Dynamic Load Balancing Algorithms for Embarrassingly Parallel Tasks

Jaderick P. Pabico
Institute of Computer Science
College of Arts and Sciences
University of the Philippines Los Baños
College 4031, Laguna, Philippines
63-49-536-2313
jppabico@uplb.edu.ph

ABSTRACT

Many scientific applications are considered embarrassingly parallel because they contain loops with large numbers of independent iterations that are computationally intensive. Given a loop with n iterations and p processors, the obvious scheduling is to assign n/p iterates per processor. However, the respective execution times of each iterate could vary due to conditional statements or variable amount of computation required. Even in applications where there is minimal code variation in each iterate, the iteration execution times may vary due to differences in effective processor speeds, latency in network traffic, latency in data access from primary and secondary memories, daemons waking up, or hardware interrupts. The cumulative effect of variances in iterate execution times could ultimately result to processor load imbalance and therefore to severe performance degradation of parallel applications at runtime. For effectively load balancing scientific parallel applications, algorithms for scheduling iterates with variable execution times have been studied such as fixed-size chunking, guided self scheduling, factoring, weighted factoring, adaptive weighted factoring, and adaptive factoring. In this paper, these algorithms are presented and compared in terms of two metrics over an increasing p, the parallel cost and the coefficient of variation of processor finishing times. The performance measurements were implemented using two scientific applications as test beds: The many-body simulation and the profiling of automatic quadrature routines.

Keywords

embarrassingly parallel tasks, parallel loops, dynamic load balancing

1. INTRODUCTION

A size-n problem is embarrassingly parallel if it is composed of n independent tasks, which when solved in parallel, the computation process achieves a speed-up of $\mathrm{O}(n)$ with very little inter-process communication. One such example of a size-n embarrassingly parallel problem is a loop with n independent iterations. The ith loop iterate is independent if it does not require the result of the computation of the $(i-1)$th iterate, and it does not cause any side effects to the other iterates. This loop can be solved by a sequential computer in $\Theta(n)$ time. However, since each loop iterate is independent, they can be computed in any order, or can be solved concurrently by a parallel computer. The loop can be solved by a n-processor parallel computer in $\Omega(1)$ time, achieving a speed-up of $\Theta(n)/\Omega(1) \approx n$. The relative speed-up S_p of any parallel problem under a p-processor computer is defined as the ratio between the execution time τ_1 of the problem when solved over a sequential computer with the execution time τ_p of the same problem when solved with a parallel computer with p processors (Equation 1). An absolute speed-up is computed if τ_1 is the execution time of the best sequential algorithm. The speed-up is ideal if it increases linearly by m when the problem size and the number of processors are both increased by m. An ideal S_p suggests scalability of the problem.

$$S_p = \frac{\tau_1}{\tau_p} \tag{1}$$

In real world situations, the ideal speed-up is never achieved, even for the most trivial embarrassingly parallel tasks. The reason for this is that in most scientific applications, where independent and parallelizable loops are prevalent, the loop iterates do not have the same execution time. Consider, for example, the loop shown in Figure 1. A loop iterate will have an execution time of either $\Theta(N \log N)$ by running line 6, or $\mathrm{O}(N)$ by running line 9, depending on the value of the variable X. In real world scientific applications, most iterates could have more than two different execution times, depending on the presence and structure of multiple branching or switching statements. For some applications with loop iterates of identical analytical execution times, the actual execution times could also vary significantly because of variabilities brought about by the underlying runtime system. These systemic variabilities are a combination of the differences in effective processor speeds, latency in network traffic, latency in data access from primary and secondary memories, I/O access latency, daemons waking up, hard-

Invited Paper to the 2009 Philippine Computing Science Congress
Silliman University, Dumaguete City, 2–3 March 2009

```
 1  N=10000000
 2  A=randomArray(1,N)
 3  for I:=1 to N do
 4    X:=random()
 5    if(X>0.5) then do
 6      quickSort(A)
 7    else
 8      y=random(1,N)
 9      sequentialSearch(A,y)
10    endif
11  next I
```

Figure 1: An example loop with iterates that could be executed in $\Theta(N \log N)$ or $\mathbf{O}(N)$ time, depending on the values of the variable X.

ware interrupts, and many others. Because of the expected various execution times of loop iterates, compounded by the variabilities brought about by the underlying runtime system, some processors might be assigned more computational tasks than others, resulting in poor performance for the parallel computer. When this happens, the parallel system is said to be *load imbalanced* with its realized speedup significantly lower than the expected theoretical one.

