

We first discuss related general load balancing techniques, and then ones specific to N-body simulations. The schemes either explicitly assign bodies to processors or indirectly assign bodies by assigning subspaces. We consider the assignment of B bodies distributed over $S = S_1 \times S_2$ cells of space (where a cell is the size of the leaf boxes) to P processors. We model the time to process each body during a time step as a random variable with mean μ_B and variance σ_B. The execution time of each cell of space is similarly modeled as a random variable with mean μ_S and variance σ_S, e.g. the average execution time of a subspace of size K_S is $\mu_S K_S$. Given these definitions, $\mu_B B$ is equal to $\mu_S S$ (but $\sigma_B B$ is, in general, not equal $\sigma_S S$).

A simple way to ensure (close to) even processor loads is to dynamically assign work one unit at a time—the resulting load imbalance will be proportional to μ_B (or μ_S depending on whether subspaces or bodies are assigned). Unfortunately, the overhead incurred with such self-scheduling will be proportional to B/P (S/P). At the other extreme, work units can be initially assigned to processors in *chunks* of size B/P (S/P) with very little overhead; however, the expected load imbalance becomes proportional to $\sqrt{\sigma_B B/P}$ $(\sqrt{\sigma_S S/P})$ [21]. Although the square root is helpful, the load imbalances from static assignments can severely degrade performance [9]. Schemes that dynamically schedule work in chunks of K units have also been investigated [21, 7]. Kruskal and Weiss derive a formula for calculating the optimal K based on μ_B (μ_S), σ_B (σ_S), B (S) and P [21].

To better trade off scheduling overhead and load imbalances, work can be dynamically assigned to processors in decreasing-size chunks [21, 29, 37, 8, 9]. Decreasing the chunk size promotes load balancing. Factoring, for example, uses a probabilistic rule to choose the decreasing chunk sizes so that there is a high probability that the work will complete by the optimal finishing time $\mu_B B$ $(\mu_S S)$ [8, 9]. So that their execution times can be modeled as Pth order statistics, in factoring, each P chunks scheduled, a batch, are the same size; Flynn and Flynn Hummel derive a distribution-free bound for selecting optimal batch chunk sizes K_i based on μ_B (μ_S), σ_B (σ_S), B (S) and P.

Unfortunately, in practice, it is not possible to use the optimal chunk size formula derived by either Kruskal and Weiss (fixed size chunks) or Flynn Hummel *et al.* because the mean and variance of parallel work execution times depend not only on applications features, but on nondeterministic system effects as well. If characteristics of work execution times are unknown or hard to predict, then Flynn Hummel *et al.* suggest using a halving rule, wherein the size of each batch is half the remaining work. The intuition for this choice is that the sums (chunk execution times) of random variables (work unit execution times) is itself a random variable with (close to) normal distribution; the expected finishing time of a batch of P chunks of K work units with a truncated normal distribution is $2\mu_B K_0$ $(2\mu_S K_0)$. Since the average processor finishing time is $\mu_B K$ $(\mu_S K)$, this suggests that only half of the remaining work should be allocated per batch. In simulations [8], and experiments [9], Factoring using the halving approximation consistently outperformed other chunking schemes on dense-matrix benchmark programs with a wide variety of characteristics (e.g., fine versus coarse grain, constant versus variable algorithmic work).

The above schemes focus solely on load balancing and do not exploit locality. On machines with complex memory hierarchies, additional performance can be achieved by placing work and data on the same processing unit. Many techniques for achieving this have been developed in the context of static scheduling. For example, *tiling* reduces communication time by statically partitioning the data/iterate space of nested parallel loops into tiles whose shape is chosen to maximize data reuse and locality. Again, much research has been done in this area [19, 20, 23, 24, 30, 31, 40, 5, 6].

Some experimentation with schemes that combine these static techniques that exploit data locality with dynamic scheduling techniques that promote load balancing has been done [26, 25, 12]. In these schemes, work units and their associated data are initially placed on the same processor. The processor executes its units using a decreasing size chunking scheme to preserve load balancing. If a processor exhausts its local work, it acquires decreasing size work from other processors.

