International Journal of Parallel Programming, Vol. 22, No. 6, 1994

Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors[1]

Jie Liu,[2] Vikram A. Saletore,[3] and Ted G. Lewis[4]

Received August 16, 1993

In this paper we present Safe Self-Scheduling (SSS), a new scheduling scheme that schedules parallel loops with variable length iteration execution times not known at compile time. The scheme assumes a shared memory space. SSS combines static scheduling with dynamic scheduling and draws favorable advantages from each. First, it reduces the dynamic scheduling overhead by statically scheduling a major portion of loop iterations. Second, the workload is balanced with a simple and efficient self-scheduling scheme by applying a new measure, the *smallest critical chore size.* Experimental results comparing SSS with other scheduling schemes indicate that SSS surpasses other scheduling schemes. In the experiment on Gauss-Jordan, an application that is suitable for static scheduling schemes, SSS is the only self-scheduling scheme that outperforms the static scheduling scheme. This indicates that SSS achieves a balanced workload with a very small amount of overhead.

KEY WORDS: Parallel loops; self-scheduling; scheduling overhead; load balancing; shared-memory multiprocessors.

[1] This research has been supported in part by the National Science Foundation under Contract No. CCR-9210568.

[2] Department of Computer Science, Western Oregon State College, Monmouth, Oregon 97361.

[3] Department of Computer Science, Oregon State University, Corvallis, Oregon 97331.

[4] Current Address: Computer Science Department, Naval Postgraduate School, Monterey California 93943.

0885-7458/94/1200-0589$07.00/0 © 1994 Plenum Publishing Corporation

1. INTRODUCTION

A difficult problem in the concurrent execution of a set of independent tasks on a parallel computer, particularly when the tasks have variable length execution times, is choosing a schedule minimizing the total execution time. To execute a set of tasks on a parallel computer, we have to specify, for tasks, the processors that execute them and the order in which they are executed. Clearly, different specifications may yield different execution times. Since one of the main reasons we employ parallel computers is to reduce the total execution time, specifications rendering short completion times are always desirable. A *schedule* is a specification listing, for each task, the processor that executes the task and the order in which different tasks are executed.

The scheduling problem has attracted the attention of many researchers.[1-9] Many results are designed primarily to satisfy a partial ordering among a set of tasks scheduled to preserve the data dependencies defined by sequential execution of the set of tasks.[10] In addition, the execution time of each task is assumed to be either known or identical.[2,10]

In many scientific applications, a set of independent tasks typically exists in a *parallel loop*, i.e., a loop in which each iteration is independent of all others. Loops are one of the richest sources of parallelism and can be found in many scientific applications.[5,7] A parallel loop is also called a *DoAll* loop that has no cycles in its dependence graph.[11] Iterations in a parallel loop are independent and can be executed in any order. Parallel *Do* and SPREAD *Do* in PCF Fortran and Butterfly Fortran are some of the other examples of parallel loops.

A set of tasks is independent if there is no data dependence between any pair of tasks. Since a set of independent tasks can be referred by using a parallel loop in which each iteration is mapped to a task, the terms *parallel loop* and *set of independent tasks* are used interchangeably in sequel.

This paper discusses the problem of scheduling a set of tasks that are *independent* and that have *variable length execution times* not necessarily known at compile time. It is important to note that independent tasks can be executed in any order or simultaneously without affecting the final result. In addition, variable length iteration execution times are expected for real applications.[7]

Based on whether the execution times of different iterations are the same or not, a parallel loop can be categorized as *uniform* or *nonuniform*. A parallel loop is uniform if the execution times of each of its iterations are roughly the same. To schedule uniform parallel loops for maximum performance, an equal number of iterations is assigned to each processor (assuming that processors start to execute the loop at the same time).

To schedule nonuniform parallel loops for maximum performance, the workloads among processors must be balanced with low scheduling overhead. A balanced workload is the key to the performance because a loop is finished only after all its iterations have been executed. Since the execution times vary from one iteration to another, assigning an equal number of iterations to each processor, as we do in scheduling uniform parallel loops, may result in one processor finishing much later than the rest of the processors. We call this last finished processor the *critical processor* p_c. Such an unbalanced workload results in poor processor utilization that degrades the performance significantly.

Many methods have been proposed to parallelize a wide range of serial loops.[5,11,12] We assume that the loop has been parallelized and parallel loop nests have been coalesced.[5] Parallel loops with variable length iteration execution times can be found in applications such as Monte Carlo calculations, sparse matrix computation, numerical partial differential equation, and image processing applications. The execution time of a complicated operation may also greatly depend on its operands. For example, integer multiplication under 2's complement representation using Booth's algorithm[13] computes 263×127 faster than $263 \times (-54)$. When a parallel loop encloses a triangular serial loop (e.g., matrix Adjoin-Convolution shown in Ref. 3), iterations of the parallel loop have different execution times. Even when iterations of a parallel loop execute exactly the same statements, the execution times of different iterations may differ due to memory access interference such as cache misses, page faults, difference in execution cost depending on its operands, processor latency, and other "random events."[5,7,8]

The rest of the paper is organized as follows. In the next section we review related work in self-scheduling. SSS strategy is presented in Section 3 and compared with other schemes. Section 4 offers a list of modifications on SSS to further improve its performance and flexibility. In Section 5 we present experimental results. We summarize and conclude with a discussion in Section 6.

2. RELATED WORK

Scheduling has been studied by several researchers in a theoretical context.[2,10] Scheduling schemes can be classified as either static scheduling schemes or dynamic scheduling schemes. Static scheduling schemes assign iterations to processors at compile time while dynamic scheduling schemes do not determine the assignment of tasks until the execution is underway. The advantage of static scheduling schemes is that they impose no runtime overhead. The main drawback of static scheduling schemes is that they are unable to respond to an imbalance in the workload among the processors. Dynamic scheduling schemes are designed to alleviate this problem.

In the presence of variable length iteration execution times, dynamic approaches are in principle superior with respect to load balancing.[5] However, this comes at the cost of additional runtime scheduling overhead. The common approach of dynamic scheduling of parallel loops is the self-scheduling technique which maintains a global list of tasks.[1,3,5,7,8,14] An idle processor selects and removes the next task from the list and executes it. That is, the processors "self-schedule" themselves as the program executes.[5,8,15] In the rest of this section we show how to schedule N iterations of a parallel loop L on a p-processor parallel computer using some of the well known self-scheduling schemes developed by other researchers.

2.1. Static Scheduling Schemes

2.1.1. Static Chunk (SC)

The *static chunk* assigns each processor with $\lceil N/p \rceil$ consecutive iterations at compile time. Except in cases when iteration execution times are roughly the same, such an assignment of iterations to processors may introduce an unbalanced workload.

2.1.2. Round Robin (RR)

Round Robin is a modification of SC. Rather than assigning a processor with a consecutive block of iterations, iterations are assigned to processors in a round-robin fashion, i.e., iteration i is assigned to processor $i \bmod p$. This approach may produce a more balanced schedule than SC only for some parallel loops. This is because all processors execute the same number of iterations, about N/p to be exact, and clearly two processors assigned with the same number of iterations may not finish at the same time for nonuniform parallel loops. In addition, this approach may suffer a low cache hit ratio due to the way data is accessed.

