An Algorithm for Load Balancing Highly Irregular Computations *

Ioana Banicescu and Vijay Velusamy
Department of Computer Science
NSF Engineering Research Center for Computational Systems
Mississippi State University
E-mail: {ioana@cs, vijay@erc}.msstate.edu

Abstract

In heterogeneous environments, dynamic scheduling algorithms are a powerful tool towards performance improvement of scientific applications via load balancing. In general, these techniques employ heuristics that require a priori knowledge about workload via profiling resulting in higher overhead as problem sizes and number of processors increase. In addition, variations in work load may appear only at run-time, making profiling work tedious and sometimes even obsolete. Recently, dynamic loop scheduling schemes such as Factoring, Fractiling, and Weighted Factoring proved to be extremely instrumental in scientific applications such as Monte-Carlo simulations, N-Body simulations, radar applications, and others. This paper reports on performance improvements obtained by integrating a dynamic loop scheduling technique that evolves from earlier schemes, and addresses these concerns, the Adaptive Factoring, into a scientific application that invloves computational field simulation on unstructured grids. Reported experimental results confirm the benefits of using this methodology, and emphasize its high potential for a successful integration in other scientific applications that exhibit substantial performance degradation due to load imbalance.

1. Introduction

In large scientific applications performance is significantly affected by the variance in workload, variance in processor performance during execution, variance in network latency, and other systemic factors. Processor imbalances might be generated by application and algorithmic effects, as well as system effects, such as data access latency and operating system interference. In heterogeneous environments such as network of workstations (NOW), clusters of NOW, and clusters of SMPs, the potential of these imbalances becoming performance bottlenecks increases, since various applications might also unexpectedly be initiated or removed. Dynamic scheduling schemes attempt to maintain balanced loads by assigning work to idle processors at run time. In this way, they accommodate systemic as well as algorithmic variances. Adapting to the variance of multiple factors, such as the ones described above, requires dynamic work assignment. In heterogeneous environments, dynamic scheduling algorithms are a powerful tool towards performance improvement of scientific applications via load balancing. Traditionally, these scheduling techniques employ heuristics that require prior knowledge about workload via profiling, and load balancing is achieved via repetitive or iterative static partitioning [36][12][32][34][40]. This approach results in higher overhead as problem sizes and number of processors increase. However, load imbalance may appear only at run-time, making profiling work tedious and sometimes even obsolete. A number of newly developed dynamic strategies to address the above concerns have been proposed and often incorporated into software libraries and packages [24][8]. They address load imbalances on partitions due to application and architectural irregularities that are known, or could be predicted, and omit the ones that may occur during execution (i.e., due to variations in network latency and load). These concerns raise the need for a fully dynamic load balancing approach that addresses application and system irregularities at run-time.

Dynamic loop scheduling schemes such as Factoring, Fractiling, Weighted Factoring, and Adaptive Weighted Factoring have recently been proposed and successfully implemented in the parallelization of, Monte-Carlo simulations [21], N-body simulations [3][5], radar applications[18], and computational fluid dynamics [6], respectively. These schemes are based on a probabilistic analysis, and therefore can accommodate load imbalances caused by predictable and unpredictable phenomena. The performance of computational simulation codes over other competitive techniques was con-

*This work was partially supported by the National Science Foundation Grants: NSF CAREER Award # ACI9984465, NSF ITR/ACS Award # ACI0081303, and Award # EEC-9730381.

siderably improved in both, distributed shared address space environments [3][21], and distributed message passing environments [5][18]. The algorithms employed derive from recent theoretical advances in research on scheduling parallel loop iterations with variable running times [22][28][27][30][29][25][37][38][19][20]. These loop scheduling techniques allow loop iterations to be executed in decreasing size chunks. The chunk-sizes are selected such that they have a high probability of being completed by the processors before the optimal time.

Adaptive Factoring is an algorithm that evolves from the above methodology and its mathematical foundations have been presented in [4] An integration of this algorithm in an implementation of a parallel computational field simulation application using unstructured grids of various sizes, and experimental results on SuperMSPARC (at the National Science Foundation Engineering Research Center for Computational Systems - NSF ERC - at Mississippi State University) are reported here. The benefits of using this technique in computational field simulation problems using unstructured grids over other earlier developed load balancing techniques are confirmed by experimental work, and underscore the advantages of using this technique to improve the performance of other scientific applications. The results indicate a high potential for future integration of this method in applications involving complex and irregular domains where fluctuations in number of data points and in workload per data point cannot be predicted from the domain structure only, and simple static evaluation of workloads.