Affinity Scheduling (AS) [26] and Locality-based Dynamic Scheduling (LDS) [25] do not address the problem of executing multidimensional tiles in decreasing subtiles of the same shape. The shape of a tile is selected to maximize reuse, so self-similar subtile assignment improves locality. Self-similar subtile assignment is also important so that a complex history

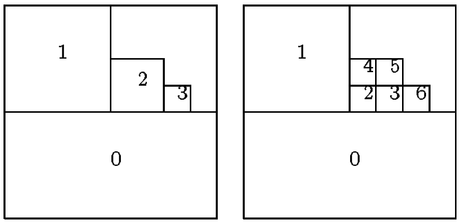

Figure 3: Illustration of self-similarity.

of executed subtiles need not be maintained: the space covered should be the same whether it was executed as 4 subtiles of size K or one of size $4K$ (see figure 3).

Flynn Hummel has developed a scheme, *fractiling* (discussed in §3), that combines factoring and tiling, to yield a dynamic scheduling method that exploits temporal and spatial locality for multidimensional problems [12]. Fractiling uses the factoring halving approximation discussed above to assign decreasing size chunks. A novelty of the fractiling approach is that processors acquire new work in the form of decreasing size subtiles, called *fractiles*, which conform to the optimal shape specified by the tiling algorithm. The subtiles assignments are computed in an efficient and elegant way by exploiting self-similarity property of fractals, namely shuffle row-major [36]. This self-similar property allows fractiling to be efficiently applied to multidimensional problems.

The techniques described thus far have been applied to computations on regular data, i.e., dense matrices. Saltz *et al.* use run-time analysis to *inspect* sparse matrix computations and to *generate* the appropriate data assignments and data transfers [32]. The cost of the inspector can be amortized if the loop is executed multiple times.

Some of the above techniques have been applied to N-body simulations to help balance processor loads. Grama *et al.*, for example, use a static decomposition. In the scheme, space is divided into subtiles of size s that are assigned randomly to processors [14]. With random assignment, the load imbalances of individual subtiles mute each other out (resulting in expected processor load imbalances proportional to $\sqrt{\sigma_S S/P}$ [21]). Gray describes a dynamic scheme wherein subtiles of size s are self-scheduled to processors [15].

Load balancing techniques specific to N-body problems, for example, orthogonal recursive bisection [3], hashed oct-tree [39] and costzones [34], use information about the distribution of bodies to guide their assignment to processors. As bodies move over time, the assignment is recomputed after each time step. Orthogonal recursive bisection (ORB) recursively divides space into P tiles that have the same number of work units [3]. It has been used to load balance N-body simulations [34, 38, 39]. The cost of applying ORB increases with the number of bodies and processors [34].

Warrren and Salmon use a load balancing technique for N-body simulations based on a hashed oct-tree (HOT). The hash function used to build the HOT is a space-filling numbering scheme, Morton ordering [27]. Bodies are "sorted" into morton order, and then consecutive subsequences of the sorted B/P bodies are assigned to processors [39]. Morton ordering is a fractal and the way that it fills space ensures good locality in the subsequences. Interestingly, the scheme used by fractiling for dense matrix computations, shuffle row-major, is equivalent to Morton ordering. The work per body is approximated and used to rebalance loads between time steps. A related scheme, costzones, is used by Singh *et al.* Costzones consisting of subtrees are assigned to each processor in turn until it has its share of the load as the tree is traversed [34]. The traversal order result in an assignment that is identical to that of the HOT [14]. In both schemes, the tree data structures are recomputed after each time step.

Grama *et al.* describe a variation of HOT algorithm where subtiles of size s are assigned in Morton order [14]. After each time step, processors compare their load of bodies to the average and adjust the subsequence endpoints accordingly.

In the rest of this paper, we describe the application of fractiling to schedule parallel N-body simulations. As described in the next section, fractling takes exploits both the locality and self-similarity properties of the shuffle row-major fractal. Moreover, because fractiling uses sizes chosen by a factoring rule, it accommodates load imbalances due to irregular data and computations, as well as system effects.