2.2. Self-Scheduling Schemes

2.2.1. Pure Self-Scheduling (PSS)

In *pure self-scheduling* (PSS) a processor fetches one iteration at a time during runtime when it becomes idle. PSS always achieves a well balanced workload. However, this well balanced load does not always yield a good performance because the amount of scheduling overhead is proportional to the number of iterations of the scheduled parallel loop. For fine-grain parallel loops, this amount could be large compared to the cost of computation. In addition, the high frequency of mutually exclusive access to shared variables, such as the loop index, may cause memory contention and seriously degrade performance. PSS may be appropriate for loops with

relatively few iterations but long variable length execution times compared to the overhead. Ni and Wu[16] have performed an extensive study of the trade-offs between scheduling overhead and workload balancing.

2.2.2. *Chunk Self-Scheduling (CSS)*

The *chunk self-scheduling* (CSS) is designed to overcome the high scheduling overhead problem of PSS. CSS assigns a fixed number k iterations (chunk) at a time to an idle processor. When $k = 1$, CSS becomes PSS. When $k = \lceil N/p \rceil$, this scheme can be realized using SC. When scheduling nonuniform parallel loops, this approach may cause an unbalanced workload. The main drawback of CSS is its performance depends on the chunk size and the characteristics of each loop which may be unknown even at runtime. Too large a chunk size may cause a load imbalance while too small a chunk size may increase the overhead and memory contention. Worse yet, even for the same loop, the execution time does not monotonically increase or decrease with the chunk size.

2.2.3. *Guided Self-Scheduling (GSS)*

Polychronopoulos and Kuck[5] show that there cannot be an optimal value of k in CSS for even the simplest cases and present the *guided self-scheduling* (GSS) scheme. In GSS, $1/p$ of unscheduled iterations is assigned to an idle processor. When processors begin to execute the loop at different times, GSS generates a balanced schedule with low overhead for uniform parallel loops. When applying GSS to a nonuniform parallel loop with N iterations, assigning N/p iterations or close to N/p iterations to the first several fetching processors may cause an unbalanced workload. In addition, near the end of the scheduling process, GSS assigns many iterations in groups of one or two. This increases the scheduling overhead. Thus, GSS(t) was proposed to avoid the problem by allocating no less than t iterations at a time to an idle processor.[5] Clearly, an optimal value of t that minimizes the overhead while maintaining a balanced workload is both application and hardware dependent.[5] Furthermore, when the granularity of a parallel loop is small, the accesses to the shared variables such as loop index in GSS algorithm may become a bottleneck on a system with a large number of processors.

2.2.4. *Trapezoid Self-Scheduling (TSS)*

Tzen and Ni[8] proposed the *trapezoid self-scheduling* (TSS) algorithm to improve GSS. TSS(N_s, N_f) assigns the first N_s iterations of a loop to the processor starting the loop and the last N_f iterations to the processor performing the last fetch, where N_s and N_f are both specified by either the programmer or the compiler. The number of iterations assigned to a

processor between the first and the last fetch is then between N_s and N_f and linearly decreases based on some step d with each assignment. [d is determined by N_s, N_f, and N. The number of processors p is not considered unlcss the values of N_s or N_f is related to p.]

Tzen and Ni proposed TSS($N/2p$, 1) as a general selection of N_s and N_f. TSS may yield an unbalanced workload because the difference between the number of iterations assigned to two processors on their last fetch can be as large as $p \times d$. For example, when $p = 256$ and $d = 2$, the difference is 512 iterations.

2.2.5. Factoring

Hummel *et al.* introduce *Factoring* in Ref. 3. In Factoring, fixed sized chunks of iterations are allocated to processors in batches (a set of p consecutive chunks forms a batch). With *Factoring*, $c_i = \lceil R_i/2p \rceil$ iterations are allocated to each of the p processors at each scheduling step i, where R_i is the number of iterations remaining, with R_1 initially set to the total number of iterations. R_i is decremented at each scheduling step to obtain $R_{i+1} = R_i - pc_i$. The chunk size of each batch is halved from that of the previous batch. Thus, the algorithm evenly distributes half of the remaining iterations at each successive allocation. The basic idea of Factoring is the following: achieving an overall optimal finishing time requires, for each batch scheduled, enough work being left to smooth out the uneven finishing time of the batch. The main focus of Factoring is to achieve a balanced workload.

2.2.6. Affinity Scheduling

Affinity Scheduling demonstrates the benefit of processor affinity.[17] Affinity Scheduling divides the N iterations of a parallel loop into p chunks within $\lceil N/p \rceil$ iterations each. The ith such chunk is placed on the local work queue of processor i. An idle processor removes $1/k$, where k is suggested to be equal to p, of the iterations from its local work queue and executes them. When a processor's work queue becomes empty, it finds "the most loaded processor," removes $\lceil 1/k \rceil$ of the iterations from that processor's work queue, and executes them. [Note that "the most loaded processor" has to mean the processor that has the largest number of unscheduled iterations, rather than the processor that needs a longest time to finish its iterations, because we cannot determine how much time is needed to finish a given number of iterations.]

3. SAFE SELF-SCHEDULING ALGORITHM

In this section we will first introduce a self-scheduling scheme called *Safe Self-scheduling* (SSS).[24] The theorems that support SSS and the basis

for combining static scheduling and self-scheduling in SSS are then given. We devote the last part of this section comparing SSS with the Factoring scheme due the similarities between the implementations of the two.

To facilitate our presentation, we assume that the parallel loop L's iteration execution times follow an unknown probability distribution $D(L)$ with expectation μ, standard deviation σ, maximum execution time E_{max}, and minimum execution time E_{min}. We define a *chore* as a set of *consecutive* iterations defined by a starting and an ending iteration number; a *fetching processor* is a processor that modifies the global variables, such as the loop index to obtain more work in the form of a chore; the *critical chore* is the last finished chore; the *critical processor* is the processor that executes the critical chore; $e(i)$ is the workload of iteration i, i.e., the execution time needed by iteration i; and $E(T_j)$ is the workload of processor j, i.e., the sum of workload of iterations executed by processor j.

Throughout the rest of the paper, we assume that the number of iterations $N \gg p$; the values of N and p are known before the loop is executed; tasks are nonpreemptive; the processors of the parallel machine are homogeneous; and the parallel loop is executed in a dedicated environment.

3.1. Safe Self-Scheduling (SSS)

The basic principle of SSS is to assign each processor the largest number m of consecutive iterations having a cumulative execution time just exceeding the average processor workload E/p, i.e., $\sum_{i=s}^{s+m-1} e(i) < E/p \leqslant \sum_{i=s}^{s+m} e(i)$ where $E = \sum_{i=1}^{N} e(i)$ and s is some starting iteration number of the chore. We call m the *smallest critical chore size* because adding any more iterations to this chore further imbalances the schedule. On the other hand, assigning a processor any less iterations results in the processor being allocated a less-than-average processor workload and another or some other processors have to execute more iterations. Note that E may be estimated using the statistical information on the execution times of the tasks due to executing different branches of the parallel loop body, the expected execution time of tasks, the total number of tasks, and the number of processors. When a parallel loop is executed on a dedicated environment, the total number of tasks and the number of processors are known before the computation. The expected execution time of tasks and execution times of different branches of the parallel loop body can be obtained through profiling or previous runs.