The paper is organized as follows. Section §2 describes recent advances in scheduling techniques for load balancing scientific applications with emphasis on the development of loop scheduling strategies over time. Section §3 gives an overview of the adaptive factoring method and a description of integrating this technique into a computational field simulation application using unstructured grids. Section §4 discusses experimental results and the scalability of current implementation with respect to other competitive techniques. Section §5 presents the benefits of extending the use of this technique for an effective parallelization of other scientific applications in heterogeneous environments.

2. Related Work

Research on load balancing methods for scientific computations on homogeneous as well as heterogeneous systems has been extensively surveyed in the literature [11][39][13][33]. In general, in scientific applications that use finite element and finite volume methods, scheduling techniques use the most recent advances in graph partitioning algorithms followed by static scheduling, to address performance degradation due to load imbalance. To adapt to variable work loads from one step to another, some applications use an iterative static approach, where for each repartitioning step the computation on each partition segment is performed until completion [9][26][34][40]. These techniques are effective in applications where the fluctuations in the number of data points and the workload per data point can be well predicted from a prior evaluation of the computation space. Some of these techniques have been gathered into software packages widely used, which include a number of algorithms that offer the user choices regarding preferences in performance tradeoffs, related to the characteristics of their particular applications. However, in some applications in addition to changes in number of points in various regions of the computation space, the amount of work per data point cannot be anticipated from a simple static analysis. For these applications, load balancing strategies that use an iterative static approach are not effective.

A number of newly developed dynamic strategies attempt to address the above concerns by providing tuning parameters to address both application and architectural characteristics [24] and [8]. However, they only address application and architectural irregularities that are known, or could be predicted prior to starting the execution of a time step [8][24][36][12][32]. These strategies also fail to address partition irregularities that occur during the execution (for instance, “within” a time step of the computation as opposed to “between” time steps).

In other scientific applications, such as N-body simulations [7][2][15][1] performance gains from the parallel execution [16][23][17] are also difficult to obtain due to load imbalance. The imbalance may be caused by the irregularity of data distribution (particles) and by the different processing requirements of data points in interior versus near the computation space boundary. In addition, the distribution of particles varies at each time step.

Previously, various methods have been employed to balance processor loads, as well as to exploit locality in N-Body simulations. They only address algorithmic variances, and use profiling by gathering information on the work load from a previous time step in order to estimate the optimal assignment of work to be distributed during the current time step. However, the cost of these methods (i.e., orthogonal recursive bisection, cost zones methods, hashed oct-tree, random assignments) increases with the number of processors and data size [35][41][14], and they employ a static assignment of work load to processors during a time step. Unfortunately, for large problems, load imbalances often occur during a time step, since processor load imbalances are induced not only by application and algorithmic effects, but also by system effects (i.e., data access latency and operating system interference). Adapting to algorithmic and system induced load imbalances requires dynamic work assignment. Dynamic scheduling schemes attempt to maintain balanced loads by assigning work to idle processors at

run time. In this way, they accommodate systemic as well as algorithmic variances.

Scientific applications contain loops with large number of iterations which could be expressed independently, thus easily amenable for parallelization. In applications such as computational fluid dynamics, Monte Carlo simulations, sparse matrix computations, iterations execution times could vary due to, for instance, conditional statements. Even in applications where there is no algorithmic variance in iteration lengths, loop iteration execution times may vary due to interference from other iterations, other applications, or the operating system. The cumulative effect of variances in loop iteration execution times could ultimately lead to processors load imbalance and therefore to severe performance degradation of parallel applications. A fundamental trade-off in scheduling parallel loop iterations is that of balancing processor loads while minimizing scheduling overhead.

In homogeneous systems, in addition to problems encountered due to application irregularities, the difference in processor speeds, architectures and memory capacities can significantly impact performance. Moreover, in case of heterogeneous networks of workstations, loads and availability of the workstations are unpredictable, and thus it is difficult to know in advance what the effective speed of each machine will be.