3 Fractiling Scheduling

To simplify the implementation of decreasing-size tiling for multidimensional problems, fractiling uses the factoring approximation mentioned in the previous section wherein the size for each consecutive P chunks (a batch) scheduled is half that of the previous P chunks. The scheme has been shown to be very robust [9] and is easily adapted to factoring rules where batch sizes decrease by other powers of two. The subtile shape can be any parallelepiped. For N-body simulations we are interested in tiles with blocked dimensions. For concreteness, we assume that a $S = S_1, ..., S_d$ space is tiled into P $S_0/\sqrt[d]{P} \times ... \times S_d/\sqrt[d]{P}$ shaped tiles. Our techniques can be applied to other distributions of processors to dimensions [12].

In general, it is not possible to divide a d-dimensional shape into two similarly shaped parts. It is, however, possible to divide a d-dimensional shape into 2^d similarly shaped parts by geometrically bisecting each dimension. This data structure is similar to quad and oct trees, the difference being that only a single box is recursively divided at each level (see figure 2).

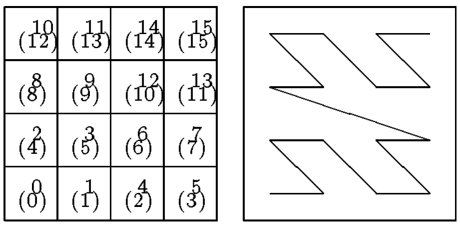

Figure 4: 2-dimensional shuffled row-major numbering and its fractal.

Related to the above parts problem is self-similarity, which means that the shape of a part is the same as the shape of the whole. When applied to a decreasing-size fractiling scheme, it means that the shape of the subtile is the same for all sizes, and when consecutive subtiles are combined, they form a larger similarly shaped subtile. Self-similarity is important so that the work units that have been executed in a tile do not depend on the particular sizes of the subtiles that have been executed—only on the total number of units that have been executed (see figure 3). Thus, the iterates in each tile can be allocated using a single counter (as described below), and a complex history of executed subtiles need not be maintained.

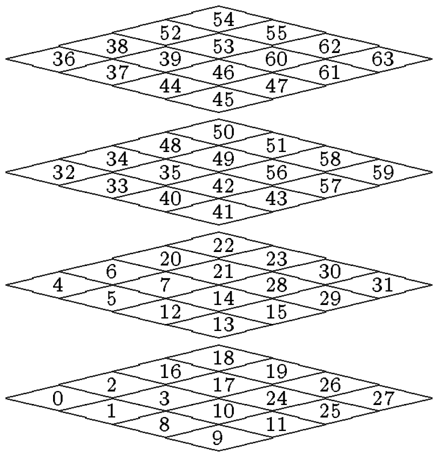

Figure 5: 3-dimensional shuffled row-major order numbering.

Fractiling uses the fractal shuffled row-major numbering [36] to allocate tiles. A shuffle operation maps a number with bits $[n_7, n_6, n_5, n_4, n_3, n_2, n_1, n_0]$ to $[n_7, n_3, n_6, n_2, n_5, n_1, n_4, n_0]$, i.e., it "shuffles" the upper and lower halves, just as one shuffles the two halves of a deck of cards. An unshuffle is the inverse operation. To map from a position on the shuffled row-major curve to one on the row-major curve (see figure 4), its x and y coordinates are concatenated (coalesced [28]) and unshuffled, and the result is split into two halves (de-coalesced) yielding the new x and y coordinates.

Shuffled ordering naturally extends to higher dimensions: the bits are divided into d equal-size segments and a d-way shuffle performed (see figure 5). To use the card analogy, a deck is divided into d stacks and then shuffled back together by interleaving their cards. So, for instance, a four-way shuffle of $[n_7, n_6, n_5, n_4, n_3, n_2, n_1, n_0]$ yields $[n_7, n_5, n_3, n_1, n_6, n_4, n_2, n_0]$.