In the implementation of SSS, the size of chores in the first batch, denoted by CS_0, is $\alpha \times N/p$, where $0 < \alpha \leqslant 1$. α is called the *Allocation Factor*, and it determines the fraction of the unscheduled iterations allocated during each batch. The chores in the first batch are assigned to

processors at compile time. At runtime, when a processor finishes the iterations assigned to it at compile time, the ith fetching processor is then assigned a chore of $\max((1-\alpha)^{\lceil i/p \rceil} \times N/p \times \alpha, k)$ iterations, where k is used to control the granularity. A method for accurately calculating α is given later.

After the value for α is determined, the SSS can be implemented as follows:

1. (a) Before starting the statically assigned iterations, one processor (say processor 0) calculates the starting iteration numbers for the chores scheduled in the dynamic scheduling phase, stores them in an array, say *chore_list*, and appends it with p 0's.
   (b) The same processor sets the shared variable *count* to 0 then starts to execute the chore assigned to it statically.
   (c) All other processors perform computation on the statically scheduled chores.
2. During the dynamic scheduling phase an idle processor does the following in the given order:
   (a) begins mutual exclusion;
   (b) copies the value of *count* to i and increments *count*;
   (c) ends mutual exclusion;
   (d) if $chore_list[i+1] > 0$, then executes the chore defined by $chore_list[i]$ and $chore_list[i+1]-1$.

For systems such as IBM's RP3 and the Ultracomputer[18] that can perform *fetch&add* atomically, the first three items of step 2 can be reduced to $i = fetch\&add(count, 1)$.

Note that the calculation of chores can be modified to suit the characteristics of the loop executed to best realize the basic principle of SSS. Other scheduling schemes such as GSS, TSS, or Factoring can also be used to calculated the chore sizes scheduled in the dynamic scheduling phase.

3.2. Theoretical Basis for SSS

Definition (Balanced Workload). A schedule that maps iterations of a parallel loop to processors of a parallel computer is *balanced* if the difference in workload between any two processors is no greater than the maximum execution time of a loop iteration.

Theorem 1. If (i) m is selected in the way that for the first m iterations we have $\sum_{i=1}^{m-1} e(i) < E/p \leq \sum_{i=1}^{m} e(i)$, and (ii) all p processors begin to execute the loop at the same time, then (a) if a processor, say p_1, executes

$m-1$ iterations then it finishes no later than the critical processor p_c and (b) if the processor executes m iterations, the difference in workload between any two processors is less than $p/(p-1)\, E_{max}$.

Proof. Let $E(T_1)$ be the workload for processor p_1 and E_{rem} be the maximum workload of the other $p-1$ processors on the best possible distribution of the remaining iterations;

(a) If p_1 executes $m-1$ iterations then we have

$$E_{rem} = \frac{E - E(T_1)}{p-1} = \frac{E - \sum_{i=1}^{m-1} e(i)}{p-1} > \frac{E - E/p}{p-1} = \frac{E}{p} > E(T_1) = \sum_{i=1}^{m-1} e(i)$$

What is indicated by $E_{rem} > E(T_1)$ is that the processor p_1 finishes no later than the critical processor p_c. That is, the critical processor p_c must be one of the other $(p-1)$ processors different from p_1.

(b) If p_1 executes m iterations then $E(T_1) = \sum_{i=1}^{m}$. Let $E/p \leqslant \sum_{i=1}^{m} e(i)$ be represented as $\sum_{i=1}^{m} e(i) = E/p + \beta$, where $0 \leqslant \beta < E_{max}$, then $E_{rem} = (E - E/p - \beta)/(p-1) = E/p - \beta/(p-1)$. The difference in workload between processor p_1 and any other processors is less than $E(T_1) - E_{rem} = E/p + \beta - E/p - \beta/(p+1)$, which is $\beta[p/(p-1)]$. Since $\beta < E_{max}$, the difference in workload between any two processors is less than $p/(p-1)\, E_{max}$.

Theorem 1 states that assigning m consecutive iterations to the first fetching processor, when $\sum_{i=1}^{m} e(i) - E/p \leqslant [(p-1)/p]\, E_{max}$, achieves balanced workload with minimum scheduling overhead since the processor only fetches once and the difference in finish times between any two processors is less than E_{max}. The difference in workload between any two processors is no greater than E_{max}, by our definition, the workload is balanced. When $\sum_{i=1}^{m} e(i) - E/p > [(p-1)/p]\, E_{max}$, the difference in workload between any two processors is less than $p/(p-1)\, E_{max}$ and can be considered to be very well balanced for large p. However, it is not generally possible to so determine m since $e(i)$ can only be known after the task t_i has been executed; therefore, m can only be estimated. Theorems 2 and 3 specify the lower and upper bound of m.

Theorem 2. If processor p_j executes no more than $(E/p)/E_{max} - 1$ iterations and all the processors start to execute the loop at the same time, then processor p_j will not be the critical processor.

Proof. Let $E(T_j)$ be the workload of processor p_j, then

$$E(T_j) \leqslant \left(\frac{E/p}{E_{max}} - 1\right) E_{max} = \frac{E}{p} - E_{max}$$

The average workload for the other $p-1$ processors then is $E-E(T_j)/(p-1)$. In addition, $E-E(T_j)/(p-1) \geqslant E-(E/p-E_{max})/(p-1)=E/p+E_{max}/(p-1)>E/p$. That is, there must exist at least one processor that has a workload greater than $E(T_j)$, therefore, processor p_j will not be the last one to finish.

According to Theorem 2, assigning a chore with less than $(E/p)/E_{max}-1$ iterations to a processor guarantees that this particular chore will not imbalance the schedule. Therefore, $(E/p)/E_{max}-1$ is called the *safe chore size*. Since it is desirable to assign chores with as many iterations as possible while maintaining load balance, chores with less than $(E/p)/E_{max}-1$ iterations should never be considered.

Theorem 3. If (i) all the processors start to execute the loop at the same time; (ii) the loop body consists of an if-then-else statement and *prob*(*then*) is the probability of executing the *then branch* that has an execution time of E_{max} and is the same for all the iterations; (iii) processor p_j is assigned a chore of size N/p and more than $N/p \times prob(then)$ iterations in the chore have a workload E_{max}; and (iv) $E_{max}>2E_{min}$, then the workload cannot be balanced.

Proof. The average workload of a processor is:

$$\frac{E}{p}=\frac{N(prob(then)\,E_{max}+prob(else)\,E_{min})}{p}$$

Let $N/p \times prob(then)+1$ iterations (out of N/p iterations assigned to processor p_j) have a workload of E_{max}, then there must be a processor that has no more than $N/p \times prob(then)-1$ iterations having a workload of E_{max}. The minimum difference in workload between the two processors is $2(E_{max}-E_{min})$, which is greater than E_{max}. Then according to our definition the workload is not balanced.

Usually, for static scheduling, N/p iterations are assigned to a processor. When the execution times of iterations vary, chores of the same size may result in different finishing times. Only if iterations assigned to one processor happen to have more iterations having a long execution time, the workload cannot be balanced. For this reason, N/p is called the *risk chore size*.