Algorithms that derive from theoretical advances in research on scheduling parallel loop iterations with variable running times [22][28] [25][38][21], have extensively been studied for dynamically load balancing scientific applications. Recently, dynamic loop scheduling algorithms based on a probabilistic analysis have been proposed, and successfully implemented for a number of scientific applications [21][18][3][5]. In Monte-Carlo simulations, N-body simulations, and Radar applications, dynamic scheduling schemes, based on *Factoring* [21], have been proposed. They derive from recent advances in research on scheduling parallel loops with variable running times, and accommodate load imbalances caused by predictable phenomena, such as irregular data, as well as unpredictable phenomena, such as data-access latency and operating system interference. In these algorithms, loop iterations are executed in decreasing size chunks, so that earlier larger chunks have relatively little overhead, and their unevenness can be smoothed over by later smaller chunks. The selection of chunk sizes requires that they have a high probability of being completed by the processors before the optimal time. These schemes allow the scheduled batches of chunks of iterations to be fixed portions of those remaining (each batch contains P chunks, where P is the number of processors).

One of the methods based on Factoring is *Fractiling*, a combined scheduling technique that balances processor loads and maintains locality by exploiting self-similarity properties of fractals. It has successfully been implemented for N-Body simulations in distributed shared address space, as well as message passing environments [3][5].

As the heterogeneity in processors performance could lead to severe load imbalance, a *Weighted Factoring* approach was proposed, where the decreasing chunks sizes are proportional to the relative processor speeds [18]. Experiments involving networks of workstations have shown that weighted factoring significantly outperformed previous methods as well as *work-stealing*, a scheme in which work is dynamically migrated from heavily loaded processors to lightly loaded ones [10][31]. In environments where processor workloads dynamically vary, many scientific applications whose solutions require a number of iterations over the computation space would benefit from a dynamic adjustment of weights after each iteration. To address this concern, the *Adaptive Weighted Factoring* technique has been proposed, successfully implemented, and reported in [6]. In weighted factoring and adaptive weighted factoring, weights are statically assigned to processors, and are considered to remain unchanged either throughout the entire computation, or throughout running the algorithm for one iteration over the computation space, respectively. This characteristic results in performance limitations of applications where highly unpredictable imbalances occur at runtime. The following section describes the *Adaptive Factoring* scheme, where this limitation is addressed, and the dynamic assignment of work for each processor closely follows its rate of change in load.

3 Adaptive Factoring and its Implementation

The effectiveness of scheduling parallel loops with independent iterations on shared address-space homogeneous multiprocessors, as well as on networks of heterogeneous workstations, has received considerable attention in the scientific community. In general, the schemes deriving from Factoring and presented in Section §2 assume that loop iterations' execution times are independent random variables, and that their mean and standard deviation are known and do not change during application execution. In real applications, these assumptions do not hold, since the loop iterations' execution times may vary, due to both application and system irregularities that may occur during runtime. A new, more generalized technique for scheduling parallel loops with independent iterations, *Adaptive Factoring*, has recently been introduced, and the mathematical foundations have earlier been presented in [4]. A theoretical study was also conducted under a more generalized assumption that values of mean and standard deviation of loop iterations' execution times are unknown and vary during runtime. An analysis of work assignment has also been presented. The *Adaptive Factoring* algorithm dynamically esti-

mates the means and variances of loop iterations' execution times, and employs a probabilistic and statistical model for the dynamic allocation of chunks of loop iterations for each processor. The sizes of chunks of loop iterations are dynamically computed and allocated to processors such that the average finishing time for completion of all chunks occurs before the optimal time with high probability. This insures a more efficient method for balancing processor workloads, highly tuned to the rate of change in processor speeds.

The optimal time can be achieved only if the loop iterations' execution times are nonrandom. We adopt a criteria for scheduling loop iterations such that the expected execution time of a batch is less than the optimal time for all the remaining loop iterations to be executed. This criteria is applied to different situations where factoring scheduling methods are used to balance processors' workloads (i.e., equal versus weighted processor speeds, known versus unknown means and standard deviations of iteration execution times) [4].

