J Supercomput (2008) 44: 41–63
DOI 10.1007/s11227-007-0148-y

Dynamic load balancing with adaptive factoring methods in scientific applications

Ricolindo L. Cariño · Ioana Banicescu

Published online: 5 October 2007

Abstract To improve the performance of scientific applications with parallel loops, dynamic loop scheduling methods have been proposed. Such methods address performance degradations due to load imbalance caused by predictable phenomena like nonuniform data distribution or algorithmic variance, and unpredictable phenomena such as data access latency or operating system interference. In particular, methods such as factoring, weighted factoring, adaptive weighted factoring, and adaptive factoring have been developed based on a probabilistic analysis of parallel loop iterates with variable running times. These methods have been successfully implemented in a number of applications such as: N-Body and Monte Carlo simulations, computational fluid dynamics, and radar signal processing.

The focus of this paper is on adaptive weighted factoring (AWF), a method that was designed for scheduling parallel loops in time-stepping scientific applications. The main contribution of the paper is to relax the time-stepping requirement, a modification that allows the AWF to be used in any application with a parallel loop. The modification further allows the AWF to adapt to load imbalance that may occur during loop execution. Results of experiments to compare the performance of the modified AWF with the performance of the other loop scheduling methods in the context of three nontrivial applications reveal that the performance of the modified method is

This work was partially supported by the National Science Foundation Grants: #9984465, #0081303, #0313274, #0085969 and #0132618.

R.L. Cariño (✉)
Center for Advanced Vehicular Systems—HPCC, Mississippi State University, Mississippi State, MS 39762, USA
e-mail: rlc@cavs.msstate.edu

I. Banicescu
Department of Computer Science and Engineering, and Center for Computational Sciences—HPCC, Mississippi State University, Mississippi State, MS 39762, USA
e-mail: ioana@cse.msstate.edu

comparable to, and in some cases, superior to the performance of the most recently introduced adaptive factoring method.

Keywords Dynamic load balancing · Adaptive weighted factoring

1 Introduction

A major source of parallelism in scientific computing is the *parallel loop,* a loop without dependencies among its iterates. The straightforward parallelization strategy of statically dividing the iterates equally among available processors results in load imbalance if the iterate execution times vary and/or the processors have different speeds. Strategies to address load imbalance have been developed in cases where the variations are known or are predictable before loop runtime. However, load imbalance might be induced during execution time by system effects such as data access latency and operating system interference; thus, fully *dynamic loop scheduling* methods are needed, in which runtime information is taken into account when distributing loop iterates to processors.

Dynamic loop scheduling methods such as factoring [24] (FAC), fractiling [6] (FRAC), weighted factoring [23] (WF), adaptive factoring [7, 9] (AF) and adaptive weighted factoring [8, 10] (AWF), have been proposed. Based on a probabilistic analysis, these methods schedule the execution of loop iterates in chunks with variable sizes. The chunk sizes are determined during loop runtime such that chunks have a high probability of being completed before the optimal time. Some of the parallel applications in which the methods have been successfully incorporated include: FAC in Monte-Carlo simulations, WF in radar signal processing, FRAC in N-body simulations, and AWF in computational fluid dynamics.

In this paper, we propose a modification of the AWF. Originally, the AWF was designed for scientific applications like the N-body and CFD simulations which involve time-stepping. The method assigns weights to processors, with the weights initially set to unity. During a time step, the time per iterate ratio in each processor is measured for the parallel loop under consideration. At the end of the time step, a weighted average of the ratios in each processor is computed with the time step as the weighing factor. The average ratios are then normalized to become the processor weights for the next time step. Thus, the original AWF method does not adapt to load imbalance factors that arise while the parallel loop is being executed. Our proposed modification to the AWF makes use of the ratios based on timings from *earlier chunks* to compute the processor weights for the succeeding chunks, instead of using ratios collected from *previous time steps.* This modification allows the AWF to be used in a wider range of scientific applications such as those that do not involve time steps. The modified AWF adjusts to irregularities that may occur during loop runtime since the processor weights are adapted while the loop is being executed, unlike the original AWF where the processor weights are fixed during loop execution.

The rest of this paper is outlined as follows: Sect. 2 traces the development of loop scheduling methods, with emphasis on the factoring methods. This section also describes a strategy for dynamic loop scheduling on message-passing environments. Section 3 details the proposed modification to the adaptive weighted factoring

method; specific implementation variants are also described. Section 4 summarizes the results of experiments to compare the performance achieved by the modified AWF and other loop scheduling methods for three nontrivial applications: profiling of a quadrature routine, simulation of wave packet dynamics using the quantum trajectory method, and statistical analysis of multiple datasets. The results indicate that the performance achieved by the modified AWF is comparable to, and in some cases better than the performance achieved by adaptive factoring, the most recently introduced loop scheduling method. Section 5 concludes the paper.

2 Loop scheduling methods

Assume that in a parallel application, a loop with N independent iterates is to be executed by P processors. The iterates are stored in a central ready work queue from which idle processors obtain chunks. The sizes of the chunks are determined according to a scheduling method which attempts to minimize the overall loop execution time. The method is classified as *nonadaptive*, when the chunk sizes are predictable from information that is available or assumed before loop runtime, or *adaptive*, when the chunk sizes depend on information available only during loop execution.

2.1 Nonadaptive methods

Nonadaptive loop scheduling methods generate equal size chunks or predictable decreasing size chunks. Equal size chunks are generated by *static scheduling* (STATIC), where all the chunks are of size N/P, *self-scheduling* (SS), where all the chunks are unit size, and *fixed size chunking* [28] (FSC). STATIC has very low overhead, but it provides good load balancing only if the iterate times are constant and the processors are homogeneous and equally loaded. SS is suitable only when communication overhead is negligible as it requires sending N control messages. In FSC, the chunk size is computed as

$$\text{FSC chunk size} = \left((\sqrt{2}Nh)/(\sigma P\sqrt{\log P})\right)^{2/3}$$

which will minimize the parallel execution time on homogeneous and equally-loaded processors if the following parameters are known: h—a constant overhead time, μ—the mean iterate execution time, and σ—the standard deviation of the iterate execution times.

For methods that generate predictable decreasing size chunks, the underlying idea is to initially allocate large chunks and later use the smaller chunks to smoothen the unevenness of the execution times of the initial larger chunks. *Guided self-scheduling* [32] (GSS) computes the chunk size as

$$\text{GSS chunk size} = \lceil \textit{remaining}/P \rceil,$$

where *remaining* is the number of unscheduled iterates. GSS assumes that the loop iterates have uniform execution times. In *trapezoid self-scheduling* [34] (TSS), the chunk sizes decrease linearly, in contrast to the geometric decrease of the chunk sizes

in GSS. *Factoring* [24] (FAC) on the other hand, schedules iterates in batches, where the size of a batch is a fixed ratio of the unscheduled iterates, and the batch is divided into P chunks. The ratio is determined such that the resulting chunks have a high probability of finishing before the optimal time. In general, the ratio depends on the mean and standard deviation of the iterate execution times; when these statistics are not known, the ratio 0.5 has been experimentally proven to be practical. In this case,

$$\begin{aligned} \text{FAC batch size} &= 0.5 * \textit{remaining}, \\ \text{FAC chunk size} &= \lceil \text{FAC batch size}/P \rceil. \end{aligned} \tag{1}$$

The next batch size is calculated after all the chunks in the current batch are scheduled. In N-body simulations, the combination of factoring and *tiling*, a technique for organizing data to maintain locality, is known as *fractiling* (FRAC) [6]. *Weighted factoring* [23] (WF) incorporates information on processor speeds in computing chunk sizes. In particular, the factoring chunk sizes are multiplied by relative processor speeds:

$$\text{WF chunk size} = w_i * \text{FAC chunk size}, \tag{2}$$

where w_i is the relative speed of processor i. Thus, from a batch, the faster processors get bigger chunks than slower processors. The relative processor speeds are assumed to be fixed throughout the execution of the loop, so the size of chunks executed by a given processor monotonically decrease in size.

A number of methods that generate adaptive size chunks have evolved from factoring and weighted factoring. These methods are summarized next.