For a general approach we propose to select the first chore that has a size such that the probability that a processor may or may not perform an additional fetch to be equal (see Fig. 1). For loops where the execution times follow Bernoulli distribution with E_{max} having probability $prob(E_{max})$ and being constant for all the iterations, the size of the first chore is the

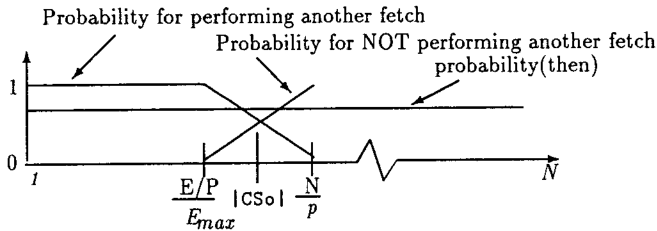

Fig. 1. Safe self-scheduling, calculation of the first chore size.

average of the safe chore size and the risk chore size. Using $\mu \times N$ to replace its statistical equivalence E we have

$$CS_0 = \frac{\frac{N}{p} + \frac{\mu N}{E_{max}\, p}}{2} = \frac{N}{p} \frac{\left(1 + prob(E_{max}) + \frac{prob(E_{min})\, E_{min}}{E_{max}}\right)}{2} \tag{1}$$

$$\alpha = \frac{1 + prob(max) + \frac{prob(min)\, E_{min}}{E_{max}}}{2} \tag{2}$$

Note that, by assigning a larger number of iterations than the safe chore size, we have accepted a moderate amount of risk of imbalance in exchange for a lower overhead. In cases where the scheduling overhead is small compared to the iteration execution times, a smaller value of α may be used to balance the workload. The relationship among α, workload balance, and scheduling overhead is that a small α generally offers a well balanced workload with the cost of high scheduling overhead.

The smallest critical chore size can be calculated according to the theorem given here if we assume that the execution time of an iteration is independent and all the iterations have their execution times follow the same distribution function.

Theorem 4. For a given confidence factor $c \geqslant \sqrt{2\ln(p)}$ and a set of n-iteration chores where n is given by

$$n = \frac{2\mu^2 \frac{N}{p} + c^2\sigma^2 - \sqrt{\left(2\mu^2 \frac{N}{p} + c^2\sigma^2\right)^2 - 4\mu^2 \left(\frac{\mu N}{p}\right)^2}}{2\mu^2} \tag{3}$$

will, statistically speaking, not imbalance the workload.

Proof. The Central Limit Theorem states that the sums of independent random variables tend to be normally distributed. Therefore, for a

set of n-iteration chores, the expected execution time is $n * \mu$ and the variance is $n * \sigma^2$. The normal distribution curve is defined as,

$$f(t)=\frac{1}{\sqrt{2\pi}\,\sigma_n} e^{-\frac{1}{2}\left(\frac{t-\mu_n}{\sigma_n}\right)^2}, \qquad \text{for} \quad -\infty < t < +\infty$$

where μ_n and σ_n are the expected value and standard deviation of the values of the random variable that has a normal distribution. In our case $\mu_n = n * \mu$ and $\sigma_n = \sqrt{n} * \sigma$. The probability for an n-iteration chore to finish before time t_0 is,

$$pr(t \leqslant t_0) = \int_{-\infty}^{t_0} f(t)\, dt$$

Let

$$c = \frac{t-\mu_n}{\sigma_n} \tag{4}$$

$pr(t < t_0)$ can then be calculated by

$$pr(c \leqslant c_0) = \int_{-\infty}^{c_0} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}c^2}\, dc$$

Let c_0 denotes the value of c in Eq. (4) when $t = (N/p)\,\mu$, we have

$$c_0 = \frac{\frac{N}{p}\mu - n * \mu}{\sqrt{n * \sigma^2}} \tag{5}$$

Kruskal and Weiss[19] have shown that if each processor receives a chore of equal size n the expected finishing time can be approximated as,

$$n\mu + \sqrt{2n\sigma^2 \ln(p)}$$

Thus, by definition of smallest critical chore n,

$$n\mu + \sqrt{2n * \sigma^2 \ln(p)} \leqslant \frac{N}{p}\mu$$

$$\sqrt{2n * \sigma^2 \ln(p)} \leqslant \frac{N}{p}\mu - n\mu$$

$$\sqrt{2\ln(p)} \leqslant \frac{\frac{N}{p}\mu - n*\mu}{\sqrt{n*\sigma^2}}$$

$$\sqrt{2\ln(p)} \leqslant \frac{\frac{N}{p}\mu - n*\mu}{\sqrt{n*\sigma^2}} = c_0$$

Therefore, the lower bound of c_0 can be

$$c_0 \geqslant \sqrt{2\ln(p)}$$

Clearly, c_0 can then be interpreted as the confidence factor. The larger the c_0 the smaller is the probability for the execution time of the chore to exceed $(N/p)\,\mu$. Solving n from Eq. (5), we have,

$$n = \frac{2\mu^2 \frac{N}{p} + c^2\sigma^2 - \sqrt{\left(2\mu^2 \frac{N}{p} + c^2\sigma^2\right)^2 - 4\mu^2\left(\frac{\mu N}{p}\right)^2}}{2\mu^2}$$

3.3. Comparison of SSS and Factoring

The particular implementation of SSS given in this paper is similar to that of Factoring in the methods used to calculate the chore sizes. Furthermore, in both schemes the chores in the same batch have the same size. However, there are several main differences between the two schemes. The first one is that Factoring uses the no-more-than-half rule, i.e., $\alpha \leqslant 0.5$ while in SSS, $0 < \alpha \leqslant 1$. The second difference is that SSS has two phases: a static scheduling phase and a dynamic scheduling phase. In SSS, a processor starts to execute a parallel loop with statically assigned iterations and smoothes out the uneven finishing times with a self-scheduling scheme. Third, the implementation given in this paper assumes that little is known about the iteration execution times. When more information is available, the amount of iterations assigned to each processor can also vary to best fit SSS's basic principle. Fourth, SSS's static scheduling phase increases the level of affinity between iterations and the processor because the first $N \times \alpha$ iterations' designating processors do not change on different runs of the loop. This property improves the performance of SSS by increasing the cache hit ratio and is proved to be beneficial[17] and useful in implementing self-scheduling on distributed-memory machines.[20,21]

The argument given by Factoring is that to achieve an overall optimal finishing time, for each batch scheduled there must be enough work left to smooth out the uneven-finishing times of that batch.[3] They argue that for some of the common distributions of chunk execution times including bell-shaped distributions, the expected finishing time of the first batch approaches $2\mu F_0$ (F_0 is the same as CS_0 used in this paper) as the number of processors p increases. Therefore, there must be pF_0 iterations left to smooth out its unevenness. Hence, to have a high probability of even finish times, no more than half the iterations should be scheduled in the first batch.

Clearly when $2\mu > E_{max}$ the expected finishing time of the first batch does not approach $2\mu F_0$ because the execution times of chores in the first batch must not be greater than $E_{max} \times F_0$. Let us further consider the following example.

