AN ABSTRACT OF THE THESIS OF

Jie Liu for the degree of Doctor of Philosophy in Computer Science presented on September 22, 1993.

Title: Scheduling Non-uniform Parallel Loops on MIMD Computers

Redacted for Privacy

Abstract approved: ______________________

Dr. Vikram A. Saletore.

Parallel loops are one of the main sources of parallelism in scientific applications, and many parallel loops do not have a uniform iteration execution time. To achieve good performance for such applications on a parallel computer, iterations of a parallel loop have to be assigned to processors in such a way that each processor has roughly the same amount of work in terms of execution time. A parallel computer with a large number of processors tends to have distributed-memory. To run a parallel loop on a distributed-memory machine, data distribution also needs to be considered. This research investigates the scheduling of non-uniform parallel loops on both shared-memory and distributed-memory parallel computers.

We present Safe Self-Scheduling (SSS), a new scheduling scheme that combines the advantages of both static and dynamic scheduling schemes. SSS has two phases: a static scheduling phase and a dynamic self-scheduling phase that together reduce the scheduling overhead while achieving a well balanced workload. The techniques introduced in SSS can be used by other self-scheduling schemes. The static scheduling phase further improves the performance by maintaining a high cache hit ratio resulting from increased affinity of iterations to processors. SSS is also very well suited for distributed-memory machines.

We introduce methods to duplicate data on a number of processors. The methods eliminate data movement during computation and increase the scalability of problem size. We discuss a systematic approach to implement a given self-scheduling scheme on a distributed-memory. We also show a multilevel scheduling scheme to self-schedule parallel loops on a distributed-memory machine with a large number of processors to eliminate the bottleneck resulting from a central scheduler.

We proposed a method using abstractions to automate both self-scheduling methods and data distribution methods in parallel programming environments. The abstractions are tested using CHARM, a real parallel programming environment. Methods are also developed to tolerate processor faults caused by both physical failure and reassignment of processors by the operating system during the execution of a parallel loop.

We tested the techniques discussed using simulations and real applications. Good results have been obtained on both shared-memory and distributed-memory parallel computers.

Scheduling Non-uniform Parallel Loops on MIMD Computers

by

Jie Liu

A Thesis
submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Completed September 22, 1993

Commencement June 1994

Approved:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented September 22, 1993

Typed by Jie Liu for Jie Liu

ACKNOWLEDGMENTS

I would like to express my deep appreciation to Dr. Vikram Saletore, my thesis advisor; it is only with his guidance, insight, understanding, and patience that the early part of my research developed into this thesis.

I thank Dr. Mike Quinn whose book "Designing Efficient Algorithms For Parallel Computers" is like a fountainhead that tirelessly feeds me with knowledge. I thank Dr. Bella Bose for his friendship and for those interesting discussions. I thank Dr. Lawrence Crowl for his suggestions concerning this research. I thank Dr. Wilson Schmisseur for serving on my committee and for his encouragement.

I have benefited in many ways from the discussions with fellow students Bob Broeg, Phyl Crandall, Mark Clement, Jason Moore, Brad Seavers, Gowri Ramanathan, Seungjin Park, Manojith Padala, Lam Ben Yiu, and Joseph Jacob. I thank you all and wish you all the best.

My special thanks are due to my wife, Lan Chen, and my daughter Sandy, for their support, understanding, patience, sacrifices, and love.

I thank my parents, Qing-huai Liu and Bao-ying Zhang, who have always loved me, supported me, and encouraged me with the value of knowledge and a good education. This thesis is part of my filial piety to them, and I am sure that it will make them even happier. I also want to thank my younger brother Jin Liu for always standing behind me.

I am grateful to the Department of Computer Science, Western Oregon State College for employing me. I also thank Inner-Mongolia Agriculture and Animal Husbandry College for sending me to the USA for my education.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Over the past fifty years we have witnessed dramatic increases in computing speed. However fast the fastest computer of today may be, there are always applications demanding computers that are many orders of magnitude faster. For many years, computer engineers have been admirably successful in increasing the speed of computers by employing better hardware technologies. Unfortunately, a limiting factor—the speed of light in a vacuum—is putting an end to this trend. It becomes inevitable that a substantial increase in computer speed can only come about by increasing the number of operations taking place concurrently. This fact has been well noted by computer researchers and computer manufacturers. Consequently, a large number of parallel computers have been built in research laboratories, and many parallel computers are available commercially on the market.