To effectively balance the load of the participating processors, algorithms for assigning tasks to processors at either compile time or runtime have been studied [23, 26, 24, 14, 20, 21, 10]. Those algorithms that assign tasks to processors at compile time are called static or offline scheduling. In these algorithms, the decision of which processor will work on a task has been decided before hand by the programmer. The most trivial offline scheduling decision for a given n-task problem on a p-processor machine, where realistically $p << n$, is to assign n/p tasks to each processor. For example, processor P_0 is assigned the tasks $T_0, T_1, T_2, \ldots, T_{n/p-1}$; Processor P_1 is assigned the tasks $T_{n/p}, T_{n/p+1}, T_{n/p+2}, \ldots, T_{2n/p-1}$; And processor P_{p-1} is assigned the tasks $T_i, \forall i = k, k+1, k+2, \ldots, n-1$ and $k = (p-1)n/p$. The obvious parallel execution time for this assignment is $\tau_p = \max_{i=P_0}^{P_{p-1}} \tau_i$. Performance degradation of the parallel system will happen if the actual distribution of task execution times is skewed. For example, if the distribution of the task execution times is skewed to the left, processors P_0, P_1, and P_2 will have more load than the rest. When this happens, processors $P_3, P_4, \ldots, P_{p-1}$ will finish early and will be idle while the first three processors are still computing. Since the assignment happened at compile time, the idle processors can not help in computing the load of the busy ones, increasing the realized τ_p and effectively reducing S_p. In the worst case, if the tasks are ordered in non-increasing way by execution time, then P_0 will process the longest time of

$$\tau_p = \sum_{i=0}^{n/p-1} \tau_i. \tag{2}$$

To solve the problems inherent in static scheduling algorithms, researchers thought of assigning tasks to processors at runtime. The algorithms that assign tasks to processors at runtime are also called dynamic or online scheduling algorithms. In these algorithms, the decision of which processor will work on a given task is decided by the program during runtime. Several dynamic scheduling algorithms have been developed and evaluated by researchers [19, 18, 5, 6, 7, 8]. These algorithms are fixed-size chunking (FSC), guided self scheduling (GSS), factoring (FACT), weighted factoring (WFACT), adaptive weighted factoring (AWF), and adaptive factoring (AF). All of these were incorporated in a dynamic load balancing library that has been recently developed, evaluated, and used by researchers [2, 4, 3, 15, 12]. In this paper, these algorithms will be presented and their performance compared in terms of two metrics used in parallel computing: the parallel cost C_p and the coefficient of variation of processor finishing times (COV). Their respective performance will be measured under two scientific applications as test beds: The many-body simulation [1] and the profiling of automatic quadrature routines [11].

2. DYNAMIC LOAD BALANCING

Dynamic scheduling for the purpose of load balancing parallel tasks have been developed by several researchers [25, 13, 22, 16]. The process of assigning tasks to processors by these scheduling algorithms will be discussed in this section, together with the profile of the tasks' execution times that could make the parallel system perform badly, if not its worst.

2.1 Fixed-Size Chunking

In FSC, a chunk of tasks is chosen at compile time. The chunk size c is usually chosen to be $c << n/p$ and, as much as possible, is divisible by p (i.e., p modulo c is 0). The first p chunks of tasks are initially distributed by the scheduler[1] to the processors[2]. When a processor happens to be assigned a chunk of tasks with short execution times, it will finish early. Since there are still chunks to be computed, it can request the scheduler for the next chunk, incurring a cost t_c brought about by communication latency. The communication latency is incurred by both the scheduler and the requesting processors, and happens when the processor requests for the next chunk until the scheduler answers with the next assignment. For scientific applications with independent loops, only the indexes of the iterates are communicated by the scheduler, incurring a smaller t_c. At the end of the computation, the processors' finishing times will vary at a maximum of c, improving the realized τ_p. FSC will perform badly if the profile of the tasks' execution time is such that the tasks T_i, $\forall i = 0, 1, \ldots, (n-c-1)$, have identical execution time of τ_α, while the remaining c tasks have an increasing $\tau_\beta > \tau_\alpha$. The execution time is