Consider a *for* loop that has an *if-then-else* statement as its loop body. Let $N = 400$, $E_{max} = 4.0$ time units, $E_{min} = 1.0$ time unit, $prob(E_{max}) = 0.75$, $prob(E_{min}) = 0.25$, and $p = 5$. Therefore,

$$\mu = 0.75(4.0) + 0.25(1.0) = 3.25$$

$$\mu \frac{N}{p} = 3.25(400/5) = 260.0$$

$$\sigma^2 = 0.75(4.0 - 3.25)^2 + 0.25(1.0 - 3.25)^2 = 1.6875$$

$$\textit{safe chore size} = \frac{400 \times 3.25}{5} \Big/ 4.0 = 65$$

$$\textit{risk chore size} = \frac{400}{5} = 80$$

$$\alpha = \frac{0.75 + 0.25\frac{1}{4} + 1}{2} = 0.90625$$

$$CS_0 = \frac{400}{5} \times 0.90625 \approx 72$$

From the example we can see that assigning a processor a chore of 65 iterations (safe chore size) cannot imbalance the workload. This is because each processor needs to spend an average of 260 time units to finish the given parallel loop. Had there existed a processor spending less than 260 time units on the loop, there would have been another processor spending more than 260 time units on the loop; therefore, the schedule would be less balanced. However, the longest execution time of a 65 itera-

Table I. Chore Sizes for Different Scheduling Schemes

Scheme	Chores	$N = 400$ and $p = 5$																	
SSS	15	72	72	72	72	72	7	7	7	7	7	1	1	1	1	1			
GSS	25	80	64	51	41	33	26	21	17	13	11	8	7	5	4	4	3	2	...
TSS	16	40	38	36	34	32	30	28	26	24	22	20	18	16	14	12	10		
Factoring	40	40	40	40	40	40	20	20	20	20	20	10	10	10	10	5	5	5	...
CSS	$\lceil N/f \rceil$	f	f	f	f	f	...												

tion chore is 260 time units. Hence we conclude that assigning a processor fewer than 65 iterations (equivalent to set $\alpha \leqslant 0.8125$) only results in an increased scheduling overhead. Clearly, the no-more-than-half rule should not always be followed.

In SSS, the value of α is a main factor determining the total number of chores produced during the execution of a given parallel loop. The larger the value of α, the smaller the number of chores produced, resulting in a smaller overhead. When α becomes too large, chores with long execution times may be produced resulting in an unbalanced workload. The smaller the value of α, the fewer iterations in a chore, and the workload is more likely to be balanced because processors fetch small amounts of work at a time. However, scheduling overhead increases. Calculating CS_0 with an α smaller than μ/E_{max} increases scheduling overhead without further balancing the workload.

The total number of chores produced by Factoring is at least $p(1+\lg(N/p))$. The total number of chores produced by SSS is $p\lg(N/p)/\lg(1/(1-\alpha))$. For the example given earlier, SSS produces 10 chores while Factoring produces at least 37 chores. Note that a scheduling function needs to modify some global variables that have to be accessed exclusively. Therefore, the more frequent accessing of the shared variables such as the loop index, the longer the average time required to access them. We believe that for fine and medium grain parallel loops or for systems where accessing shared variables is an expensive operation, SSS will surpass Factoring. For large grain parallel loops, SSS will perform as good as Factoring. This is because loops that are suitable for Factoring can be scheduled using SSS with $\alpha = 0.5$.

Finding an appropriate value of α requires some information, such as the maximum and the minimum execution times and *prob*(*then*). We argue that it is possible to obtain approximation of these pieces of information. The execution times can be obtained through profiling utilities. The probabilities of a particular execution time can be obtained through sampling.[22] In addition, a program that solves a particular problem runs many times to solve different instances of the same problem. In cases like this, information regarding the parameters used in SSS can be collected from the earlier runs and used to benefit the later runs.

Table I shows the chore sizes for several scheduling schemes on the example given earlier. Since the safe chore size is 65, it is not necessary to assign a processor a chore less than 65 iterations to start with. Note that although SSS generates a total of 15 chunks, which is the smallest among all the schemes, only 10 chunks are assigned to a processor during runtime.

4. MODIFICATIONS TO THE SAFE SELF-SCHEDULING ALGORITHM

In this section we introduce some simple modifications to the SSS algorithm that further improve the performance and the flexibility of SSS.

4.1. Achieving a Higher Degree of Balanced Workload

As mentioned earlier, selecting a value for α is a trade-off between increasing the scheduling overhead and achieving a more balanced workload among the processors. SSS can be easily modified to achieve a even better balanced workload with roughly the same amount of overhead by applying a smaller value for α during the dynamic scheduling phase. For example, using the no-more-than-half rule during the dynamic scheduling phase of SSS may improve the performance, particularly for a parallel loop where iterations at the end of the loop are likely to have longer execution times than iterations at the beginning of the loop. Reverse Adjoint-Convolution application in Ref. 3 is an example that exhibits such a behavior.

4.2. Tolerating Faulty Processors

GSS is insensitive to faulty processors, i.e., even if one or more processors drop out after executing some chores, GSS would still balance the workload. This is not true with SSS. Consider the case when a processor drops out after executing some chores, the rest of the chores defined in the array *chore_list* no longer reflect the configuration of the current system. This may cause an unbalanced workload.

Simply using GSS in the dynamic scheduling phase makes SSS also insensitive to faulty processors.[23] The SSS-GSS scheduling can be described as given here.

1. Calculate the value for α.
2. Each processor is then assigned $(N/p)\,\alpha$ iterations statically.
3. Set the global variable *count* to the first unscheduled iteration's number.
4. The processors then
   (a) begin mutual exclusion;
   (b) copy the value of *count* to i;
   (c) $t \leftarrow max((N - count)/p, 1)$;
   (d) $count \leftarrow count + t$;
   (e) end mutual exclusion;
   (f) execute the chore defined by i and $i + t$ and repeat step 4 if $i > N$;

When the number of processors p is large, the value of p does not need to be modified if some processors become faulty and drop out of the system. This is because the old values of the chore sizes are only slightly smaller than the new ones calculated based on the new value of p. As we already know, a schedule using smaller chores, in general, results in at least as well balanced a workload as a schedule using larger chores. Note that step 4 can be precalculated and stored in an array. By doing so, the critical section can be replaced by a *fetch&add* command.

4.3. Differing Start Times

It is possible that not all of the processors begin to execute the loop at the same time. Waiting until all processors become free to start the loop will reduce the overall processor utilization. However, assigning chores in the first batch of $(N/p)\,\alpha$ iterations to a processor that starts at a much later time than the first processor that starts the execution of the loop may lead to an unbalanced workload. To prevent this from happening, we propose that SSS immediately enters the dynamic phase and determines the first batch of chore sizes as follows. Let t_s be the starting time of the processor that starts first, and t_i be the starting time of processor p_i. Then, a chore of the size

$$\max\left(\frac{N}{p}\alpha - \frac{(t_i - t_s)}{\mu}, 0\right) \tag{6}$$

is assigned to processor p_i. When $(N/p)\,\alpha < (t_i - t_s)/\mu$, the processor should then use the first available chore in the *chore_list*. The effect of this rule is that the later a processor starts, the less work it needs to complete. Following the first batch, the remaining batches are computed with the same approach previously described. This way, SSS continues to provide the benefits of a low overhead and a balanced workload. If the maximum delay time $t_x = \max_{j=1}^{p}(t_j - t_s)$ for a processor is known, then $((N/p)\,\alpha - t_x/\mu)\,p$ iterations can be scheduled statically by assigning to that processor with $(N/p)\,\alpha - t_x/\mu$ iterations.