To study the performance of scheduling scientific applications with the *Adaptive Factoring* (AF), the solution of heat conduction equation on an unstructured grid was chosen. The heat solver computes the solution of the steady heat conduction equation (Laplace equation), using an iterative method (Jacobi method). During each algorithms' iteration over the computation space, the temperature norm value is computed by solving the equation for all the interior points on the grid, and is compared with a given tolerance value.

The adaptive factoring scheme can be implemented in distributed computing environment using three approaches: a centralized management, a distributed management or a hierarchical management approach. Our current implementation employs the centralized management approach, which uses a master processor and several slave processors. Other possible implementations that are presently sought are beyond the scope of this paper. Master/slave communication patterns in our implementations are similar to the ones described in detail in [5]. In this scheme, only the master has the authority to access and modify shared variables (such as, chunk sizes to dynamically be assigned to slaves, and the the total amount of remaining work to be computed by the system). Thus, this scheme guarantees data consistency and less complexity of programming. Each slave computes updated values for the means and the standard deviations of loop iterations of recently finished chunks, and send this information to the master. The master then computes the chunk sizes to next dynamically be assigned to each slave. All updated values are computed according to the algorithm described in detail in [4]. Performance bottlenecks may thus occur with increasing number of slaves being controlled by a single master. This problem can be alleviated by a hierarchical management scheme, consisting of multiple masters, each controlling a number of slaves.

The adaptive factoring algorithm was integrated into a parallel implementation of the heat solver in C++ using the Message Passing Interface (MPI). Previously, four other implementations of the heat solver using earlier developed loop scheduling techniques, such as Static Scheduling (SS), Weighted Static Scheduling (WSS), Factoring (FAC) and Adaptive Weighted Factoring (AWF), have also been reported [6]. This paper considers evaluating the adaptive factoring implementation by comparing its performance with that of each of the above mentioned techniques.

4 Experimental Results

The performance of the parallel implementation using the Adaptive Factoring (AF) for an application using unstructured grids is evaluated from collected results of running experiments using parallel implementations of the heat solver with Static Scheduling (SS), Weighted Static Scheduling (WSS), Factoring (FAC), Adaptive Weighted Factoring (AWF), and Adaptive Factoring (AF) on SuperMSPARC. The SuperMSPARC is a 32-processor cluster comprised of eight 4-processor clusters (designed and constructed at the NSF ERC at Mississippi State University). The clusters are essentially tightly coupled Sun SparcStation 10s with CPU upgrades. The clusters are interconnected using ATM or Myrinet networks, characterized by low latency and high bandwidth. Timing results obtained from five executions of each implementation on a particular environment (characterized by problem size and number of processors), were collected and averaged (to avoid measurement biases) for further analysis.

The implementations of AF, AWF, FAC, WSS and SS were run on 2, 4, 8, 16 and 32 processors of SuperMSPARC, in a loaded environment where external jobs were also executed along with the heat solver (in order to introduce variable loads on the processors). Grid sizes of 78K, 101K, 134K and 158K points were used for these experiments.

In order to simulate external loads on the processors which are used to run our applications on, a 'slowburn program' is executed on processors according to the load percentages mentioned below. The slowburn program is essentially a load simulator that uses CPU cycles up to an amount that can easily be specified by the user on the command line. This load simulator is designed to be separate from the application, such that the actual load on the CPU could more accurately be accounted for our analysis, and therefore reused for analyzing the performance of our scheduling techniques in other applications. Half of the processors were externally loaded by running a program that loads the CPU to a preset value. Half of these loaded processors were loaded with medium load and the rest with high load. For example, when the application was executed on 32 proces-

sors, 8 of them were loaded with 25% external load and 8 of them with 50%.

The elapsed time between the moment the computation in parallel started, until the moment the last processor finished, was measured for the purpose of analysis. The values of costs (pT_p), improvement percentage in cost, and coefficients of variation (c.o.v.'s) of processor finishing times, were calculated to measure the performance of all the scheduling techniques mentioned above.

Experimental results indicate that the AF costs are in most cases lower than the ones of AWF, FAC, WSS, and always lower than the ones of SS. In most cases, both AF and AWF show substantial reduction in cost against the other techniques. In case of the grid with sample size of 78K points, values of cost improvements of up to 8%, 14%, and 25% over AWF, FAC, and WSS, were respectively obtained, underscoring the benefits of AF (see Figure 1., Tables 1-4.).