The availability of parallel computers has led to an expectation that most computation-intensive scientific applications will be routinely sped up using parallel processing. In these applications, loops are the most time-consuming parts and are the richest source of parallelism [68]. In many scientific applications that run on parallel computers, the loop is (or can be converted into) a *parallel loop*, i.e., a loop in which each iteration is independent of all others. A parallel loop, also called a `DoAll` loop, has no cycles in its dependence graph [54]. 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.

When the iteration execution times of a parallel loop do not vary significantly, the loop is a *uniform* parallel loop; otherwise, the loop is a *non-uniform* parallel loop. In order to execute a parallel loop concurrently, iterations of the loop have to be assigned to processors, either at compile time or at run time. Clearly, different assignments of iterations to processors yield different execution times. Since one of the main reasons to employ parallel computers is to reduce the total execution time, assignments of iterations to processors rendering short completion times are always desirable. A *schedule* of a parallel loop is an assignment listing, for each iteration, the processor executing the iteration. A *static scheduling* scheme assigns iterations to processors at compile time; in contrast, a *dynamic scheduling* scheme assigns iterations to processors at run time.

To schedule a uniform parallel loop for maximum efficiency, an equal number of iterations are assigned to each processor (assuming that processors start executing the loop at the same time). In scheduling a non-uniform parallel loop, assigning an equal number of iterations to each processor does not always result in each processor having an equal amount of workload measured in some time units of execution time. Since a parallel loop is finished only after all iterations have finished, a balanced workload is a key factor to good performance for a non-uniform parallel loop. In the presence of variable length iteration execution times, a dynamic scheduling scheme is in principle superior in balancing the workload [79].

1.1 Problems Studied

The scheduling of parallel loops is a special case of the general scheduling problem, which has been studied extensively by many researchers in a theoretical context [12, 29]. Scheduling in general is NP-complete [12].

Let $I = \{t_1, t_2, \cdots, t_N\}$ be the iteration space formed by the N iterations of a parallel loop. Let $e(t_i)$ be the execution time for iteration t_i. Further, let M be a parallel machine that has P processors denoted as $\{p_1, p_2, \cdots, p_P\}$. N, P, and $e(t_i)$ are positive integers and may be ***unknown*** at compile time. Let $D = \cup d_i$, where d_i is a partition of I into P *disjoint* subsets $I_1^i, I_2^i, \cdots, I_P^i$ and D is the collection of all possible partitions. In addition, the size of D is P^N. The loop scheduling problem is to find an optimal partition d_o such that[1] the execution time of the loop is minimized. That is,

$$\min_{d_i \in D} \left\{ \max_{1 \le k \le P} \{ \sum_{t_j \in I_k^i} e(t_j) \} \right\} \ge \max_{1 \le k \le P} \{ \sum_{t_j \in I_k^o} e(t_j) \} \tag{1.1}$$

Except for some trivial cases, such as when $P = 1$, $e(t_i)$ are constants, $P = N$, or other limited number of special cases discussed later, the yes-no version of the scheduling problem is NP-complete [12]. To complete scheduling in a reasonable amount of time, heuristics are used to approximate such problems in polynomial time.

In practice, $e(t_i)$ is often not a constant and may not be known at compile time; therefore dynamic scheduling techniques are applied to achieve a high processor utilization. In this case the loop scheduling problem discussed in this dissertation is different from the traditional one given above. Let I_j^i be the collection of iterations assigned to processor p_j according to partition d_i, $E(I_j^i)$ be the total iteration execution time of the iteration in I_j^i, and O_j be the scheduling overhead

[1]Note that the scheduling cost is neglected here.

for processor p_j, then the total execution time of the parallel loop under partition D_i is

$$E(I) = \max_{j=1}^{P}\{E(I_j^i) + O_j\}. \tag{1.2}$$

From Eq.(1.2) we can see that either balancing $E(I_j^i)$ or reducing O_j or both together reduce the total execution time $E(I)$. Balancing $E(I_j^i)$ improves performance because the total amount of work W of a parallel loop is independent of both the assignment of iterations to processors and the number of processors employed in executing the loop. When more than one processor is used to execute the loop, the amount of work done by all the processors cannot be less than W. Therefore, if a processor is assigned a smaller than the average amount of work, there must be some other processors that are assigned a larger than the average amount of work. In addition, since the execution of the loop is not completed until all the iterations are executed, assigning each processor an equal amount of work so that all processors finish at the same time, improves performance by achieving a high processor utilization. The objectives of this study are to find practical methods of partitioning the iteration space to produce a balanced workload with a low scheduling overhead.