2.2 Adaptive methods

Adaptive weighted factoring [8, 10] (AWF) was developed specifically for time-stepping applications containing parallel loops. In such an application where a parallel loop is executed in every step, the amount computations in a loop iterate may evolve as the application progresses. Furthermore, the loads of the processors running the application may also be changed by the operating system. Under these evolving conditions, assigning chunk sizes to processors based on the above nonadaptive loop scheduling techniques may not yield the best possible performance. AWF attempts to incorporate both loop characteristics and processor capacities in determining the chunk sizes. Within a time step, the iterates of a parallel loop are assigned to processors as in WF; however, the processor weights w_i are adjusted at the end of a step using information collected during the current and previous steps. For the first time step, $w_i = 1.0$. In every step, the execution times of chunks in each processor are recorded. These timing data are used to update the w_i in preparation for the next time step. More details about the AWF are presented in Sect. 3, where a modification is proposed to make the method suitable for applications that do not involve time-stepping.

Adaptive factoring [7, 9] (AF) relaxes the assumptions in the original factoring (FAC) technique that the mean and standard deviation of the iterate execution times are known *a priori*, and that these are the same on all processors. Adaptive factoring dynamically estimates these statistics during runtime. First estimates are obtained

from the execution times of chunks from an arbitrary sized initial batch. The sizes of succeeding chunks are then computed using these estimates; these estimates are refined by using more information from recently executed chunks. In AF, the chunk sizes are computed as follows: let μ_i, σ_i denote the mean execution time and standard deviation of the execution times, respectively, of the most recent iterates done by processor i. Let

$$R = \textit{remaining}, \qquad D = \sum_{i=1}^{P} \sigma_i^2/\mu_i, \qquad T = \left(\sum_{i=1}^{P} 1/\mu_i\right)^{-1}.$$

Then, the next chunk size for processor i is

$$\text{AF chunk size} = \left(D + 2TR - \sqrt{D^2 + 4DTR}\right)/(2\mu_i).$$

2.3 Loop scheduling overhead and performance

Assuming that the scheduling operation has the same cost regardless of the size of the chunk, analytical expressions for the scheduling overhead can be derived for most of the techniques. The overhead depends on the number of chunks generated by the technique. Each chunk triggers some arithmetic operations to compute the chunk size, and bookkeeping operations for information about the chunk. In a message-passing environment, the chunk size may have to be requested and communicated. The fixed sized techniques (STAT, SS, FSC) generate N/k chunks, where k is the fixed size, while GSS, FAC and AWF generate $O(P \log(N/P))$ chunks. For the adaptive factoring (AF) and the proposed AWF variants, the number chunks is unknown since this depends on measurements taken during runtime; however, experience indicates that it is no more than twice that of FAC. Since the number of chunks does not asymptotically exceed the problem size N, loop scheduling is theoretically considered scalable.

When an application invokes loop scheduling to execute a parallel loop on a set of processors, the loop completion time in each processor is easy to measure. Denote this time by t_r for processor r, and let W be the amount of work done by the loop. Then, the following performance metrics can be computed:

parallel time $T_P = \max\{t_r\}$;
cost $= P * T_P$, the aggregate time used by the parallel system to execute the loop;
performance $= W/T_P$, the ratio of total work to parallel time; and
effectiveness $\Gamma = (W/T_P)/(P * T_P) = W/(P * T_P^2)$, the ratio of performance to cost [30].

When using these metrics to compare different methods to schedule a loop on the same computational platform, the "better" methods are those with lesser T_P and *cost*, or those with greater *performance* and *effectiveness*.

If the processors are homogeneous, let u_r denote the time spent by processor r executing loop iterates only (i.e., doing useful work). Then, the total useful work time for the loop is $T_1 = \sum_{r=0}^{P-1} u_r$, and the following additional metrics can be computed:

coefficient of variation $(c.o.v.) = \sigma/\mu$ *of work times*, where $\mu = T_1/P$, $\sigma = (\sum_{r=0}^{P-1}(\mu - u_r)^2)/(P-1)$;
speedup $= T_1/T_P$; and
efficiency $= (T_1/T_P)/P$.

2.4 Message-passing implementation

The loop scheduling methods described above were developed assuming a central ready work queue of iterates where idle processors obtain chunks of iterates to execute. This assumption is ideally suited for a shared memory architecture: the processors have to synchronize on a very small set of variables in order to determine the next chunk to be executed, and the data for the iterates can be accessed concurrently. On a message-passing environment where there is no centralized memory, the initial implementations of FAC, WF, AWF and AF utilize a work queue of iterates that is *replicated* in all processors. One of the processors, the *scheduler*, polls for requests for chunk sizes. The scheduler maintains a `chunk table` to keep track of which chunks are executed by each worker. A dedicated processor for the scheduler may not be necessary since chunk size computations involve simple arithmetic; hence, the scheduler can also participate in executing iterates. The replicated work queue setup has the advantage of using only very short control messages to and from the scheduler; data movement during loop computation is not necessary since each processor has a local copy of the work queue. However, it may be necessary to consolidate all the results of the computations into one of the processors, logically the scheduler. The disadvantages include the possibility of the scheduler becoming a bottleneck when the number of processors becomes large, and more significantly, the problem size must be limited such that the work queue will fit into the memory available on any of the processors. In contrast, a *partitioned* work queue is utilized in the implementations of FRAC for the N-body application where only a subset of the application data is stored in each processor. This partitioned setup accommodates larger problem sizes; however, during loop execution, the scheduler may direct the redistribution of data for load balancing, which will contribute substantially to the communication overhead.

Figure 1 illustrates the loop scheduling strategy with a replicated work queue on a message-passing environment. A worker sends a `Request` message to the scheduler to request a chunk size. When received, this message triggers a scheduling event, to interrupt the scheduler from executing iterates if the scheduler doubles as a worker. The scheduler determines a chunk size according to the loop scheduling technique and responds with a `Size` message containing this size to the requesting worker. The `Request` message contains the performance data of the worker based on timings of the previously executed chunk. These timings provide the scheduler with a continuously updated global view of worker performance during loop execution. These timings are used by the scheduler to determine subsequent chunk sizes according to the loop scheduling technique. The execution of iterates in the workers may be interleaved with the `Request` message to reduce waiting time of workers for chunk sizes. With interleaving, a worker sends the request before executing the last few iterates of the current chunk; hopefully, the next chunk size would have arrived from the

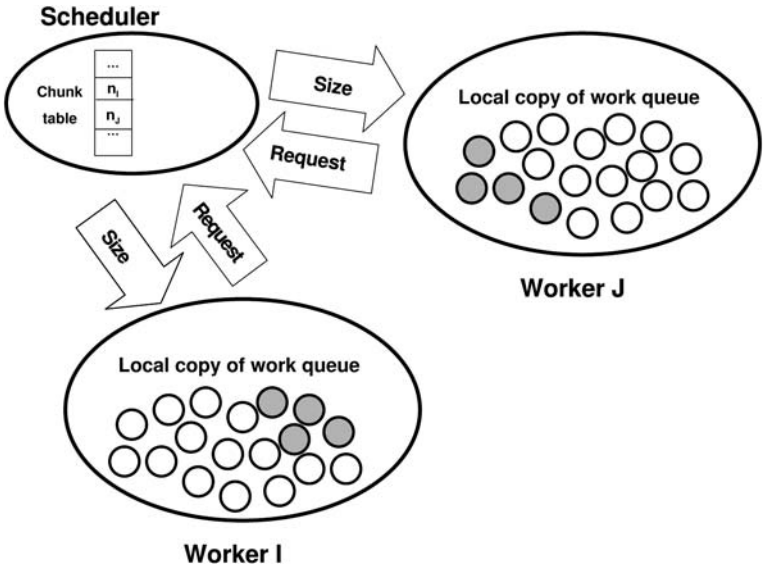


Fig. 1 Dynamic loop scheduling with a replicated work queue on a message-passing environment

scheduler before the computation for the last iterate of the current chunk is finished. To establish the number of iterates left in the current chunk before sending the request for a new chunk size, precise measurements of send/receive message latencies would have to be conducted.