5. EXPERIMENTAL RESULTS

Different scheduling schemes are evaluated on a 20-processor Sequent Symmetry, a shared-memory parallel computer. In this section, we discuss the results of three different tests. The first test compares the SSS scheme with other well-known scheduling schemes GSS,[5] TSS,[8] and Factoring[3] using a parallel loop with an *if-then-else* statement as its loop body. We

```
......
Doall i = 1 to SIZE do
   if (λ(i))
      then for (j =0; j < DIVERSITY*N1; j++) ct1 += 1;
      else for (j =0; j < N1; j++) ct2 += 1;
```

Fig. 2. A parallel loop containing branches.

implement GSS as GSS(1) and TSS as TSS($N/2p$, 1). In the other two experiments, we apply SSS scheme to real applications, namely matrix multiplication and Gauss-Jordan.

5.1. A Parallel Loop with an *If-then-else* Statement

The first test was conducted on the loop shown in Fig. 2. The loop has four parameters, i.e., SIZE, $\lambda()$, $N1$, and DIVERSITY. SIZE indicates the problem size. $\lambda()$ determines the frequency of executing the *then branch.* Parameter $N1$ specifies the granularity of an iteration. Parameter DIVERSITY specifies the diversity between the two branches.

We define the *cost* of executing a problem on a parallel system as the product of the parallel execution time and the number of processors used. The cost curves for different self-scheduling schemes executing the loop of

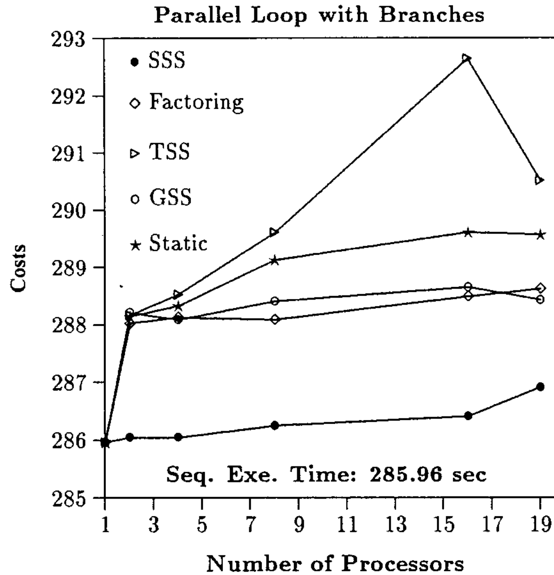

Fig. 3. Cost curves for different scheduling schemes.

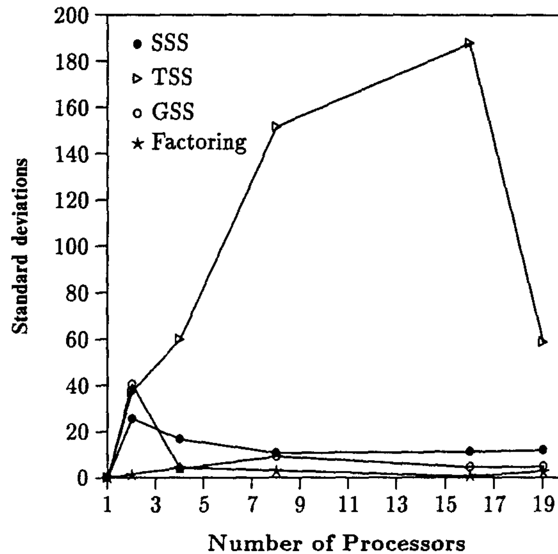

Fig. 4. Standard deviations in workload for different scheduling schemes.

Fig. 2 up to 19 processors are shown in Fig. 3. SSS outperforms the other scheduling schemes. The static scheduling scheme performs fairly well because the probability of executing the *then branch* is the same for all iterations.

Figure 4 shows the standard deviations for the processor workload on the corresponding runs of Fig. 3. The workload was calculated by counting DIVERSITY time units for each *then branch* and 1 time unit for each *else branch*. All the self-scheduling schemes except TSS provide a balanced workload. Factoring gives the most balanced workload followed by GSS and SSS. The well balanced workload of Factoring does not result in a good performance because it comes at the cost of an increased overhead in scheduling.

Figure 5 shows the speedup achieved by different scheduling schemes using different values of granularity of iterations, i.e., $N1$. Increasing the granularity of an iteration decreases the ratio between communication time and computation time. Therefore, all the scheduling schemes tested show improvement in performance. SSS scheme surpasses other schemes in all the tests with noticeable margins. The corresponding workload balance indicated by the standard deviations is given in Fig. 6. The workload for static scheduling is 28.3 and is not shown in the figure. The workload for TSS is also not shown in the figure since it is too large (170) and does not change much. Although both GSS and Factoring have a better balanced workload than SSS, they do not result in a better perfor-

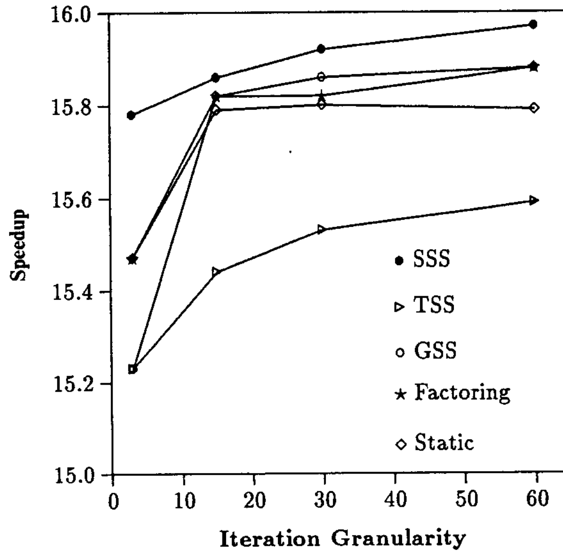

Fig. 5. Speedup of different schemes for different granularities.

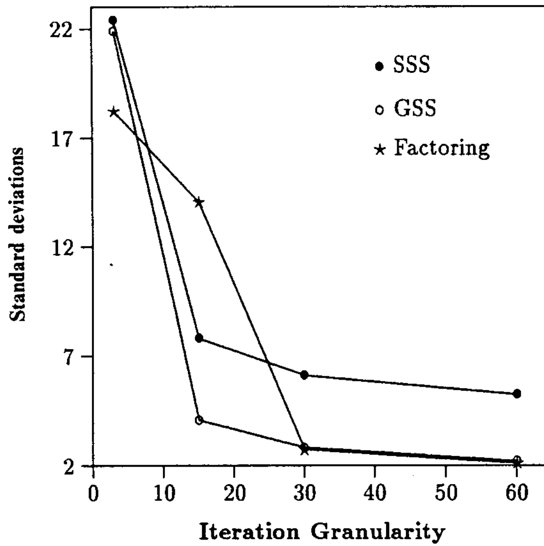

Fig. 6. Standard deviations in workload for different schemes using different granularities.