Self-scheduling is the most common approach to dynamic scheduling of non-uniform parallel loops. In this approach, a ready task queue is created. Whenever a processor becomes idle, it removes the first task from the queue and executes it, i.e., processors "self-schedule" themselves as the program executes [83]. The research in this thesis investigates combining static scheduling and self-scheduling to schedule a non-uniform parallel loop on both shared-memory and distributed-memory parallel computers.

1.1.1 Self-Scheduling of Parallel Loops on Shared-Memory Parallel Computers

A shared-memory parallel computer is an ideal environment to implement self-scheduling schemes. Recall that a ready task queue needs to be maintained in self-scheduling, and this ready task queue needs to be shared in the sense that all processors have access to the queue. A parallel loop with N iterations can be considered as a ready task queue with N tasks and the loop index points to the head of the queue. When a processor removes an iteration, it only needs to modify the loop index through exclusive access, however

The main issue in self-scheduling a parallel loop on a shared-memory machine is balancing the trade-off between assigning each processor roughly the same amount of work—achieving a balanced workload—and incurring a low scheduling overhead. On one hand, an unbalanced workload lengthens the execution time of the parallel loop while, on the other hand, achieving a balanced workload by incurring a high scheduling overhead may diminish the benefit of having a balanced workload.

When a parallel loop is enclosed in a serial loop, assigning an iteration of the parallel loop to the same processor in every iteration of the serial loop also improves the performance because this helps to maintain a high cache hit ratio.

1.1.2 Self-Scheduling of Parallel Loops on Distributed-Memory Parallel Computers

Self-scheduling on distributed-memory machines faces many challenges. The first one is that since there is no shared-memory, the shared ready task queue has to be either distributed or stored on some processors. If the queue is distributed on all the processors, maintaining the consistency of the queue becomes too costly. An alternative way is to designate a processor as the scheduling processor that maintains the ready task queue. When a processor becomes idle, it sends a request message to the scheduling processor requesting additional work.

If all the iterations are thus self-scheduled, a balanced workload may be achieved but at the cost of a high communication overhead. In addition, the scheduling processor may become a bottleneck as the number of processors increases.

Another issue is the need of a processor in a distributed-memory machine to store the data needed by an iteration in order to execute the iteration. This is not a trivial problem because data is usually distributed to processors before the execution of a loop begins; in contrast, the iterations are assigned to processors during the execution of the loop.

1.1.3 Compiler Level Support of Self-Scheduling of Parallel Loops

Few parallel languages or environment, support self-scheduling of parallel loops. It is left totally to the user to implement the scheduling schemes of his or her choice. Implementing a self-scheduling scheme is not an easy task. Bugs are often introduced into the program during implementation. In addition, the code for scheduling techniques is often interspersed with the code for the underlying algorithm. This make the program more complicated, more difficult to port from one machine to another, and more difficult to debug.

On a distributed-memory machine, the programmer also has to implement some data distribution policies to ensure that an iteration is assigned to a processor storing the data needed by the iteration. It is also possible that several parallel loops in the same program need the same data. If these loops are scheduled using different self-scheduling schemes, the data distribution suitable for one loop may not be suitable for other loops. In this case data may need to be redistributed at run time for efficiency reason.

1.1.4 Self-Scheduling of Parallel Loops under Faulty Processors

Although many schemes have been proposed in the past, they all assume that the number of processors remains unchanged during the execution of the parallel loop. However reliable today's computer may be, the more processors a system has, the more likely one will become faulty. This can also happen during the execution of a parallel loop. To ensure both the correctness and the efficiency of the execution of a parallel loop, measures must be taken to tolerate faulty processors.

Two different cases are studied. The first is a hardware failure; the second is when the operating system reassigns processors from one task to another before the first task is completed.

1.2 Contributions of This Research

In this dissertation we have studied how to efficiently execute a scientific application. This problem is, to a certain extent, the essence of parallel processing.