Dynamic loop scheduling in a message-passing environment with a partitioned work queue occurs in two phases. Before loop execution, the loop data is already divided among the participating processors. The first phase proceeds in a manner similar to the strategy with a replicated work queue, until a worker finishes ahead with its portion of the work queue. Figure 2 illustrates the second phase which commences when a worker, say Worker I, that has finished its share of the queue, sends a `Request` message. The scheduler recognizes the situation from the chunk table; it determines the next chunk size and the slowest worker, say Worker J, and sends these information to Worker I through a `Get` message. Upon receipt of this message, Worker I prepares a receive buffer, sends a `Give` message containing the size to Worker J and posts a receive. Worker J then sends a chunk of iterates to Worker I. In computing the size, the scheduler may consider many factors such as the availability of space for the data of new iterates in Worker I, estimates of the cost of moving data from Worker J to Worker I, and the penalty of load imbalance if the iterates were to be done by Worker J. If one sided communications are supported by the message-passing environment, an alternative data transfer strategy is for Worker I to perform a one sided get operation for a chunk of iterates from Worker J.

A general purpose loop scheduling routine based on a partitioned work queue is proposed in [16]. The routine is utilized by the applications listed in Sect. 4, as well as in other work presented elsewhere [1, 17, 19, 20]. The routine is designed to be invoked by MPI applications that contain parallel loops like `y(i) = f(x(i)), i=1,2,...,N`, where `x()` and `y()` are arrays of possibly complex types, and are

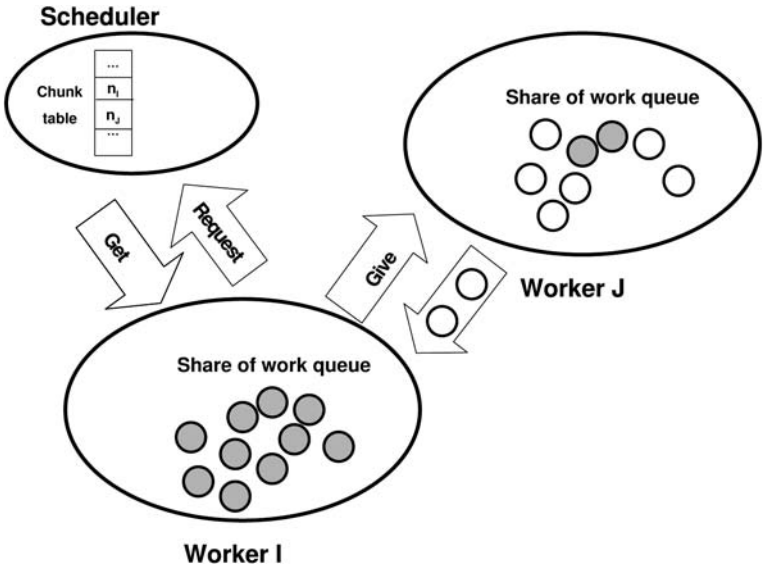


Fig. 2 Dynamic loop scheduling with a partitioned work queue on a message-passing environment

partitioned among the processors. The loop scheduling routine takes as arguments the routine that encapsulates `f()` for a chunk of iterations, a pair of complimentary routines for sending and receiving chunks of `x()`, a similar pair of routines for chunks of `y()`, and the partitioning of the `N` data items among the processors. The routine performs load balancing only: a chunk of `x()` is sent, for example, from worker J to worker I, and worker I returns a chunk of `y()`, to maintain the partitioning specified by the application. The routine has subsequently been modified to support a replicated work queue, in which case, the routine for sending a chunk of `x()` is not used.

3 Adaptive weighted factoring variants

Adaptive weighted factoring [8, 10] (AWF) was originally designed for executing a parallel loop in a scientific application which involves time-stepping. The amount computations associated with a loop iterate may evolve as the application progresses, or the loads of the processors running the application may be changed by the operating system. Adaptivity in the chunk sizes is, therefore, necessary to obtain the best possible application performance.

Based on weighted factoring (WF), AWF incorporates adaptivity by updating the relative processor weights w_i in Eq. 2 after every time step. Initially, $w_i = 1$. Suppose during step s, processor i allocated t_{is} time units to execute n_{is} iterates. Then, the time per iterate *ratio* in processor i during this step is t_{is}/n_{is}. Considering all the

steps until s, the *weighted average ratio* is

$$\pi_i = \left(\sum_{j=1}^{s} j \times t_{ij}\right) \Big/ \left(\sum_{j=1}^{s} j \times n_{ij}\right). \tag{3}$$

π_i is biased towards the latest timings, since measurements from recent steps are given higher weights. The collection of weighted average ratios from each processor serves as the basis for the processor weights required by the WF technique. The conversion from the weighted average ratio to processor weight proceeds as follows: let

$$\bar{\pi} = \left(\sum_{j=1}^{P} \pi_j\right) \Big/ P, \qquad \rho_i = \bar{\pi}/\pi_i, \qquad \hat{\rho} = \sum_{j=1}^{P} \rho_j. \tag{4}$$

The quantities may be interpreted as: $\bar{\pi}$ is the *average* weighted average ratio; ρ_i is the *raw weight* of processor i; and $\hat{\rho}$ is the *sum* of the raw weights used to normalize the raw weights. Then, the *normalized weight* of processor i for computing its WF chunk sizes for the next time step is

$$\text{AWF weight} \quad w_i = (\rho_i \times P)/\hat{\rho}. \tag{5}$$

From Eq. 3, an increase in π_i because processor i is taking longer to execute its chunks, results in a smaller ρ_i and w_i when these are computed at the end of the time step. Consequently, processor i will be assigned smaller chunks in the next time step.

However, adaptation in AWF is based on information collected from *previous* time steps. Changes in effective processor speeds or changes in the computational requirements of iterates during the current time step will be reflected only in the weights for the succeeding time step. Thus, the method does not adapt the weights during a time step for any load imbalance that occurs during the step.

Our proposed modification to the AWF is to utilize the timings of the execution of earlier chunks to adapt the processor weights for the succeeding chunks. This modification relaxes the requirement of the AWF that the parallel loop must be inside a time-stepping loop: timing data from previous execution of the parallel is no longer necessary. This modification further allows more frequent adaptation since the processor weights can be adjusted while the parallel loop is being executed.

The modified AWF can be implemented with a few variations, some of which are described below. Initially, the processor weights are set to unity like in AWF. An arbitrary-size batch of $\beta_0 * N$ iterates, $0 < \beta_0 < 1$, is selected like in AF to determine an initial chunk size. From this batch, processor i will be assigned the first chunk of size $n_{i1} = \beta_0 * N/P$. The succeeding chunks may be determined by one of the following strategies.

AWF-B (*batched* AWF). The remaining iterates are scheduled by batches like in FAC. The chunk sizes are computed as in AWF using Eqs. 4–5, but with

$$\pi_i = \left(\sum_{j=1}^{s_i} j \times t_{ij}\right) \Big/ \left(\sum_{j=1}^{s_i} j \times n_{ij}\right), \tag{6}$$

where s_i is the number of chunks executed by processor i. Timings from previous chunks are used in Eq. 6, unlike in Eq. 3 which uses timings from previous time steps. Further, Eq. 6 allows updates to the weights while the loop is being executed.
AWF-C (*chunked* AWF). The remaining iterates are scheduled by chunks like in AF, instead of by batches. The scheduling strategy formulated in FAC, WF, AWF and AWF-B stipulates that the chunk sizes are computed as a fraction of the *current* FAC batch size (see Eq. 2). With this formulation, a processor that has completed its portion of the batch will be assigned remaining iterates from the current batch. This may result in faster processors being assigned a chunk of less-than-optimal size, leading to higher overhead due to more frequent communications. As a remedy, AWF-C recomputes a new batch size each time a processor requests for work, before applying Eqs. 6, 4, and 5. With the AWF-C strategy, faster processors are assigned larger chunks from all the remaining iterates, not just from the remainder of the current batch.
AWF-D. This strategy is similar to AWF-B, but with t_{ij} redefined as the *total chunk time*. In the original AWF, t_{ij} is the execution time (only) of n_{ij} iterates. In AWF-D, t_{ij} includes the time spent by the processor doing other tasks associated with the execution of a chunk of iterates, like bookkeeping and checking for messages. The total chunk time provides the loop scheduler a more accurate measure of the load associated with a chunk.
AWF-E. This strategy is similar to AWF-C, but using total chunk time as in AWF-D.

In comparison with the AF, the modified AWF can also be used in any application with a parallel loop, since information from a previous run of the loop is no longer necessary. Both techniques adapt to load imbalance that may occur during loop runtime since the processor weights are updated while the loop is being executed. In contrast, the modified AWF measures the execution times of chunks instead of individual iterates as required in AF. Thus, the modified AWF incurs less overhead than AF for timing measurements since the clock is checked much less frequently.