With grid sample size of 101K points, the costs of AF, AWF, and FAC are consistently better than those of the other techniques. Cost improvements of up to 4%, 17%, and 32%, have been obtained by AF over AWF, FAC, and WSS, respectively. However, for smaller number of processors (less than 8), AWF and FAC outperformed AF by up to 13% (see Figure 2., Tables 1-4.).

With grid sample size of 134K points and larger number of processors (greater than 8), the cost of AF is always better than the costs of other techniques. Cost improvements of up to 8%, 10%, and 21% have been obtained by AF over AWF, FAC, and WSS, respectively (see Figure 3., Tables 1-4.).

With grid sample size of 158K points, the cost of AF is always better than any of the other techniques with only one exception, in which at 8 processors FAC outperforms AF by 1%. Cost improvements of up to 7%, 26%, and 48% have been obtained by AF over AWF, FAC, and WSS, respectively (see Figure 4., Tables 1-4.).

The experimental results clearly indicate that the AF scales with the number of processors and problem sizes. Its best cost improvement percentage over all the other techniques is in general achieved for the largest number of processors and the largest problem sizes used.

In all the experiments, the average values for the coefficients of variation (c.o.v.) of times have been computed. The average c.o.v. values of AF, AWF, and FAC obtained are much lower than those of SS and WSS, for all problems sizes and number of processors considered (see Figures 5, 6, 7, and 8). Moreover, in all cases, the c.o.v. values of AF outperformed the ones of AWF and FAC, underscoring the AF's strengths in dynamically balancing loads over other factoring methods.

5. Conclusions and Future Work

For improved performance in highly irregular and computationally intensive applications running in heterogeneous environments, it is essential that adequate parallel dynamic loop scheduling methods are used. Of particular interest are the classes of problems, such as computational field simulation on unstructured grids and N-body simulations, where load balancing strategies used by previous methods are not effective.

In this paper, the benefits of using the Adaptive Factoring (AF) method to improve the performance of an irregular application using unstructured grids is presented. The AF is a new dynamic loop scheduling algorithm, that allows a relaxation of some of the theoretical assumptions imposed by models used in earlier methods, therefore making this technique more robust to any load variations present in the software environment. In this way, performance of a larger class of parallel applications, whose load imbalance may be caused by a wider range of characteristics, can considerably be improved.

In our application, the AF proved to perform better than all the other methods employed (SS, WSS, FAC, and AWF) under all conditions (up to 48% cost improvement). This indicates that the performance gains due to load balancing with AF outweigh the increase in scheduling overhead. The average values of the coefficients of variation of processors' finishing times of the AF are consistently lower than those of the other techniques. Experiments with AF in N-body simulations are currently underway and will be reported in the future.

References

[1] C. Anderson. An Implementation of the Fast Multipole Method Without Multipoles. *SIAM J. Sci. Stat. Comput.*, 13(4):923–947, July 1992.

[2] A. W. Appel. An Efficient program for Many-Body Simulations. *SIAM Journal of computing*, 6, 1985.

[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 Z. Liu. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In *Proceedings of the High Performance Computing Symposium (HPC 2000)*, pages 122–129, 2000.

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

[6] I. Banicescu and V. Velusamy. Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring. In *Proceedings of the Heterogenous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium*, page to appear, 2001.

[7] J. Barnes and P. Hutt. A Hierarchical O(Nlog(N)) Force Calculation Algorithm. *Nature*, 324, 1986.

[8] R. Biswas, S. Das, D. Harvey, and L.Oliker. Portable Parallel Programming for Dynamic Load Balancing of Unstructured Grid Applications. In *Proceedings of 13th International Parallel Processing Symposium*, April 1999.

[9] R. Biswas and L. Oliker. Load Balancing Unstructured Adaptive Grids for CFD Problems. *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[11] T. Casavant and J. Kuh. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Software Eng SE-14(2)*, pages 141–154, Feb. 1988.