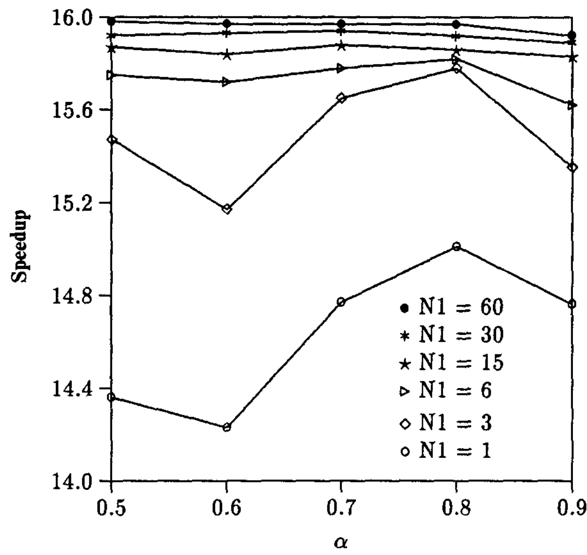

Fig. 7. Speedup of different granularities for different α values.

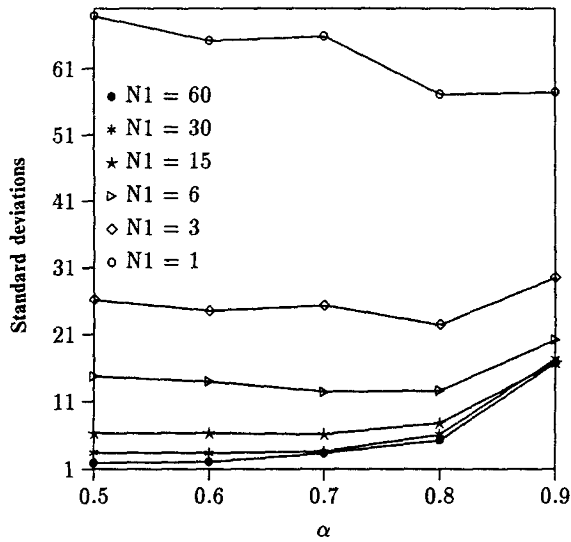

Fig. 8. Standard deviations in workload for different granularities.

mance than SSS because the balanced workload is achieved at the cost of a much higher scheduling overhead.

Figure 7 shows the speedup achieved by SSS for different values of α for different granularities. Again, the performance of SSS improves as the iteration granularity increases. When the granularity is small, the selection of α has a greater influence on the performance. An accurate value of α that reflects the characteristics of the loop produces better performance. When we increase iteration granularities, the value of α that yields the best performance decreases. This mainly is because the performance degradation caused by scheduling overhead becomes less significant. This suggests that a relatively smaller value of α should be used when scheduling parallel loops with a large granularity.

The workload balances of Fig. 7 are indicated by the standard deviations given in Fig. 8. The figure shows that the workload is more balanced when the iteration granularity increases. It also shows that as long as the value of α is not too large, smaller α values do not necessary result in a more balanced workload, except when $N1 = 1$. Also, since with a larger value of α, more iterations are scheduled statically (i.e., smaller scheduling overhead), a larger value of α should be used whenever possible to reduce scheduling overhead.

5.2. Matrix Multiplication

The code in Fig. 9 performs matrix multiplication when many elements of matrix **a** are zero. In our experiment, 43.75% of the elements in **a** are zero and all of them are located at lower-triangular of the matrix. The outer two loops are coalesced using the technique in Ref. 5. The execution time of an iteration is between 297 μs (microseconds) and 793 μs. Using the idea of Theorem 1, we find that $\alpha = 0.67$. Note that Eq. (1) is no longer applicable because the loop body is no longer a parallel loop with an *if-then-else* statement. Rather, the loop body is a sequential loop. The results of using SSS are shown in Fig. 10 with the comparative results given by SS (static scheduling), TSS, GSS, and Factoring. GSS assigns too much work at the beginning. This result in a very unbalanced workload and poor performance.

```
for i = 1 to N
   for j = 1 to N
      for k = 1 to N
         if a[i][k] <> 0 then
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Fig. 9. Matrix multiplication where many elements of array **a** are zero.

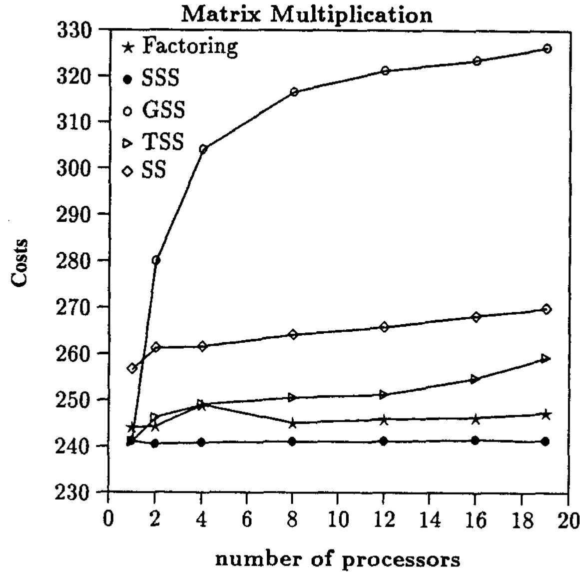

Fig. 10. Execution cost for the matrix multiplication loop given in Fig. 9.

5.3. Gauss-Jordan

Figure 11 shows the algorithm that performs Gauss-Jordan on an $N \times N$ array in Ref. 3. Note that the iteration granularity of Gauss-Jordan is small and is independent of the problem size. The amount of variance in the iteration length is also small. Problems of this kind are more suitable for static scheduling schemes than self-scheduling schemes. To outperform the static scheduling schemes on problems of this kind, a self-scheduling scheme must be able to achieve load balance with very small scheduling overhead. As shown in Fig. 12, SSS is the only dynamic scheduling scheme that outperforms the static scheduling scheme. The reason is that SSS