4 Applications

Three nontrivial applications were employed to compare the performance obtained by the modified AWF and other loop scheduling methods. These applications are: profiling a quadrature routine, simulation of wave packet dynamics, and statistical analysis of multiple datasets. In these applications, the bulk of the execution times are spent on computationally intensive parallel loops. In the first two applications, a general purpose loop scheduling routine is invoked to execute the parallel loop; while in the third application, a load balancing framework for the simultaneous analysis of related datasets was developed based on a dynamic loop scheduling approach, the choice of the loop scheduling method having a major effect on performance of the framework.

4.1 Profiling a quadrature routine

Automatic quadrature routines (AQRs) are often used in scientific computations such as multivariate statistics, finite element methods, particle physics, and other applications. An AQR is designed 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 be a single variable or a vector. Typically, the inputs to such a routine are: a description of the domain D, the code for the integrand $f(x)$, absolute and relative error tolerances (ϵ_a, ϵ_r), a limit to the number of function evaluations in case the error tolerances are not achievable by the AQR, and the quadrature rule to be used. The AQR returns *result*, hopefully satisfying $|result - I| \leq \max(\epsilon_a, \epsilon_r * |I|)$, an estimate *errest* of the absolute error in *result*, the actual number of function evaluations used, and a termination condition indicator. Since the AQR does not know the value of I, it attempts to satisfy $errest \leq \max(\epsilon_a, \epsilon_r * |result|)$.

The profiling of an AQR is an empirical study which attempts to highlight the accuracies (ϵ_a, ϵ_r) that are achievable by the routine and the costs involved (in terms of function evaluations), using various types of integrands for $f(x)$. The integrands are chosen such that the answers are known analytically to facilitate the computation of the true error in *result*, as well as the accuracy of *errest*. The use of profiles to evaluate AQRs has been advocated [31] as an alternative to the practice of running AQRs on a small set of test cases. A package for profiling two-dimensional AQRs that runs on a message-passing environment has been developed [18], but the environment is now obsolete. For this work, the control section of the package was reimplemented to employ dynamic loop scheduling and standard MPI.

Generating the profile of an AQR is a very simple but a time-consuming three-stage procedure. The first stage generates a large set of parameters where each element defines a sample integral with specific properties and error requirements; the second stage is an embarrassingly parallel loop in which each iterate invokes the AQR on a sample integral and determines the true error in *result* and the accuracy of *errest*; and the third stage generates summary statistics. The bulk of the execution time is spent in the second stage. The determinants for the number of loop iterates in the second stage are the following quantities: *nfam*—the number of integrand families, *ndif*—the number of difficulty levels for each integrand family, *neps*—the number of relative accuracy requirements, *nrul*—the number of quadrature rules to use, and *nsamp*—the number of samples to compute for each combination of integrand family, difficulty level, accuracy requirement, and quadrature rule. The total number of integrals to be evaluated is *nfam* * *ndif* * *neps* * *nrul* * *nsamp*. In order to control the granularity of the computations, *grpsize* integrals can be analyzed at a time. The number of iterates, therefore, is $N = nfam * ndif * neps * nrul * (nsamp/grpsize)$. The granularity of the computational task of an iterate of the loop in the second stage can be set from one integral per task ($grpsize = 1$, "small" granularity), and up to *nsamp* integrals per iterate ($grpsize = nsamp$, "large" granularity). Due to the differences in integrand families, difficulty levels, accuracy requirements, and quadrature

rule settings, the iterate execution times are highly variable even if the same number of integrals is evaluated by each iterate.

An AQR previously developed by one author for estimating integrals over triangulated domains [33] was selected for profiling. It is written in C and is linked with a general purpose loop scheduling routine [16]. The AQR uses the Lyness and Jespersen quadrature rules with degrees 1, 4, 6, 8 and 11 for triangles and incorporates nonlinear extrapolation. The routine was tested on fourteen families of integrands ($nfam = 14$), each family having three difficulty levels ($ndif = 3$). The integrand functions may be broadly classified as well behaved, oscillatory, C_0 function, with Gaussian peak or internal peak, or singular at an interior point or along an edge. The relative error tolerances 0.5×10^{-i}, $i = 1, \ldots, 6$, ($neps = 6$), and the degree 4, degree 6, degree 8 quadrature rules ($nrul = 3$) were specified. Three values for *nsamp* were chosen (50, 100, 200), with $grpsize = 10$ in each case; these values give rise to loops of sizes $N = 3{,}780$, 7,560 and 15,120, respectively, with each loop iterate making *grpsize* invocations of the AQR. An iterate analyzing a family of singular integrands and/or a high accuracy setting would require more integrand evaluations, and hence will execute longer than an iterate analyzing a family of well-behaved integrands or a low accuracy setting.

To investigate the effect of the location of time consuming iterates within the iteration space on the performance of the loop scheduling methods, four (4) evaluation sequences for the integrand families were chosen. In Sequence 1, the integrand families with severe singularities are evaluated first, followed by the families with less severe singularities, and the families with no singularities being evaluated last. In Sequence 2, families with severe singularities are in the middle of the sequence, preceded and succeeded by the families with less severe singularities, and the families without singularities making the head and tail of the sequence. Sequence 3 is the reverse of Sequence 1, and Sequence 4 is obtained from Sequence 2 by interchanging the positions of the families with severe singularities with the positions of the families without singularities.

The profiling experiments were conducted on the UltraMSPARC cluster using $P = 4$, 8, 16, 32 processors. Designed and constructed by the Mississippi State University Engineering Research Center, the UltraMSPARC is a 16-node cluster with Myrinet interconnects, each node with four (4) Sun UltraMSPARC II 400 MHz processors. The cluster has 32 gigabytes aggregate RAM, and uses the MPICH implementation of MPI. Other jobs were also running in the cluster along with the experiments, thus network traffic volume may have varied across runs. In order to reduce timing measurement biases, each simulation run was repeated 3 times, and the processor timings were averaged over these 3 runs.

The distribution of the useful work times when the profiling application is executed without load balancing is illustrated by Fig. 3, for $N = 15{,}120$ and $P = 32$. Load imbalance in the application is evident from this figure, for all the integrand family evaluation sequences. The graphs for other combinations of N and P are similar. The loop completion time without load balancing is essentially the finishing time of the processor that was assigned the most difficult integrand family.

Figures 4, 5, 6, 7 compare the parallel costs of profiling the AQR when using various loop scheduling methods. Each cost is the average cost of three runs; the costs

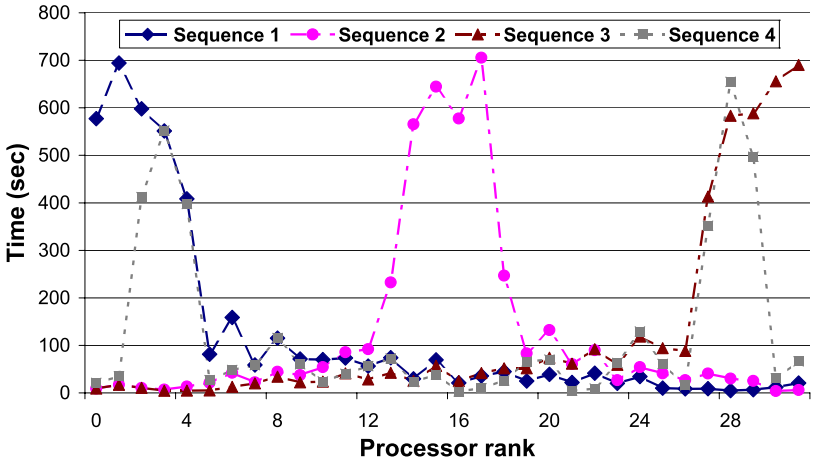


Fig. 3 Distribution of useful work times for AQR profiling without loop scheduling