[12] J. Chen and V. Taylor. Mesh Partitioning for Distributed Systems: Exploring Optimal Number of Partitions with Local and Remote Communication. In *Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. Software Eng SE-12(5)*, pages 662–675, May 1996.

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

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

[16] L. Greengard and W. Gropp. A Parallel Version of the Fast Multipole Algorithm. *Scientific Computing*, pages 213–222, 1987.

[17] Y. Hu and S. L. Johnsson. Implementing O(N) N-body Algorithms Efficiently in Parallel Languages. *Journal of Scientific Programming*, 5(4):337–364, 1996.

[18] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of Symposium on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[19] S. F. Hummel and E. Schonberg. Low-overhead Scheduling of Nested Parallelism. *IBM Journal of Research and Development 35(3/6)*, pages 743–765, Sept/Nov 1991.

[20] S. F. Hummel, E. Schonberg, and L. E. Flynn. A Practical and Robust Method for Scheduling Parallel Loops. *1991 Supercomputing Conference*, pages 610–619, November 1991.

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

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

[23] J. F. Leathrum Jr. *Parallelization of the Fast Multipole Algorithm: Algorithm and Architecture Design*. PhD thesis, Duke University, 1992.

[24] B. Maerten, D. Roose, A. Basermann, and J. Fingberg. DRAMA: A Library for Parallel Dynamic Load Balancing of Finite Element Applications. In *Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[25] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *Proceedings of Supercomputing '92*, pages 104–113, Nov. 1992.

[26] L. Oliker and R. Biswas. Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations. In *Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 33–42, 1997.

[27] C. Polychronopoulos. Loop Coalescing: A Compiler Transformation for Parallel Machines. *Proc. ICPP*, pages 235–242, 1987.

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

[29] C. Polychronopoulos, D. J. Kuck, and D. A. Padua. Optimal Processor Allocation to Nested Parallel Loops. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 519–527, Aug. 1986.

[30] C. Polychronopoulos, D. J. Kuck, and D. A. Padua. Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds. *IEEE Trans. Computer.*, C-36, Apr. 1987.

[31] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[32] K. Schloegel, G. Karypis, and V. Kumar. Dynamic Repartitioning of Adaptively Refined Meshes. In *Proceedings of Supercomputing '98*, November 1998.

[33] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, pages 33–45, Dec. 1992.

[34] J. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, 1993.

[35] J. Singh, J. Hennessy, and A. Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *Computer*, pages 42–50, July 1993.

[36] A. Sohn and H. Simon. S-HARP: A Scalable Paralel Dynamic Partitioner for Adaptive Mesh-based Computations. In *Proceedings of Supercomputing '98*, November 1998.

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

[38] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers. *IEEE Trans. Parallel Distributed Syst.*, 4:87–98, Jan. 1993.

[39] Y. T. Wang and R. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, Mar. 1985.

[40] 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.

[41] 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.

Table 1. AF percentage improvement in cost over AWF

Improvement of	Grid Size	%	Procs.
AF over AWF	78K	-2.6	2
		4.6	4
		3.8	8
		5.3	16
		8.3	32
	101K	-13.0	2
		-2.6	4
		4.4	8
		-3.7	16
		3.9	32
	134K	5.8	2
		1.2	4
		7.9	8
		4.7	16
		3.5	32
	158K	7.2	2
		2.3	4
		1.4	8
		4.5	16
		4.8	32

Table 2. AF percentage improvement in cost over FAC

Improvement of	Grid Size	%	Procs.
AF over FAC	78K	2.0	2
		3.9	4
		6.9	8
		14.5	16
		14.2	32
	101K	-8.2	2
		-4.7	4
		16.7	8
		5.0	16
		7.0	32
	134K	4.0	2
		-3.4	4
		8.6	8
		10.0	16
		4.6	32
	158K	10.6	2
		7.4	4
		-1.5	8
		25.5	16
		25.8	32

Table 3. AF percentage improvement in cost over WSS

Improvement of	Grid Size	%	Procs.
AF over WSS	78K	9.9	2
		6.0	4
		8.3	8
		25.8	16
		14.9	32
	101K	18.7	2
		9.6	4
		32.5	8
		21.4	16
		13.4	32
	134K	-4.1	2
		2.0	4
		20.9	8
		9.5	16
		7.0	32
	158K	17.8	2
		4.8	4
		3.0	8
		27.6	16
		48.5	32