```
for i = 1 to N
   Doall l = 1 to N*(N - i) {
      j = l div (N - i);
      k = i + 1 + l mod (N - i);
      if (i ≠ j) then a[j][k] = a[j][k] - a[j][i]*a[i][k]/a[i][i];
      }
   for j = 0 to N - 1
      if (i ≠ j) then a[j][i] = 0;
```

Fig. 11. Gauss-Jordan.

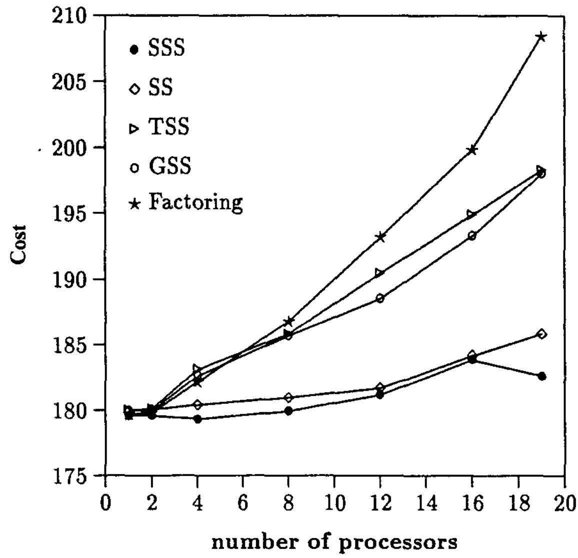

Fig. 12. Costs of running different schemes on Gauss-Jordan.

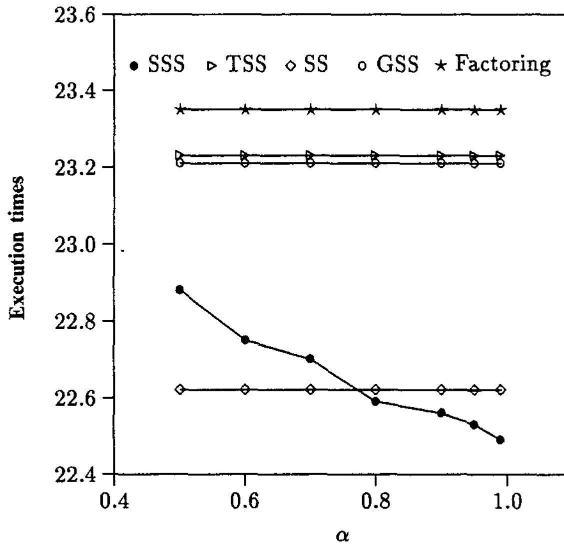

Fig. 13. The effect of using different values of α in Gauss-Jordan.

schedules a major portion (over 80%) of iterations to processors statically, the rest of the iterations being used to balance the workload dynamically.

Factoring does not perform well, particularly when the number of processors increases. This is because in Factoring the processors perform the largest number of fetches. The second reason is that since all except one processor obtain the same amount of work, when one processor finishes its work, all other processors (except one) also finish their work; therefore, the contention to access the critical section is likely to be much higher than that for other schemes. This problem becomes even more serious when the number of processors increases.

Figure 13 shows how the scheduling overhead affects the performance on eight processors. When α is small, the scheduling overhead is high. The result is that the static scheduling scheme performs well. As the value of α increases, SSS's performance improves. Finally, SSS outperforms the static scheduling scheme.

6. CONCLUSIONS

We have presented the Safe Self-Scheduling (SSS) scheme to schedule parallel loops with variable length iteration execution times not necessarily known at compile time. We have shown how to combine static and self-scheduling schemes in SSS and draw the advantages from both. SSS schedules statically a major portion of the loop iterations to processors to reduce the scheduling overhead while it uses self-scheduling to balance the workload at runtime.

Experimental results obtained from a shared-memory parallel computer indicate that while maintaining a well-balanced workload, the performance of SSS is superior to those provided by other well-known scheduling schemes.

SSS achieves a well-balanced workload with a low scheduling overhead. SSS's static scheduling phase improves the performance in two ways. One is that it increases the affinity between an iteration and the processor that executes the iteration thus increasing the cache hit ratio. The other is that it reduces the scheduling overhead by assigning a large portion of iterations to processors statically. The importance of having a static scheduling phase is further demonstrated when self-scheduling is implemented on distributed-memory machines.[20,21]

The preliminary work of adopting SSS to a distributed memory machine can be found in Ref. 21. We believe that scheduling parallel loops on distributed-memory parallel computers can benefit from the two phase approach of SSS, since the increased communication cost for a completely self-scheduling scheme will degrade the performance.

ACKNOWLEDGMENTS

We would like to thank the Computer Systems Architecture program at CCR Division at the National Science Foundation, and the College of Engineering and Prof. Walter G. Rudd of the Department of Computer Science at Oregon State University for supporting this research in part. Our thanks go to Prof. Michael J. Quinn for reading an earlier version of this paper and for his constructive suggestions. We also thank Yiu B. Lam for developing the model for statistical analysis for our loop scheduling scheme.

REFERENCES

1. Z. Fang, P. Tang, P. Yew, and C. Zhu, Dynamic Processor Self-Scheduling for General Parallel Nested Loops, *IEEE Trans. on Computers* **39**(7):919–929 (1990).
2. R. L. Graham, Bounds on Multiprocessor Scheduling Anomalies and Related Packing Algorithms, *Proc. of Spring Joint Computer Conf.* (1972).
3. S. F. Hummel, E. Schonberg, and E. L. Flynn, Factoring: A Method for Scheduling Parallel Loops, *Comm. of the ACM* **35**(8):90–101 (1992).
4. K. Kimura and N. Ichuyoshi, Probabilistic Analysis of the Optimal Efficiency of the Multi Level Dynamic Load Balancing Scheme, *Proc. of the Sixth Distributed Memory Comput. Conf.*, pp. 145–152 (1991).
5. C. Polychronopoulos and D. J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. on Computers* **36**(12):1425–1439 (1987).
6. V. A. Saletore, A Distributed and Adaptive Dyanmic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks, *Proc. of the Fifth Distributed Memory Comput. Conf.*, pp. 994–999 (1990).
7. P. Tang, P. Yew, and C. Zhu, Compiler Techniques for Data Synchronization in Nested Parallel Loops, *Proc. of Int'l. Supercomputing Conf.*, pp. 177–186 (1990).
8. T. H. Tzen and L. M. Ni, Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers, *IEEE Trans. on Parallel and Distrib. Syst.* **4**(1):87–98 (1993).
9. J. Xu and K. Hwang, Heuristic Methods for Dynamic Load Balancing in a Message-Passing Supercomputer, *Proc. of Supercomputing*, pp. 888–897 (1990).
10. E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley and Sons, New York (1976).
11. T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, New York (1992).
12. M. Wolfe, Loop Rotation, Language and Compilers for Parallel Computing, *Research Monographs in Parallel and Distributed Computing*, MIT Press (1990).
13. W. Stallings, *Computer Organization and Architecture*, Macmillan, New York (1990).
14. E. L. Lust and R. A. Overbeek, Implementation of monitors with macros: A programming aid for the HEP and other parallel processors, Argonne National Laboratory, ANL-83-97, Argonne, IL (1983).
15. M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York (1987).
16. L. M. Ni and C. E. Wu, Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems, *IEEE Trans. on Software Engineering* **15**(3):327–334 (1989).
17. E. P. Markatos and T. J. LeBlance, Using Processor Affinity in Loop Scheduling on Shared-Memory, The University of Rochester, Computer Science Department, TR 410 (1992).

18. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU ultracomputer—Designing an MIMD shared-memory parallel computer, *IEEE Trans. on Computers* **C-32**(2):175–189 (1983).
19. C. P. Kruskal and A. Weiss, Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. Software Engineering* **SE-11**(10):1001–1016 (1985).
20. J. Liu and V. A. Saletore, Self-Scheduling on Distributed-Memory Machines, *Proc. of Supercomputing*, Portland, Oregon, pp. 814–823 (1993).
21. V. A. Saletore, J. Liu, and B. Y. Lam, Scheduling Non-uniform Parallel Loops on Distributed Memory Machines, *Proc. of Hawaii Int'l. Conf. on Syst. Sci.* **2**:516–525 (1993).
22. K. Kant, *Introduction to Computer System Performance Evaluation*, McGraw-Hill, New York (1992).
23. J. Liu, J. C. Marsaglia, B. Broeg, and V. A. Saletore, Scheduling Parallel Loops Under Faulty Processors, *Proc. of Sixth Int'l. Conf. on Parallel and Distrib. Comput. Syst.*, pp. 387–392 (1993).
24. J. Liu, V. A. Saletore, and T. G. Lewis, Scheduling Parallel Loops with Variable Length Iteration Execution Times on Parallel Computers, *Proc. of ISMM 5th Int'l. Confer. on Parallel and Distrib. Comput. and Syst.*, pp. 83–89 (1992).