Fig. 4 Parallel cost of profiling an AQR on Sequence 1 with loop scheduling

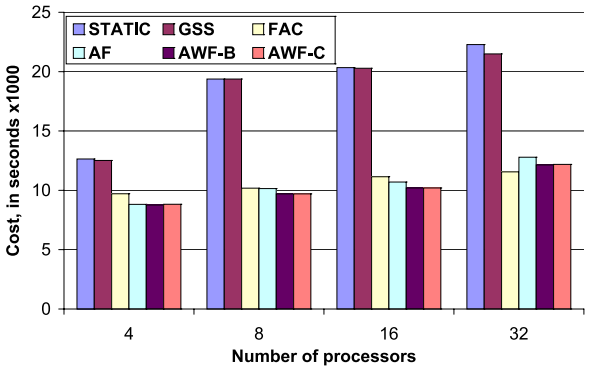
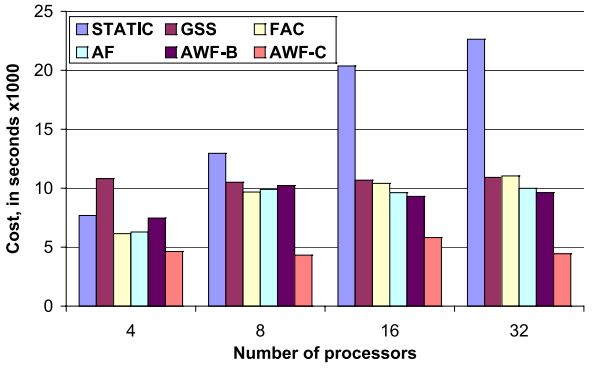


Fig. 5 Parallel cost of profiling an AQR on Sequence 2 with loop scheduling



for AWF-D and AWF-E have been excluded from the graphs since these methods had costs similar to AWF-B and AWF-C, respectively. The graphs indicate that AWF-C incurs the least cost overall.

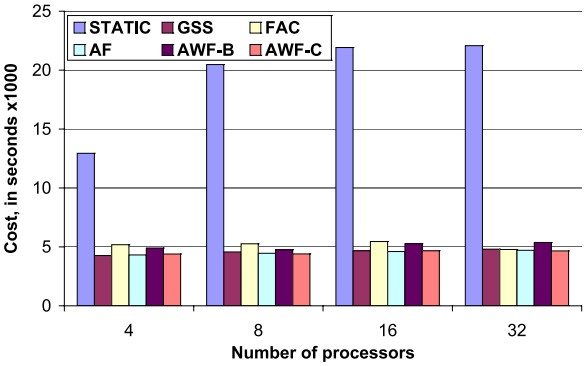


Fig. 6 Parallel cost of profiling an AQR on Sequence 3 with loop scheduling

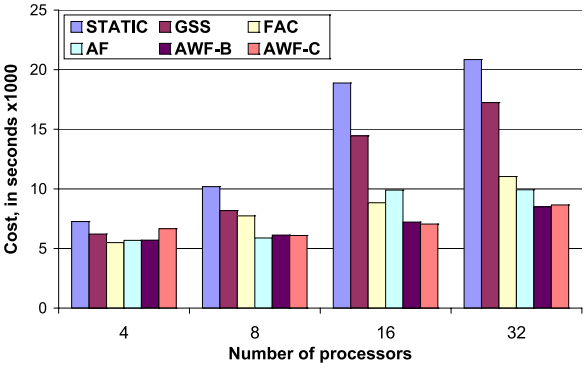


Fig. 7 Parallel cost of profiling an AQR on Sequence 4 with loop scheduling

When profiling with Sequence 3, all the methods, with the exception of STATIC, uniformly achieve the lowest costs. Recall that these methods schedule chunks in decreasing sizes; with Sequence 3, the time-consuming iterates are assigned to small chunks, which is ideal for load balancing. This result suggests a heuristic for load balancing a parallel loop on homogeneous processors when the distribution of the iterates execution times is known in advance: if sorting the iterates according to increasing execution time incurs little overhead, then scheduling the loop using a method like GSS or FAC that generates decreasing sized chunks may be very efficient.

When profiling with Sequences 1 and 4, GSS which initially allocates large chunks, performs poorly because these initial chunks consist of time-consuming iterates; the processors assigned to these chunks inevitably finish last. FAC is not affected severely as GSS since the initial FAC chunks are half the size of the largest GSS chunk. The adaptive factoring methods (AF and modified AWF) further allocate smaller initial chunks than FAC in determining first estimates of the performance parameters; hence, these adaptive methods are also not severely affected as in GSS. These results highlight a disadvantage of the decreasing size chunks strategy for scheduling a parallel loop with time-consuming iterates near the beginning of the loop; in this situation, small starting chunks are preferable.

The advantage of AWF-C over the other methods is evident when profiling with either Sequence 2 or 4. In particular, AF—the most recently introduced method de-

signed for parallel loops with variable iterate execution times, costs more than AWF-C. One possible explanation is that, with Sequences 2 and 4, the AF is incompatible with the strategy of allocating iterates in natural order to chunks; that is, if L is the number of chunks, and $k_1, k_2, \ldots, k_L$ are the chunk sizes, then the chunks are $[1..k_1]$, $[k_1 + 1..k_1 + k_2]$, $[k_1 + k_2 + 1..k_1 + k_2 + k_3]$, $\ldots$, $[(\sum_{i=1}^{L-1} k_i) + 1..N]$. This strategy assigns chunks from different regions in the iteration space to a processor. Intuitively, if the behavior of the iterates in two successive chunks assigned to a processor are totally different, then the predicted size of the second chunk based on the mean and standard deviation of the iterate execution times in the first chunk may not be appropriate. The AWF-C appears to be less affected by this strategy with Sequences 2 and 4 due to its use of the weighted averaging approach. The performance of AF for Sequences 2 and 4 may be improved by initially assigning blocks of iterates to processors; if there is load imbalance, a processor that finishes its initial share early will be assigned iterates from other blocks. In this manner, while a processor is not finished with its initial block, the chunks assigned to it will be contiguous and the behavior of the iterates from successive chunks are likely to be related. Results of preliminary experiments using this strategy coupled with a partitioned work queue of iterates are reported elsewhere [14].

4.2 Simulation of wave packet dynamics

Time-dependent wave packets are widely used to model various phenomena in physics. Classical approaches for computing the wave packet dynamics include space-time grids, basis sets, or combinations of these methods. An unstructured grid approach based on the Bohm hydrodynamic interpretation of quantum mechanics [12], the quantum trajectory method (QTM), has been implemented for a serial computing environment [29]. The QTM solves the quantum hydrodynamic equations using a moving weighted least-squares (MWLS) algorithm to compute needed derivatives. The solutions to the equations of motion give the quantum trajectories for "fluid particles" or pseudoparticles. An implementation of the QTM on a shared-memory environment employing OpenMP for loop parallelization has been developed, as well as a message-passing version that utilizes dynamic loop scheduling [13, 21, 22]. An outline of the simulation is given by the following pseudocode:

Initialize positions $r[\,]$, velocities $v[\,]$, and probability densities $\rho[\,]$
For each timestep s in turn
  For pseudoparticle $i = 1$ to N (**Loop 1**)
    Call MWLS $(i, r[\,], \rho[\,], Np_1, Nb)$
    Compute quantum potential $Q[i]$
  For pseudoparticle $i = 1$ to N (**Loop 2**)
    Call MWLS $(i, r[\,], Q[\,], Np_2, Nb)$
    Compute quantum force $f_q[i]$
  For pseudoparticle $i = 1$ to N (**Loop 3**)
    Call MWLS $(i, r[\,], v[\,], Np_2, Nb)$
    Compute derivative of velocity $dv[i]$
  For pseudoparticle $i = 1$ to N (**Loop 4**)
    Compute classical potential $V[i]$, force $f_c[i]$

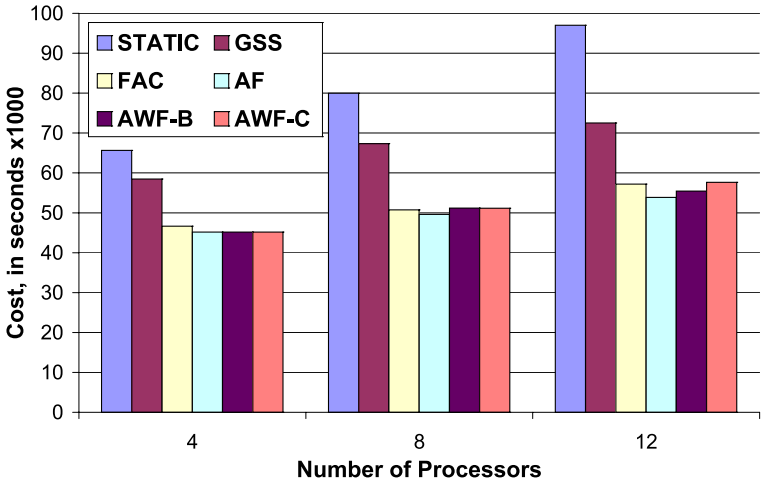


Fig. 8 Parallel cost of simulating wave packet dynamics using the quantum trajectory method