$$\tau_p = \left(\frac{n}{cp} \times \tau_\alpha\right) + \left(\sum_{\beta=n-c}^{n-1} \tau_\beta\right) + \left(\frac{n}{c} \times t_c\right). \tag{3}$$

2.2 Guided Self Scheduling

GSS is simply implemented as FSC with $c = 1$. As in FSC, the first p tasks are distributed among the p processors. When a processor finishes the tasks, it requests for the next task from the scheduler. The request incurs a communication cost of t_c. This scheduling algorithm will perform the

[1]In practice, the scheduler is usually P_0.

[2]If P_0 is the scheduler, then it also distributes to itself.

worst if the profile of the tasks execution times is such that that first $n-1$ tasks have identical execution time of τ_α, while the last task T_{n-1} has an execution time of $\tau_\beta >> \tau_\alpha$. The execution time of this example is given in Equation 4. If the profile of the tasks execution times, however, is such that any of the first p tasks $T_i, i = 0, 1, \ldots, p-1$, has an execution time of τ_β, while the other $n-1$ tasks have identical execution time of $\tau - \alpha$, then the parallel execution time is given in Equation 5. Notice here that depending on the profile of the tasks execution times, GSS will either perform worst or best.

$$\tau_p = \left(\frac{n}{p} \times \tau_\alpha\right) + (\tau_\beta) + \left(\frac{n}{p} \times t_c\right) \quad (4)$$

$$\tau_p = \max\left(\tau_\beta, \quad \frac{n}{p} \times \tau_\alpha\right) + \left(\frac{n}{p} \times t_c\right) \quad (5)$$

2.3 Factoring

FACT is a scheduling algorithm that implements variable-sized chunk c using some factoring rules. This algorithm has been proposed as a scheduling scheme to a number of scientific applications such as Monte-Carlo simulation, many-body simulations [1] and others [19]. The idea behind this scheme is to accommodate load imbalances caused by predictable phenomena, such as irregular data, as well as unpredictable phenomena, such as data-access latency due to I/O bursts and OS interference. One example of a factoring rule is to schedule chunks of tasks such that c is a fixed factor of those remaining. For example, setting the factor to be $0 < \gamma < 1$, then P_0 is assigned $\gamma \times n$ tasks and P_1 is assigned $\gamma^2 \times n$ tasks. In general, P_j is assigned $\gamma^{j+1} \times n$ tasks. The selection of c requires that the chunks have high probability of being completed by the processors before the optimal time. The chunk sizes are dynamically computed by the scheduler at runtime. When computing for the larger chunks, the processors incur a relatively low communication overhead. The unevenness of the respective finishing times of the larger chunks can be smoothed over by the smaller chunks made available towards the end of the computation.

One method that has been developed from FACT is *Fractiling*. During scheduling, fractiling takes into consideration the locality of data by exploiting the self-similarity properties of fractals. This method has been implemented successfully in the 2D version of the many-body simulations on both distributed shared address space and message-passing systems [5, 6].

2.4 Weighted Factoring

WFACT was proposed to take into consideration the effect of processor heterogeneity of the underlying runtime system. This method, derived from FACT, computes c by taking into consideration the relative computing speed of the processor as a weight for γ during the time of computation [18]. At the start of the computation, the relative processing speeds of the processors will be profiled resulting in an array $S = \{s_0, s_1, \ldots, s_{p-1}\}$ of relative processor speeds corresponding to processors $P_0, P_1, \ldots, P_{p-1}$. A vector of chunk sizes $C = \{c_0, c_i, \ldots, c_{p-1}\}$ will be computed from $c_i = s_i/s_{\max} \times \gamma$, where $s_{\max} = \max(s_0, s_1, \ldots, s_{p-1})$. These chunk sizes are statically assigned to processors and are considered to remain unchanged throughout the entire lifespan of the application. Experiments involving network of workstations, where relative processor speeds are extremely heterogeneous, have shown that WFACT significantly outperformed FACT.