Table 4. AF percentage improvement in cost over SS

Improvement of	Grid Size	%	Procs.
AF over SS	78K	13.9	2
		12.1	4
		17.6	8
		17.2	16
		16.0	32
	101K	27.7	2
		0.6	4
		18.4	8
		17.9	16
		9.9	32
	134K	19.6	2
		11.1	4
		35.5	8
		21.9	16
		7.1	32
	158K	19.5	2
		12.9	4
		3.8	8
		23.5	16
		47.8	32

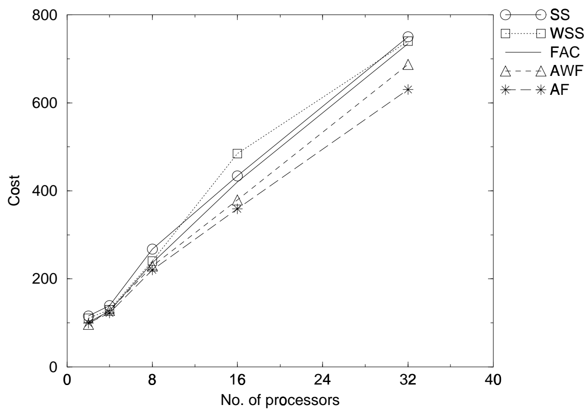

Figure 1. The cost (pXT_p) of SS, WSS, FAC, AWF and AF for a 78K size grid.

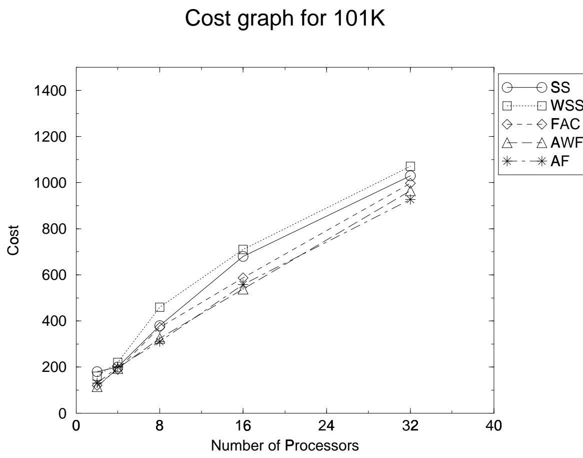

Figure 2. The cost (pXT_p) of SS, WSS, FAC, AWF and AF for a 101K size grid.

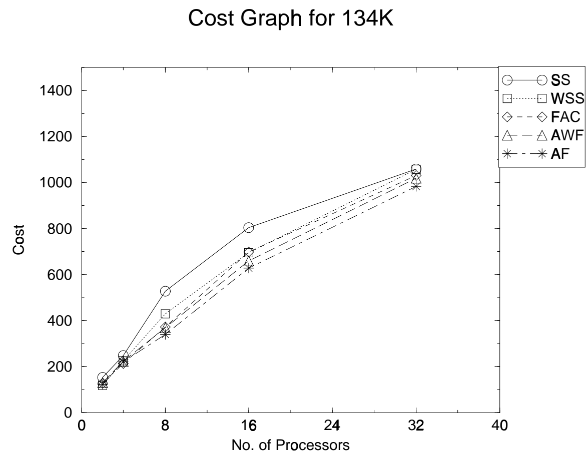

Figure 3. The cost (pXT_p) of SS, WSS, FAC, AWF and AF for a 134K size grid.

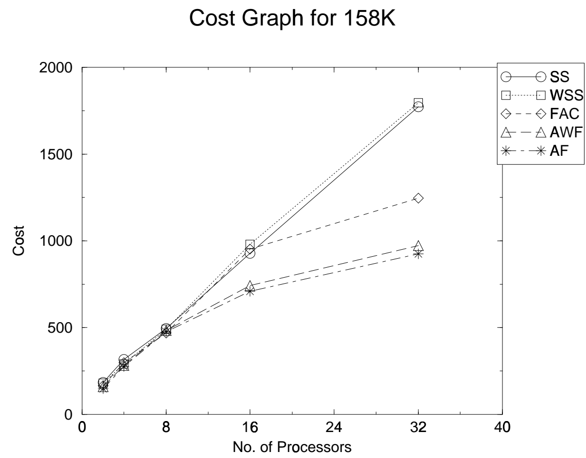

Figure 4. The cost (pXT_p) of SS, WSS, FAC, AWF and AF for a 158K size grid.