Output $s, r[], v[], \rho[], V[], f_c[], Q[], f_q[], dv[]$
For pseudoparticle $i = 1$ to N (**Loop 5**)
Update density $\rho[i]$, position $r[i]$ and velocity $v[i]$

The MWLS algorithm needed to compute $Q[\ ]$, $f_q[\ ]$, and $dv[\ ]$ solves an overdetermined linear system of size $Np \times Nb$. The execution profile of a straightforward serial implementation of the above algorithm indicates that up to 90% of the total time is spent in the MWLS routine called by **Loops 1–3**, which are parallel loops. However, due to adaptivity in the MWLS algorithm, the loop iterates perform varying amounts of computation, and that these amounts also change with the evolution of the time steps. Thus, adaptive loop scheduling is necessary. The simulation code is written in Fortran 90, and it invokes a general purpose loop scheduling routine [16] to execute **Loops 1–3**.

To compare the performance of the application with various loop scheduling methods, a free particle represented by a Gaussian wave packet with 1,501 pseudoparticles was simulated for 1,000 time steps. The timings used in the results below are averaged over 3 runs of the simulation on 4, 8 and 12 processors of the "Empire" Linux cluster at the Mississippi State University High Performance Computing Collaboratory. The cluster has a number of compute nodes with dual 1.0 GHz or 1.266 GHz Intel Pentium III processors, the nodes (32 in a rack) are connected via fast Ethernet, and the racks are connected with gigabit Ethernet uplinks. The cluster scheduler attempts to assign, if feasible, the processors requested by a job, from a single rack. Figure 8 graphs the parallel costs of the simulation when using the STATIC, GSS, FAC, AF, AWF-B and AWF-C to execute **Loops 1–3**. The figure indicates that the modified AWF methods achieve performance that is comparable to the performance of the other factoring methods.

4.3 Simultaneous analysis of multiple datasets on a cluster

The simultaneous analysis of a number of related datasets using a single statistical model is an important problem in statistical computing. A parameterized statistical model is to be fitted on multiple datasets. Definitive conclusions are hopefully achieved by analyzing the datasets together. A simple "one processor per dataset" parallel strategy is not suitable for this application due to wide differences in dataset analysis times, which can range from a few seconds to several hours, depending on the number of observations in a dataset. A processor assigned to a large dataset will finish long after those assigned to smaller datasets. Also, an "all processors working on one dataset at a time" strategy precludes the exploitation of the large number of processors available on typical clusters because of the limited degree of concurrency in the analysis procedure.

The authors developed a framework (Fig. 9) for dataset analysis [2, 5, 15] based on a "processor groups" strategy. The framework is to be configured with the routines for a specific statistical analysis problem and is submitted as a parallel job to a

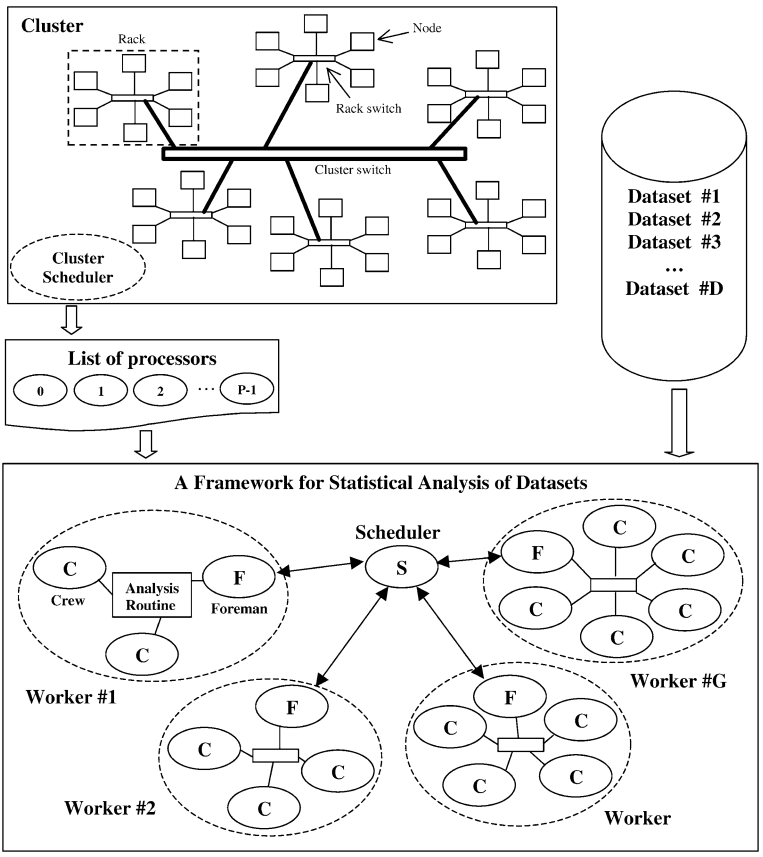


Fig. 9 A processor groups strategy for dataset analysis on a cluster

cluster. The cluster scheduler commits the number of processors requested by the job. Early during execution, the framework designates one of the processors as a dedicated scheduler S, which is responsible for managing three phases of the computations: (1) organizing the rest of the processors into groups of crew members C and appointing a foreman F for each group; (2) retrieving the datasets from disk and distributing these to the groups; and (3) scheduling the analysis of the datasets by the groups. Load imbalance among groups is expected to be induced by the differences in both the computational powers and loads of the processor groups, and the unpredictable network latencies or operating system interferences inherent in a cluster environment. Dynamic redistribution of datasets among groups is necessary during the third phase, in order to achieve high performance.

When the cluster scheduler assigns processors to a parallel job, the processors are typically fragmented across several racks, especially if a large number of processors is requested. This fragmentation is conveniently exploited by the framework to match the degree of concurrency in the analysis procedure with the appropriate number of processors. The scheduler S forms the processors residing in a rack into a processor group that acts as single worker, to carry out the analysis procedure on one dataset at a time. A very large number of processors in a rack (relative to the degree of concurrency in the analysis procedure) can be formed into two or more workers. However, if there are racks that contribute tiny numbers of processors, then the processors are combined together to form a single more powerful worker in order to avoid the possibility of "tiny" workers being assigned very large datasets. This manner of organizing processors by racks enables two levels of concurrency—the simultaneous processing of datasets, and the parallel execution of the analysis procedure for a single dataset, while exploiting the efficiency of communications among processors residing in a single rack. Load balancing can be employed on both concurrency levels, that is, *among* the processor groups and *within* a group.

After the processors assigned to the analysis job are identified and the processor groupings are established, the datasets are retrieved from disk for distribution to the processor groups. Keeping the datasets in memory offers some advantages over "on demand" retrieval of datasets from disk. If the datasets are not massive, moving a dataset from one processor to another within the cluster is at least an order of magnitude faster than retrieving the dataset from a filesystem outside the cluster. The framework initially distributes the datasets among the groups such that the total number of observations stored by a group is proportional to the number of processors in the group. To avoid the situation where all the big datasets are lumped together into one group, the distribution procedure is as follows. The datasets are sorted according to decreasing size. The largest datasets are simply distributed one for each group. Then, for the rest of the datasets in sorted order, the group which is farthest from its quota of observations is identified and the processor with the minimum number of observations in that group will store the dataset. This initial distribution strategy ensures that the big datasets are effectively scattered among the groups, and that the processors in a group store comparable numbers of observations.