2.5 Adaptive Weighted Factoring

In computational environments where processor workloads vary during the computation, chunk sizes must be assigned to processors dynamically. Many applications whose solutions require a number of iterations over the computation space are expected to benefit from a dynamic adjustment of weights after finishing each chunk of tasks. This aspect is addressed by the AWF [8], wherein the relative processor speeds are profiled after every computation of a chunk.

2.6 Adaptive Factoring

AFACT was developed as a general model for FACT, WFACT, and AWF schemes. AFACT's flexibility as a general model [7] is suited for highly irregular applications, where even within an iteration over the computation space, the load becomes unpredictably imbalanced. Because of its generality, AFACT reduces into either FACT, WFACT, or AWF under specific conditions of processor speeds and task workloads.

3. EXPERIMENT AND RESULTS

This section describes the experimental setup and its corresponding results within the two applications used as test beds for the comparison. To assess the performance of two applications without the dynamic load balancing algorithms, a static chunking (SC) algorithm was used as the basis for comparison. SC is similar to FSC with $c = n/p$ and reduces to the trivial offline scheduling decision discussed in Section 1. The dynamic load balancing algorithms evaluated in this study are FSC, GSS, FACT, WFACT, AWF and AFACT. The parallel cost $C_p = p \times \tau_p$, cost improvements over SC, and the COVs for both applications are presented and discussed.

3.1 Many-body Simulations

The many-body simulations is an application that has real-world importance in physical sciences such as molecular dynamics, astrophysics, plasma physics and many others. It simulates the dynamics of a system of n particles in 3D, given the initial positions, speeds and directions of the particles, and experiencing attraction with one another due to gravitational forces. The attractions are governed by the Inverse Square Law, while the net force acting on a particle determines the particle's next position, speed and direction. The computation per time step based on the naive algorithm has $O(n^2)$ time complexity. There exists a number of approximation algorithms that improve on time complexity of the naive algorithm [1, 9, 17]. Among these, the Fast Multipole Algorithm (FMA) [17] is $O(n)$ and is the most challenging for parallelization. Thus, FMA was used for this study. The parallel version of FMA uses an oct-tree data structure to represent a hierarchical decomposition of the 3D space being simulated. Studies that profile the parallel FMA show that the leaf level of the oct-tree is the most computationally intensive and imbalanced part of the computation. In this study, three initial distributions of the particles in 3D space were used: *uniform*, *Gaussian*

and *corner*. The corner distribution is a special case of the Gaussian distribution obtained by shifting the mean of the particle distribution on one of the octants of the 3D space. Experiments were conducted on three different sizes, n =(20K, 50K, 100K), under three different initial particle distributions (uniform, Gaussian and corner) using five different number of processors (2, 4, 8, 16 and 32) in five trials.

Figures 2, 3 and 4 show the cost $C_p = p \times \tau_p$ for, respectively, solving the uniformly distributed, the Gaussian distributed and the *corner* distributed 100K-body problem using different dynamic load balancing algorithms. Table 1 shows the percent cost improvement of dynamic load balancing algorithms over the static algorithm.

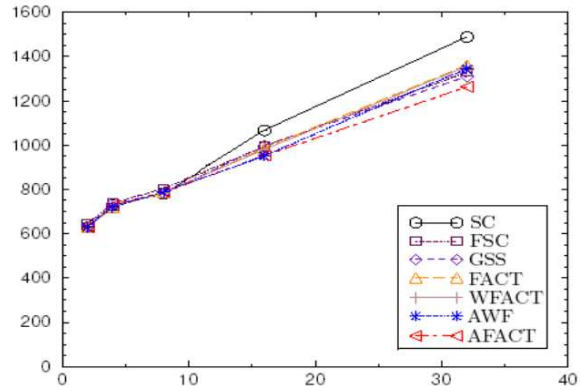

Figure 2: $p \times C_p$ plot for solving the 100K-body problem, with uniform distribution using different dynamic load balancing algorithms.