To allocate subtiles in shuffled row-major order, the bits of a per-tile executed-iterate counter are divided into d segments and d-way unshuffled. The unshuffled integer is divided into d segments yielding coordinates in the d-dimensional parallelepiped. With shuffled row-major order, the lower bits are spread out evenly in all dimensions, so each grows at the same rate (see figure 5). Unshuffle operations can be efficiently implemented using table-lookups, shifts and masks [12].

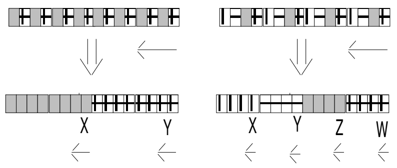

Figure 6: All dimensions grow at the same rate with shuffle row-major order.

4 N-Body Implementation on the KSR1

To determine the effectiveness of our scheme for applying fractiling to computations on irregular data, we implemented parallel and fractiled N-Body simulations on a KSR1 [22]. The KSR1 is a hardware-coherent cache-only multiprocessor. Its topology is a ring of rings, with processors distributed on the outer rings. The KSR1 used in our experiments has 128 processors distributed over 4 outer rings (32 processors per ring). The top level ring has a bandwidth of 4GB/sec., and the outer ones 1GB/sec. In the KSR there is no main memory in the conventional sense. Instead, each processor has a 32MB local cache and a 0.5MB split instruction and data subcache. The cache line (page) is 16KB, and the subcache cache line (subpage) is 128 bytes. A snoopy invalidate protocol is used to keep caches coherent. The relative data access latencies for local subcache, local cache, non-local cache same ring, and non-local cache different ring, are (2:18:150:500).

An advantage of the KSR1 architecture is that it was relatively easy to port a serial version of the N-body simulation code. A disadvantage is that data access latency depends critically on its location, which is not easy to control, so performance tuning the parallel code was difficult.

Calls to KSR Pthreads package to specify and coordinate parallelism were inserted into a serial implementation of Greengard's Fast Multipole Algorithm (FMA) for two-dimensional N-Body simulations [35]. The Greengard algorithm has two passes over the quad-tree per time step. During the upward pass, the summary of field effects of bodies in the subtress are propagated up the tree, and during the downward pass, the local expansions and direct interactions are computed.

`Pthread_create` was used to spawn a thread for each processor. Each thread was bound to a processor in the same ring using `pthread_move`. Barrier synchronization between the upward and downward passes was implemented using `pthread_barrier_checkin` and `pthread_barrier_checkout`. An additional barrier is needed when changing the number of processors, e.g., when moving from a level with 16 boxes to one with 4 boxes. In the fractiled version, although not strictly necessary, a barrier was inserted between each level in the downward pass to synchronize the re-initialization of data structures. The fractiling implementation used `pthread_mutex_lock` and `pthread_mutex_unlock` to coordinate the assignment of subtiles.

In the implementation, space is discretized into P tiles, and each tile is assigned to a processor. The downward pass was selected to be fractiled since it is the most computationally intensive step in the FMA. The tiles at each level in the downward pass are dynamically fractiled in shuffle-row major order using per-processor counters. The counters are doubly linked so that a processor that completes its tile can remove its counter and search for new work. Accesses to the linked data structure are protected by a mutex. To prevent subtiles from "ping ponging" between caches, they are aligned on cache-line boundaries.

5 Experimental Results

To test the effectiveness of fractiling, we ran the fractiled version and nonfractiled version of the parallelized FMA on uniform and nonuniform distributions. Both programs were run on from 4 to 32 processors. Neither program ran faster on 64 processors than it did on 32. There was considerable variance in the parallel execution times, so we ran each program ten times on each distribution. Below, we give the minimum, average and maximum timings obtained (figures 7, 8, and 9).