The proportionality of the number observations stored by a group to the number of processors in the group before the start of the analysis is not a guarantee for good load balance among groups. This is because the number of observations may not be

good measure of the computational load of a dataset. Furthermore, the dynamic nature of a cluster environment also induces other types of load imbalance that must be addressed during the actual analysis of the datasets, necessitating their redistribution during runtime. Dynamic load balancing in the analysis framework is accomplished via loop scheduling, using a partitioned work queue, with some contextualization. In conventional loop scheduling, a parallel loop with N iterations is to be executed on P processors. Chunks of iterations are assigned to processors with the objective of minimizing the loop completion time. The sizes of chunks are determined according to a loop scheduling technique. Mapped to the present context of the dataset analysis, the N loop iterations correspond to the total number of observations, and the P processors correspond to the number of groups, a group being a single worker. A chunk of iterations is essentially a fraction f of the total N iterations; here, a chunk of observations will correspond to a *collection* of datasets where the number of observations is a fraction f of the total observation count. Using these correspondences, dynamic loop scheduling techniques are applicable in the present context, with the possible exception of AF which requires the measurement of individual iterate execution times. The correspondence between a single loop iteration and a observation may not be valid because the execution of individual loop iterations can be timed, while observations are not analyzed individually, but only collectively in a dataset.

As a preliminary performance test, the framework was configured for the analysis of gamma-ray burst (GRB) datasets using vector functional coefficient autoregressive (VFCAR) time series models, on the "Empire" cluster. GRBs are cosmic explosions that occurred in distant galaxies and are thought to be excellent "beacons" for tracing the structure of the early universe. Scientific satellites have been recording GRB time profiles for a number of years [11, 27], and the analysis of the collected datasets is currently an open problem in the astrophysics community. The small-scale analysis mentioned by this paper is an initial investigation of the usefulness of VFCAR models [25, 26] in describing features of GRBs. The computational challenges in this investigation and their partial resolution are presented elsewhere [2–5].

The test involved the analysis of 555 GRB datasets, with sizes ranging from 46 to 9,037 observations, on 64 processors of the cluster. The analysis was executed without dataset redistribution (STAT) and with dataset redistribution using the following loop scheduling techniques: modified fixed size chunking (mFSC), GSS, FAC and AWF-C Since the required parameters for the optimal FSC chunk size are not known, the chunk size in mFSC is chosen such that the number of chunks generated is the same as the number generated by factoring (FAC), that is, mFSC and FAC both generate the same number of scheduling events but different chunk sizes. This comparison experiment was submitted as a single 64-processor job so that the same set of processors (and therefore, the same processor groupings) is used by the four load balancing schemes.

Figures 10 and 11 summarize the performance of the framework under the STAT, mFSC, GSS, FAC, AWF-B and AWF-C load balancing schemes when heterogeneous processors were assigned to the experiment. In this instance, the 64 processors were spread across racks 8, 10, 14, 15 and 16 of the Empire cluster, with the racks contributing 20, 16, 16, 8 and 4 processors, respectively. Rack 8 had 1.0 GHz processors, while the rest of the racks had 1.266 GHz processors. The scheduler S formed five

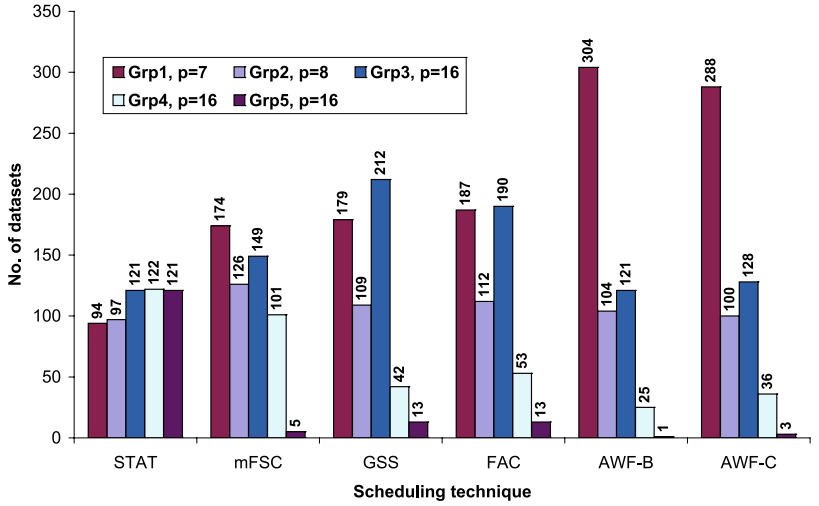


Fig. 10 Comparison of the final distribution of datasets among processor groups. The initial distribution of datasets is the same as the final distribution for STAT

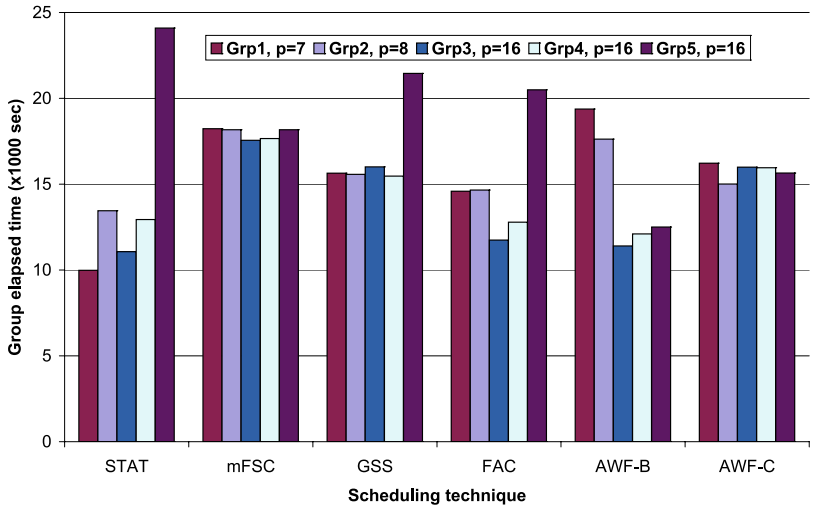


Fig. 11 Comparison of processor group elapsed times

groups: Group 1 with 7 processors split between racks 8 and 16 (hence, heterogeneous processors); Group 2 with 8 processors in rack 15; Group 3 with 16 processors in rack 14; Group 4 with 16 processors in rack 10; and Group 5 with 16 processors in rack 8. The scheduler S resided in rack 16.

Figure 10 indicates that except for STAT, datasets migrated from Groups 4 and 5 to Groups 1–3. Using the timings in Fig. 11 to calculate the parallel cost of the framework under each load balancing scheme, the percent cost improvements of the other methods over STAT (i.e., $100\times(cost(\text{STAT}) - cost(\text{method}))/cost(\text{STAT})$ are 24.3,

11.0, 14.9, 19.6 and 32.7, respectively, for mFSC, GSS, FAC, AWF-B and AWF-C. Thus, AWF-C achieved the highest cost improvement over STAT. This may be attributed to AWF-C's adaptive strategy of assigning work to processors, compared to the nonadaptive strategy of mFSC, GSS and FAC. AWF-C's better performance over AWF-B may be attributed to the more frequent adaptation of processor weights by AWF-C.

5 Concluding remarks

The adaptive weighted factoring method (AWF) was originally designed for scheduling a parallel loop in time-stepping scientific applications. During a time step, the method measures the time per iterate ratio in each processor. At the end of the step, the ratios are converted into processor weights for allocating chunk sizes in the succeeding time step. This paper proposes a modification to make the AWF also suitable for scientific applications that do not involve time steps. Essentially, the change is to utilize the ratios based on timings from earlier chunks, to adapt the processor weights for the succeeding chunks. With this revision, information from a previous run of the loop is no longer necessary. The modified method adjusts to load imbalance that occurs during loop runtime since the processor weights are updated while the loop is being executed. Tests of the modified AWF in three nontrivial applications—the profiling of a quadrature routine, simulation of wave packet dynamics using the quantum trajectory method, and statistical analysis of multiple datasets, confirm that the performance of the method is comparable, and in certain cases, superior to the performance of the most recently introduced adaptive factoring method. Work is on-going to implement a library of dynamic loop scheduling methods, including the modified AWF.

Acknowledgements We thank Ravi Vadapalli, Charles Weatherford and Jianping Zhu for their collaboration on the simulation of wave packet dynamics. We also thank Jane Harvill and John Lestrade for their collaboration on the analysis of gamma-ray burst datasets using vector functional coefficient autoregressive time series models.