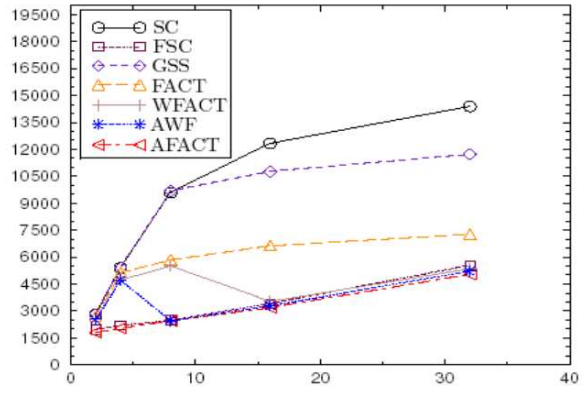

Figure 3: $p \times C_p$ plot for solving the 100K-body problem, with Gaussian distribution using different dynamic load balancing algorithms.

In the case of uniformly distributed particles, the application suffers from only little load imbalance. Figure 2 shows that there is no substantial performance difference between the scheduling algorithms. However, all dynamic scheduling algorithms perform better than SC. AFACT provided the maximum cost improvement over SC. This highlights that the dynamic load balancing algorithms is effective, even when the application has small variability of task execution times, owing to the uniformly distributed manner of the data.

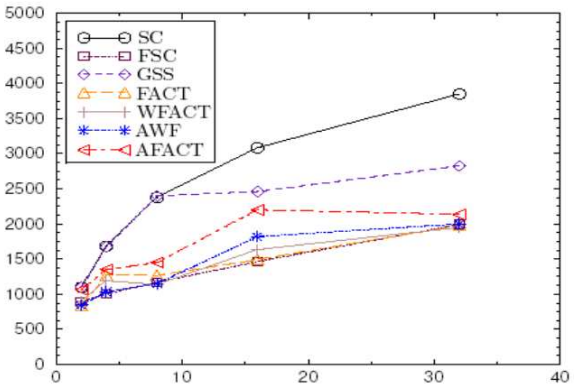

Figure 4: $p \times C_p$ plot for solving the 100K-body problem, with *corner* distribution using different dynamic load balancing algorithms.

Table 1: Maximum percent cost improvement of dynamic load balancing algorithms over SC for many-body simulations with different initial 100K particle distributions. Values in parenthesis are the number of processors where the improvement in cost is the highest.

Load Balancing Method	Distribution		
	Uniform	Gaussian	Corner
FSC	11 (32)	74 (8)	53 (16)
GSS	12 (32)	18 (32)	27 (32)
FACT	9 (32)	49 (32)	52 (16)
WFACT	9 (32)	71 (16)	52 (8)
AWF	11 (16)	74 (8)	52 (8)
AFACT	15 (32)	74 (8)	45 (32)

When the initial particle distribution is nonuniform, as in the case of Gaussian and *corner* distributions, all the scheduling algorithms consistently outperform SC (Figures 3 and 4). Algorithms like GSS and FACT tend to perform poorly in this case because too much work is allocated by these algorithms during the earlier part of the run, leaving insufficient work for the later stages of the run to smooth over the load imbalance cased by the earlier chunks. FSC provides the best percent cost improvement over SC when the initial particle distribution is nonuniform (up to 74% and 53% for Gaussian and *corner*, respectively). The reason is that the optimal c has already been empirically determined. Among the factoring variants, WFACT (up to 52%) and AWF (up to 52%) perform as good as FACT (up to 52%) when the initial particle distribution is Gaussian. AFACT (up to 45%) performed poorly when the initial particle distribution is *corner* but performed as well as FSC (both at up to 74%) when the distribution is Gaussian.