The uniform distributions had 2K bodies, which resulted in a 6-level quad tree, with a maximum of 4 bodies per box. The nonuniform distribution, called corner, was created by overlapping a Gaussian distribution, shifted towards one corner, over a uniform distribution with 1 body per box on the floor of the main computation space. The distribution had about 19K particles, which resulted in a 7-level quad tree, with a maximum of 27 bodies ber box. The results demonstrate that fractiling can improve performance for both uniform

Test case ↘ No. Procs	1	4	8	16	32
PFMA Uniform	29.5	10.2	8.5	8.0	8.0
Fract. Uniform		10.1	6.1	6.0	7.1
PFMA Corner	170.8	95.0	52.0	32.4	28.4
Fract. Corner		54.5	30.9	23.1	23.7

Figure 7: Minimum Execution Times in Seconds.

and non-uniform distributions. In all cases, fractiling consistently outperformed the non-fractiled code. The fractiled version also scaled (slightly) better than the non-fractiled one. Although neither version is scaling well, as demonstrated in their cost ($= P * T_P$) graphs (figures 10 and 11).

The variability in the parallel execution times underscores the need for a load balancing scheme that accommodates system-induced variance, and suggests that additional performance improvements can be obtained by a more careful orchestration of data movement. There are several ways to improve the scalability of the fractiled version, for example, by removing the barriers between each level of the downward pass and reducing the time spent in the mutex.

Test case ↘ No. Procs	1	4	8	16	32
PFMA Uniform	29.6	10.8	8.8	8.6	8.8
Fract. Uniform		10.8	6.5	6.8	8.1
PFMA Corner	172.4	95.6	52.3	32.9	29.1
Fract. Corner		54.9	31.3	23.5	24.6

Figure 8: Average Execution Times in Seconds.

Test case ↘ No. Procs	1	4	8	16	32
PFMA Uniform	29.8	11.3	9.1	9.4	9.8
Fract. Uniform		11.3	6.8	7.3	9.0
PFMA Corner	174.5	96.2	52.5	33.5	29.7
Fract. Corner		55.5	31.7	23.9	25.1

Figure 9: Maximum Execution Times in Seconds.

6 Conclusions and Future Work

Previous schemes to load balance computations on irregular data, such as N-body simulations and finite element analysis, have primarily addressed this algorithmic variability. However, experience has shown that unpredictable system interference causes considerable

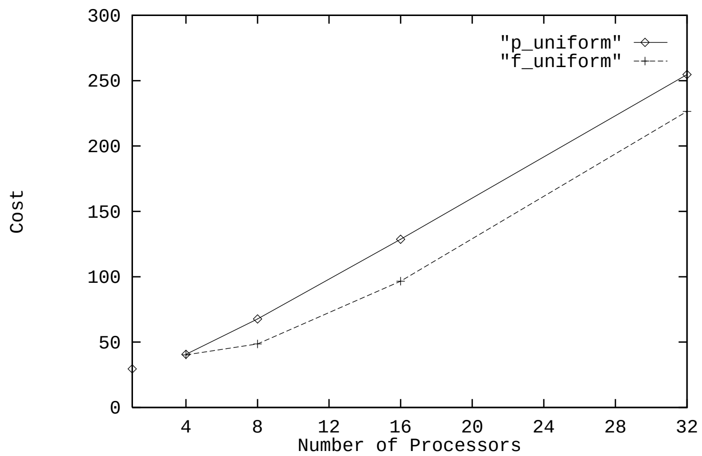

Figure 10: Cost of the Uniform Distribution.

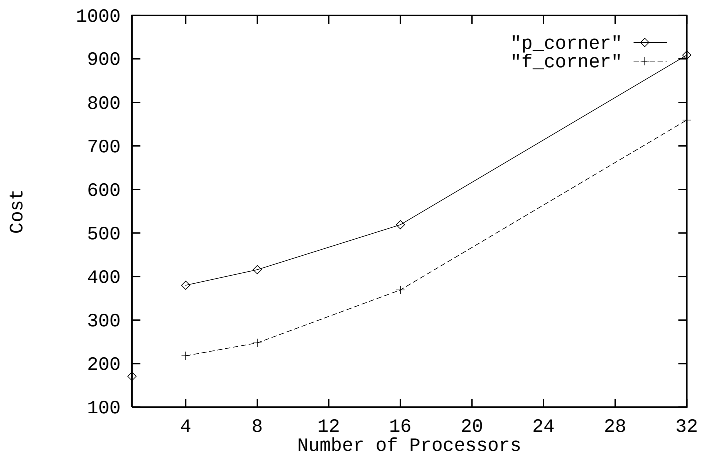

Figure 11: Cost of the Corner Distribution.

variance in execution times on parallel machines. Moreover, so that the load does not become imbalanced as bodies move over time, in these schemes, load information has to be gathered and data structures updated during execution.