References

1. Balasubramaniam M, Barker K, Banicescu I, Chrisochoides N, Pabico JP, Cariño RL (2004) A novel dynamic load balancing library for cluster computing. In: Proceedings of the 3rd international symposium on parallel and distributed computing, in association with the international workshop on algorithms, models and tools for parallel computing on heterogeneous networks (ISPDC/HeteroPar'04). IEEE Computer Society Press, Los Alamitos, pp 346–352
2. Banicescu I, Cariño RL (2005) Addressing the stochastic nature of scientific computations via dynamic loop scheduling. Electron Trans Numer Anal 21:66–80
3. Banicescu I, Cariño RL, Harvill JL, Lestrade JP (2005) Simulation of vector nonlinear time series on clusters. In: Proceedings of the 19th international parallel and distributed processing symposium—the 6th international workshop on parallel and distributed scientific and engineering computing (IPDPS-PDSEC 2005). IEEE Computer Society Press, Los Alamitos (on CD-ROM)
4. Banicescu I, Cariño RL, Harvill JL, Lestrade JP (2005) Computational challenges in vector functional coefficient autoregressive models. In: Proceedings of the international conference on computational science 2005 (ICCS 2005), part I. Springer, Berlin, pp 237–244

5. Banicescu I, Cariño RL, Harvill JL, Lestrade JP (2005) Vector nonlinear time-series analysis of gamma-ray burst datasets on heterogeneous clusters. Sci Program 13(2):415–422
6. Banicescu I, Flynn-Hummel S (1995) Balancing processor loads and exploiting data locality in n-body simulations. In: Proceedings of the 1995 ACM/IEEE conference on supercomputing. ACM, New York (on CD-ROM)
7. Banicescu I, Liu Z (2000) Adaptive factoring: dynamic scheduling method tuned to the rate of weight changes. In Proceedings of the high performance computing symposium (HPC 2000), pp 122–129
8. Banicescu I, Velusamy V (2001) Performance of scheduling scientific applications with adaptive weighted factoring. In: Proceedings of the 15th IEEE international parallel and distributed processing symposium—10th heterogeneous computing workshop (IPDPS-HCW 2001). IEEE Computer Society Press, Los Alamitos (on CD-ROM)
9. Banicescu I, Velusamy V (2002) Load balancing highly irregular computations with the adaptive factoring. In: Proceedings of the 16th IEEE international parallel and distributed processing symposium—11th heterogeneous computing workshop (IPDPS-HCW 2002). IEEE Computer Society Press, Los Alamitos
10. Banicescu I, Velusamy V, Devaprasad J (2003) On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Clust Comput: J Netw Softw Tools Appl 6(3):215–226
11. BATSE. (2005) Burst and transient source experiment BATSE home page. http://www.batse.msfc.nasa.gov/batse
12. Bohm D (1952) A suggested interpretation of the quantum theory in terms of "hidden" variables. Phys Rev 85:166–193
13. Brook RG, Oppenheimer PE, Weatherford CA, Banicescu I, Zhu J (2001) Solving the hydrodynamic formulation of quantum mechanics: A parallel MLS method. Int J Quantum Chem 85(4–5):263–271
14. Cariño RL, Banicescu I (2002) Load balancing parallel loops on message-passing systems. In: Akl SG, Gonzales T (eds) Proceedings of the 14th IASTED international conference on parallel and distributed computing and systems (PDCS 2004). Acta Press, Calgary, pp 362–367
15. Cariño RL, Banicescu I (2005) A framework for statistical analysis of datasets on heterogeneous clusters. In: Proceedings of the 2005 IEEE international conference on cluster computing. IEEE, New York (on CD-ROM)
16. Cariño RL, Banicescu I (2005) A load balancing tool for distributed parallel loops. Clust Comput 8(4):313–321
17. Cariño RL, Banicescu I, Rauber T, Runger G (2004) Dynamic loop scheduling on processor groups. In Proceedings of the 17th international conference on parallel and distributed computing systems (PDCS 2004), pp 78–84. International Society for Computers and Their Applications
18. Cariño RL (1992) Numerical integration over finite regions using extrapolation by nonlinear sequence transformations. PhD thesis, La Trobe University, Melbourne, Australia
19. Cariño RL, Banicescu I (2006) A Dynamic load Balancing tool For one- and Two-dimensional parallel loops. In: Proceedings of the 5th international symposium on parallel and distributed computing. IEEE Computer Society, Los Alamitos, pp 107–114
20. Cariño RL, Banicescu I, Lim H, Williams N, Kim S (2006) Simulation of a hybrid model for image denoising. In: Proceedings of the 20th international parallel and distributed processing symposium. IEEE Computer Society, Los Alamitos (on CD-ROM)
21. Cariño RL, Banicescu I, Vadapalli RK, Weatherford CA, Zhu J (2003) Parallel adaptive quantum trajectory Method for Wavepacket simulation. In: Proceedings of the 17th international parallel and distributed Processing Symposium—4th workshop on parallel and distributed scientific and engineering applications. IEEE Computer Society Press, Los Alamitos (on CD-ROM)
22. Cariño RL, Banicescu I, Vadapalli RK, Weatherford CA, Zhu J (2004) Message passing parallel adaptive quantum trajectory method. Kluwer Academic, Dordrecht, pp 127–139
23. Flynn-Hummel S, Schmidt J, Uma RN, Wein J (1996) Load-sharing in heterogeneous systems via weighted factoring. In: Proceedings of the 8th annual ACM symposium on parallel algorithms and architectures, pp 318–328
24. Flynn-Hummel S, Schonberg E, Flynn LE (1992) Factoring: A method for scheduling parallel loops. Commun ACM 35(8):90–101
25. Harvill JL, Ray BK (2005) A note on multi-step forecasting with functional coefficient autoregressive models. Int J Forecast 21(4):717–727
26. Harvill JL, Ray BK (2006) Functional coefficient autoregressive models for vector time series. Comput Stat Data Anal 50(12):3547–3566

27. HETE. (2005) High energy transient explorer HETE home page. http://space.mit.edu/HETE
28. Kruskal CP, Weiss A (1985) Allocating independent subtasks on parallel processors. IEEE Trans Softw Eng 11(10):1001–1016
29. Lopreore CL, Wyatt RE (1999) Quantum wave packet dynamics with trajectories. Phys Rev Lett 82:5190–5193
30. Luke E, Banicescu I, Li J (1998) The optimal effectiveness metric for parallel application analysis. Inf. Process. Lett., special issue on parallel models 66(5):223–229
31. Lyness J, Kaganove J (1976) Comments on the nature of automatic quadrature routines. ACM Trans Math Softw 2:65–81
32. Polychronopoulos CD, Kuck DJ (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
33. Cariño RL, Robinson I, de Doncker E (1994) Adaptive cubature over a collection of triangles using the d-transformation. J Comput Appl Math 50:171–183
34. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel computers. IEEE Trans Parallel Distrib Syst 4(1):87–98

Ricolindo L. Cariño finished his Ph.D. from La Trobe University, Melbourne, Australia in 1993. He was an Associate Professor of Computer Science in the University of the Philippines Los Banos until 2002. Currently, he is a research faculty in the Center for Advanced Vehicular Systems, Mississippi State University. His interests include high performance computing for scientific and engineering applications. He is a senior member of the IEEE.

Ioana Banicescu is a professor in the Department of Computer Science and Engineering at Mississippi State University (MSU). She received the Dipl. in Engineering (Electronics and Telecommunications) from Polytechnic University, Bucharest, and the M.S. and the Ph.D. degrees in Computer Science from Polytechnic University, New York. Professor Banicescu's research interests include parallel algorithms, scientific computing, scheduling theory, load balancing algorithms, performance analysis, optimization, and prediction. She also holds a research faculty position in the Center for Computational Sciences at MSU, where she presently is involved in the development of dynamic scheduling algorithms for scalable parallelization and performance optimization of problems in computational sciences. Professor Banicescu is the recipient of a number of awards for research and scholarship including the National Science Foundation (NSF) Early CAREER award, and three NSF Information Technology awards. She serves on steering and program committees of a number of international conferences, symposia and workshops, and on the Executive Board of the IEEE Technical Committee on Parallel Processing (TCPP). In 2004, Bagley College of Engineering at MSU has recognized Professor Banicescu's scholarly contributions by awarding her the Outstanding Faculty Engineering Research Award, and in 2005 she received the University Faculty Research Award for College of Engineering, and the Hearin Eminent Scholar Award.