When the initial particle distribution os uniform (Figure 2), AFACT shows the best increasing percent cost improvement over SC as p is increased. These trends suggest that even though the application may not seem to benefit from a load balancing algorithms because of the inherent initial distribution of the data, a relative cost improvement can

still be achieved. This also indicates the low scheduling overhead for AFACT. When the initial particle distribution is *corner*, all the FACT-based algorithms, FSC and GSS, exhibit increasing percent cost improvements over SC as p is increased. When the initial particle distribution is Gaussian, all the FACT-based algorithms, and FSC, exhibit increasing percent cost improvement over SC as p increases. GSS also shows an increasing percent cost improvement over SC as p is increase, but the percent increases in cost are not as high as those of other algorithms. These trends suggest that when the initial particle distribution is non-uniform, and p is increased, all the FACT-based algorithms and the FSC costs are lower than those of SC.

All runs exhibit low COV when the particles are initially uniformly distributed, with GSS, SC, and FSC exhibiting processor finishing times variations as low as the FACT-based algorithms. When the particles are non-uniformly distributed, the FACT-based algorithms, as well as the FSC, exhibit lower variability in processor finsihing times than that of GSS and SC. In general, this suggests that the dynamic load balancing algorithms have a higher impact on lowering the variability of processor finishing times than that of the SC.

3.2 Automatic Quadrature Routine

Scienfic computations such as multivariate statistics, finite element methods, particle physics, and others, use Automatic Quadrature Routines (AQR) to approximate an integral

$$I = \int_D f(x)dx,$$

where D is the domain of integration, and $f(x)$ is the integrand. Here, x could represent a single variable or a vector. The inputs to AQR are typically:

1. a description of D;
2. the code for the $f(x)$;
3. an index to a quadrature rule;
4. absolute and relative error tolerances; and
5. parameters for other stopping criteria in case the error tolerances are not achievable.

The profiling of an AQR is an empirical study that attempts to highlight the accuracies achievable by the routine and the costs involved, using various families of integrands. The integrands are chosen such that the answers are known analytically to facilitate the computation of the true error in the computed result as well as the accuracy of the error estimate. An MPI implementation of the AQR [11] is used for the performance experiments in this study.

Profiling an AQR is a very simple yet time-consuming three-phase process. During the first phase, a large set of parameters is generated, wherein each set element defines a sample integral with specific properties and error requirements. The second phase is composed of a loop with embarrassingly parallel iterates. Each iterate invokes the AQR on a sample integral and determines the quality of the approximation. The third phase simply the generation of summary statistics. The bulk of the execution time is spent on the loop in the second phase. The iterate execution times are also highly variable because of the differences in integrand types, accuracy requirements,a dn quadrature rule settings. Depending on the order in which the integrand families are profiled, a loop with a multitude of load distribution characteristics is generated. The ones used in this study are *front-heavy*, *center-heavy*, and *scatter-heavy* load distributions. In the front-heavy load distribution, the most time-consuming integrands are evaluated first followed by the evaluation of the easy ones. The order is reversed for back-heavy load distribution. The most time-consuming integrands appear in the middle of the loop for center-heavy load distribution. Experiments were conducted on three distributions, three problem sizes (37,800, 75,600 and 151,200 integrands), and four different numbers of processors (8, 16, 24 and 32). The results of 151,200-integrand experiments are representative, hence are shown here.

Figure 5 shows the cost of executing the 151,200-integrand scatter-heavy AQR profiling problem using different dynamic load balancing algorithms. AFACT and FSC perform best in terms of cost followed by both AWF and AFACT. The FACT cost is better than that of the GSS while the SC cost is the highest. The best percent in cost improvement over SC is obtained when using the FSC (up to 73%). The reason for this is that the optimum c for FSC has been determined earlier. However, since AFACT does not use *a priori* knowledge about the task profile, AFACT obtains the next best percent cost improvement (up to 71%) over SC. The other FACT-based algorithms (AWF, WFACT, and FACT) post similar percent cost improvements (up to 68%, 65%, and 52%, respectively) over SC. GSS achieves up to 23% cost improvement over SC. These observations suggest that the FACT-based algorithms can achieve percent cost improvement comparable to an algorithm that uses prior knowledge about the task profile. Thus, the FACT-based algorithms offer more flexibility for running applications where the user doe not have profiling information about the parameters of the scheduling algorithm.