Our approach to load balancing computations on irregular data is to dynamically schedule work in decreasing sizes. To exploit locality, work is initially divided among the processors. Processors that finish their work early borrow work from slower processors to promote load balancing. The sizes of the work units scheduled are chosen so that they have a high probability of finishing before the optimal time. The self-similar shapes of the work units increases locality. As the scheme dynamically adjusts to load imbalances, there is no need to gather load balancing information. Our experiments on the KSR1 using uniform and nonuniform distributions of bodies, wherein the fractiled version of a parallelized FMA consistently outperformed a non-fractiled one, illustrate the need for a load balancing scheme that accommodates both algorithmic and system induced variance.

Because of the regularity of the fractiling domain decomposition, we expect it to not be difficult to apply fractiling to 3-dimensional N-body simulations. In conclusion, fractiling is promising approach to load balancing computations on irregular data, such as the N-body problem.

Acknoledgements

We thank Surendra Talla for providing us with his sequential FMA code and discussions on N-body simulations, Ed Schonberg for suggesting that N-body simulations would be a good case study for fractiling, and Leslie Greengard for clarifications on details of the FMA.

References

[1] A. W. Appel, An Efficient Program for Many-Body Simulations, *SIAM Journal of Computing* **6**, 1985.

[2] J. Barnes and P. Hutt, A Hierarchical $O(NlogN)$ Force Calculation Algorithm, *Nature* **324**, 1986.

[3] M. Berger and S. Bokhari, A Partitioning Strategy for Nonuniform Problems on Multiprocessors, *IEEE Trans. on Computers* **C-36** pp. 570–580, 1987.

[4] J. A. Board, J. W. Causey, J. F. Leathrum, A. Windemuth and K. Schulten, Accelerated Molecular Dynamic Simulations with the Parallel Fast Multipole Algorithm, *Chem. Phys. Let.* **198**, 1992.

[5] L. Carter, J. Ferrante and S. Flynn Hummel, Efficient Parallelism via Hierarchical Tiling, to appear in *Proc. of SIAM Conf. on Parallel Processing for Scientific Computing*, 1995.

[6] L. Carter, J. Ferrante and S. Flynn Hummel, Hierarchical Tiling: A Framework for Multi-level Parallelism and Locality, to appear *Int. Parallel Processing Symp.*, 1995.

[7] M.D. Durand, T. Montaut, L. Kervella and W. Jalby, Impact of Memory Contention on Task Duration in Self-Scheduled Programs, *Int. Conf. on Parallel Processing* I258-I262, Aug. 1993.

[8] L. E. Flynn and S. Flynn Hummel, The Mathematical Foundations of the Factoring Scheduling Method, with L. E. Flynn, IBM Research Report RC18462, Oct. 1992.

[9] S. Flynn Hummel, E. Schonberg, and L. E. Flynn, Factoring: A Practical and Robust Method for Scheduling Parallel Loops, *Communications of the ACM* **35(8)** pp. 90-101, Aug. 1992.

[10] S. Flynn Hummel and E. Schonberg, Low-Overhead Scheduling of Nested Parallelism, *IBM Journal of Research and Development* **35(5/6)** pp. 743-765, Nov. 1991.

[11] S. Flynn Hummel, D. Kimelman, E. Schonberg, Experience with Program Visualization in Tuning Parallel Loop Scheduling, *Proc. of HICSS-25* pp. II275-II286, Jan. 1992.

[12] S. Flynn Hummel, Fractiling: A Method for Scheduling Parallel Loops on NUMA Machines, IBM Research Report RC18958, June 1993.