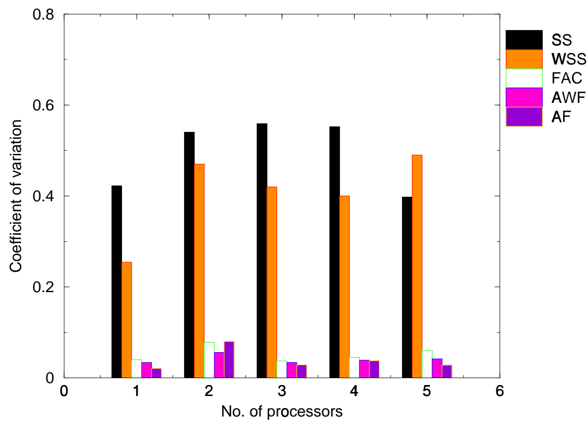

Figure 5. The average c.o.v. values of SS, WSS, FAC, AWF and AF for a 78K size grid.

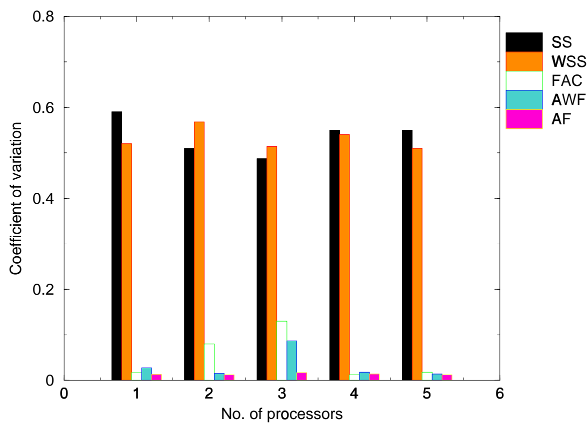

Figure 6. The average c.o.v. values of SS, WSS, FAC, AWF and AF for a 101K size grid.

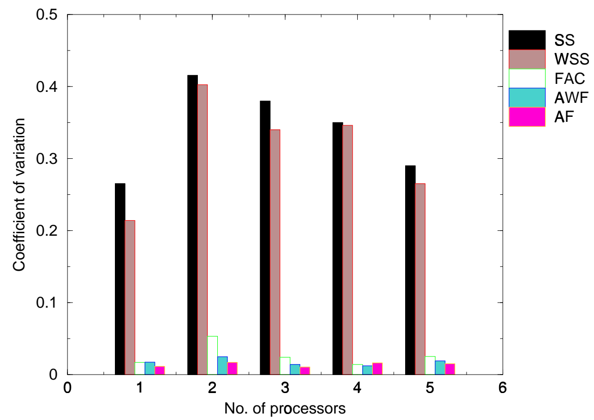

Figure 7. The average c.o.v. values of SS, WSS, FAC, AWF and AF for a 134K size grid.

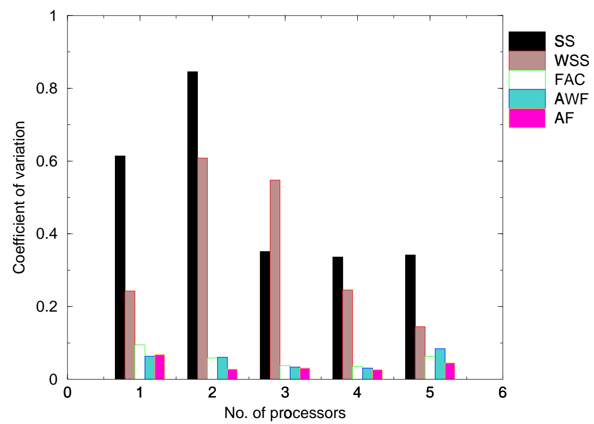

Figure 8. The average c.o.v. values of SS, WSS, FAC, AWF and AF for a 158K size grid.