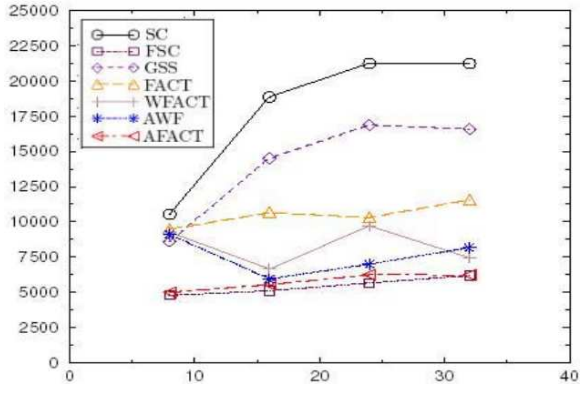

Figure 5: $p \times C_p$ plot for solving the 151,200-integrand scatter-heavy profiling problem using different dynamic load balancing algorithms.

The best COV is achieved by FSC (as lows as 0.07) while the FACT-based methods AFACT, WFACT, and AWF have the next best COVs (as low as 0.12, 0.16, and 0.16, respectively). FACT and GSS have COVs as low as 0.50 and 0.66, respectively, while SC posts finishing times with the highest variability (COV of 0.85).

4. SUMMARY AND CONCLUSION

This paper presents the comparison of six dynamic load balancing algorithms namely, FSC, GSS, FACT, WFACT, AWF, and AFACT. These algorithms were developed for load balancing scientific applications on distributed memory architectures. The use of the algorithms emphasize the performance enhancement obtained by applications since even in applications that exhibit low amount of load imbalance, such as that of the uniform distribution, the algorithms were able to make corrections that resulted in cost improvements.

The experimental results highlights the benefits of using the algorithms. The many-body simulations and the AQR profiling are described and used as test beds for the comparison between the algorithms. Performance metrics and quantitative analyses of dynamic task scheduling algorithms for solving the many-body simulations and the AQR profiling applications are presented. In many-body simulations, cost improvements of up to 74% are obtained while in AQR, cost improvements of up to 73% are achieved. The algorithms provide flexibility to the users because it offers load balancing procedures that do not require *a priori* knowledge about its parameters.

5. ACKNOWLEDGMENTS

The author thanks the Institute of Computer Science and the College of Arts and Sciences, University of the Philippines Los Baños for its financial support of this work through CAS-TF #8217300 and ICS-GF #2326103, respectively. A big portion of this work was performed in the Engineering Research Center, Mississippi State University (MSU) and was supported by the National Science Foundation through several grants: CAREER #9984465, ITR/ACS #0085969, ITR/AC #0081303, #0132618, and #0082979. The author thanks his collaborators:

- Dr. Ioana Banicescu, Professor of Computer Science, Department of Computer Science and Engineering, Worth Bagley College of Engineering, MSU
- Dr. Ricolindo L. Cariño, Research Professor, Center for Advanced Vehicular Systems, Engineering Research Center Collaboratory, MSU.

6. ABOUT THE AUTHOR

J.P. Pabico is Associate Professor 2 of the Institute of Computer Science, University of the Philippines Los Baños, and Affiliate Associate Professor of the Faculty of Information and Communication Studies, University of the Philippines Open University. He is currently working on three research projects:

- Static, Dynamic, and Adaptive/Reinforcement Learning-Based Scheduling and Load Balancing of Parallel Tasks on Distributed Desktop Grids;
- Application of Computational Intelligence Techniques in Optimizing Agricultural, Biological, Environmental, Natural and Artificial Engineering Systems; and
- Structural Characterization and Temporal Dynamics of Various Natural, Social and Artificial Networks in the Philippines.

He is one of the **2008 Outstanding Young Scientists of the Philippines**, a **Metro Manila Commission Professorial Chair** awardee, and recently **Finalist** in the DOST-PCASTRD *2008 Search for Outstanding Research and Development in Advanced Science and Technology.*

7. REFERENCES

[1] A.W. Appel. An efficient program for many-body simulations. *SIAM Journal of Scientific Statistical Computing*, 6(1):85–93, 1985.