[13] S. Flynn Hummel, C. Wang, and J. Wein, Simulations of Fractiling in the logP Model, unpublished manuscript, 1994.

[14] A. Y. Grama, V. Kumar, A. Sameh, Scalable Parallel Formulations of Barnes-Hut Method for N-Body Simulations, *Proc. of Supercomputing '94* pp. 439-448, Nov. 1994.

[15] D. C. Gray, Load Balancing the Parallel Fast Multipole Algorithm, Duke University Tech Report *94-003*, Apr. 1994.

[16] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, ACM Press, 1987.

[17] L. Greengard and W. Gropp, A Parallel Version of the Fast Multipole Algorithm, *Scientific Computing* pp. 213-222, 1987.

[18] L. Greengard and V. Rokhlin, A Fast Multipole Algorithm for Particle Simulations, *J. Comp. Physics* **73** pp. 325-348, 1987.

[19] M. Gupta and P. Banerjee, Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers, *IEEE Transactions on Parallel and Distributed Systems*, **3(2)** pp. 179-193, Mar. 1992.

[20] F. Irigoin and R. Triolet, Supernode Partitioning, *Proc. 15th ACM Symp. Principles of Programming Languages* pp. 319-329, Jan. 1988.

[21] C. Kruskal and A. Weiss, Allocating Independent Subtasks on Parallel Processors, IEEE Trans. Software Eng. **SE-11(10)** pp. 1001-1016, Oct. 1985.

[22] S. Frank, H. Burkhardt and J. Rothnie, The KSR1: Bridging the Gap between Shared Memory and MMPs, *Proc. Compcon '93*.

[23] D. Kulkarni, K. G. Kumar, A. Basu, and A. Paulraj, Loop Partitioning for Distributed Memory Multiprocessors as Unimodular Transforms, *Proc. of the ACM International Conference on Supercomputing* pp. 206-215, June 1991.

[24] F. F. Lee, Partitioning of Regular Computation on Multiprocessor Systems, *J. of Parallel and Distributed Computing* **9** pp. 312-317, 1990.

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

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

[27] G. M. Morton, A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, IBM, Ottowa, Canada, 1966.

[28] C. Polychronopoulos, Loop Coalescing: A Compiler Transformation for Parallel Machines, *Int. Conf. on Parallel Processing* pp. 235–242, 1987.

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

[30] J. Ramanujan and P. Sadayappan, Tiling of Iteration Spaces for Multicomputers, *Proc. Int. Conf. on Parallel Processing* pp. II179-II186, 1990.

[31] D. A. Reed, L. M. Adams, and M. L. Patrick, Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systemn, *IEEE Trans. on Computers* **c-36(7)** pp.845-858, July 1987.

[32] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berrymen, Run-time Execution of Loops on Message Passing Machines, *J. of Parallel and Distributed Computing* **8** pp. 303-312, 1990.

[33] K. E. Schmidt and M. A. Lee, Implementing the Fast Multipole Method in Parallel, *J. Stat. Phy.* **63:1120**, 1991.

[34] J. Singh, C. Hotl, T. Totsuka, A. Gupta, and J. Hennesy, A Parallel Adaptive Fast Multipole Algorithm, *Proc. of Supercomputing '93* pp. 54-65, 1993.

[35] S. Talla, C implementation of Greengard's FMA, email, June 1994.

[36] C. D. Thompson and H. T. Kung, Sorting on a Mesh-Connected Parallel Computer, *Comm. of the ACM* **20(4)** pp. 263–271, 1977.

[37] T. H. Tzen and L. M. Ni, Dynamic Loop Scheduling for Shared-Memory Multiprocessors, *Proc. Int. Conf. on Parallel Processing* pp. II247–II250, 1991.

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

[39] M. Warren and J. Salmon, A Parallel Hashed Oct Tree N-Body Algorithm, *Proc. of Supercomputing '93*, 1993.

[40] M. E. Wolf and M. S. Lam, A Data Locality Algorithm, *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* pp. 30-44, June 1991.