[2] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J.P. Pabico, and R.L. Cariño. A novel dynamic load balancing library for cluster computing. In *Proceedings of the Joint 3rd International Symposium on Parallel and Distributed Computing and 3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 346–353, 2004.

[3] M. Balasubramaniam, R.L. Cariño, I. Banicescu, and J.P. Pabico. Performance evaluation of a dynamic load balancing library for cluster computing. *International Journal of Computational Science and Engineering*, 1(2/3/4):118–133, 2005. (Special Issue on Adaptivity in Parallel Scientific Computing).

[4] I. Banicescu, R.L. Cariño, J.P. Pabico, and M. Balasubramaniam. Design and implementation of a dynamic load balancing library. *Parallel Computing*, 31(7):736–756, 2005. (special issue on Heterogeneous Computing).

[5] I. Banicescu and S.F. Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *Proceedings (CDROM) of the 1995 ACM/IEEE Conference on Supercomputing*, 1995.

[6] I. Banicescu and R. Lu. Experiences with Fractiling in N-body simulations. In *Proceedings of the 1998 High Performance Computing Symposium*, pages 121–126, 1998.

[7] I. Banicescu and V. Velusamy. Load balancing highly irregular computations with the Adaptive Factoring. In *Proceedings of the Joint IEEEE International Parallel and Distributed Processing Symposium and Heterogeneous Computing Workshop*, pages 87–98, 2002.

[8] I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with Adaptive Weighted Factoring. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 6(3):215–226, 2003.

[9] J.E. Barnes and P. Hut. A hierarchical $O(N \log(N))$ force calculation algorithm. *Nature*, 324(6096):446–449, 1986.

[10] R. Biswas, S.K. Das, D. Harvey, and L. Oliker. Portable parallel programming for dynamic load balancing of unstructured grid applications. In *Proceedings of the Joint 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 338–342, 1999.

[11] R.L. Cariño. *Numerical Integration Over Finite Regions Using Extrapolations by Nonlinear Sequence Transformations*. PhD thesis, La Trobe University, Australia, 1992.

[12] R.L. Cariño, M. Balasubramaniam, I. Banicescu, and J.P. Pabico. Overhead analysis of a dynamic load balancing library for cluster computing. In *Proceedings (CDROM) of the Joint 19th International Parallel and Distributed Processing Symposium and 14th International Heterogeneous Computing Workshop*, 2005.

[13] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[14] J. Chen and V.E. Taylor. Mesh partitioning for distributed systems: Exploring optimal number of partitions with local and remote communication. In *Proceedings (CDROM) of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1998.

[15] S. Dhandayuthapani, I. Banicescu, R.L. Cariño, E. Hansen, J.P. Pabico, and M. Rashid. Automatic selection of loop scheduling algorithms using reinforcement learning. In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments*, pages 87–94, 2005.

[16] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1996.

[17] L. Greengard and V.A. Rokhlin. A fast algorithm for particle simulation. *Journal of Computational Physics*, 72(2):325–348, 1987.

[18] S.F. Hummel, J. Schmidt, R.N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of hte 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[19] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[20] B. Maerten, D. Roose, A. Basermann, J. Fingberg, and G. Lonsdale. DRAMA: A library for parallel dynamic load balancing of finite element applications. In *Proceedings (CDROM) of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1998.

[21] K. Schloegel, G. Karypis, and V. Kumar. Dynamic repartitioning of adaptively refined meshes. In *Proceedings (CDROM) of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1998.

[22] N.G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

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

[24] A. Sohn and H. Simon. S-HARP: A scalable parallel dynamic partitioner for adaptive mesh-based computations. In *Proceedings of the IEEE/SIAM Supercomputing Conference*, pages 7–13, 1998.

[25] Y.T. Wang and R.J.T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, 1985.

[26] M.S. Warren and J.K. Salmon. A parallel hashed oct tree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 12–21, 1993.