We demonstrated a technique of combining a static scheduling scheme with a dynamic scheduling scheme. This combination of schemes has the following advantages:

1. reducing the scheduling overhead,
2. achieving a balance workload,
3. simplifying data distribution,
4. making it easier to employ other well known scheduling schemes to utilize their advantages, and
5. increasing the affinity of iterations to processors which further improves performance by maintaining a high cache hit ratio.

The combination also makes self-scheduling a parallel loop on a distributed-memory machine more feasible and dramatically increases the size of solvable problems. Further contributions of our research follow below.

We developed a method for duplicating data on a number of processors. This method eliminates any data movement during the computation of a parallel loop and increases the problem size scalability.

We devised a systematic approach for implementing a given self-scheduling scheme on a distributed-memory computer.

We also studied multilevel scheduling. This further enhanced the scalability of self-scheduling schemes on distributed-memory machines.

We proposed a method using abstractions to automate both self-scheduling of parallel loops and data distribution in parallel programming environments. The method was tested using CHARM, a architecture independent parallel programming environment [19].

Methods were developed to tolerate the loss of a processor because of physical failure or reassignment by the operating system during the execution of a parallel loop.

All the methods proposed in this dissertation have been implemented on real parallel computers using both simulation and real applications. Good results have been obtained. For example, we improve the performance by 79% over static scheduling for the false color image problem on an NCUBE/7, a distributed-memory machine.

1.3 Organization For the Rest of This Dissertation

- Some of the well known self-scheduling schemes developed by other researchers are presented in Chapter 2. The assumptions on which this research is based is given in Chapter 2.

- In Chapter 3 we present *safe self-scheduling* (SSS), a new scheme that self-schedules parallel loops on shared-memory machines.
- Chapter 4 discusses the implementation of SSS on a distributed-memory machine.
- Chapter 5 introduces a general method for implementing a self-scheduling scheme on a distributed-memory machine. Data distribution methods are also the focus of this chapter.
- We present an approach for automating data distribution methods and parallel loop self-scheduling schemes in CHARM in Chapter 6.
- In Chapter 7 we discuss methods for enhancing SSS to tolerate faulty processors.
- In Chapter 8 we summarize our work and discuss related issues for future studies.

Chapter 2

RELATED WORK

A scheduling problem emerges whenever there is a choice as to the order of performing a number of tasks. The goal of scheduling is to determine an assignment of tasks to processing elements and an order which achieves some optimal performance measures. Scheduling problems can be found in a manufacturing plant where a number of operations transform raw materials into a final product, in a bank where customers wait to be served by tellers, in a computer lab where students wait for computers, and in a parallel program where tasks need to be assigned to processors.

2.1 The General Scheduling Problem

In our research a schedule is a mapping of tasks to processors. A general task system can be defined as a system (I, $\prec$, $[e(t_i)]$, M) as follows:

1. I $= \{t_1, t_2, \cdots, t_N\}$ is the task space.

2. $\prec$ represents the irreflexive partial orders defined on I.

3. $e(t_i)$ is the execution time for task t_i

4. M $= \{p_1, p_2, \cdots, p_p\}$ is the processor space.

Often the tasks and the partial orders among the tasks can be represented in a directed graph, the *task graph*, in which each task is represented as a node and the partial order between two tasks t_i and t_j is represented by an edge from t_i to t_j. Some researchers also consider the inter-processor communication cost [17]. In our study, this cost is included in the scheduling overhead.

In the problem of assigning tasks to processors in a parallel computer, performance is measured by the amount of time needed from the start of the first task to the completion of the last task. This type of problem, usually referred to as the minimum execution time multiprocessor scheduling problem, has been studied by many researchers in a theoretical context [11, 12, 29, 90]. Scheduling in general is NP-complete [26]. When the communication overhead is not considered and the task execution times are identical, there are only three cases where an optimal schedule can be obtained in polynomial time.

The first case is given in [34]. It is a linear algorithm (in number of tasks) that give an optimal schedule for a tree shaped task graph. The second case is when the task graph is in an interval order. The complexity of the algorithm is linear in the sum of the number of nodes and edges of the task graph [76]. In the third case, a quadratic algorithm (in number of tasks) exists producing an optimal schedule for an arbitrary task graph on two processors [12]. Each algorithm above becomes NP-complete if any of the restricting conditions assumed is removed. In practice, however, the above algorithms are not very useful because they assume the task execution times are identical.

The *critical path* algorithm and its many variations [12] are the central result from classical scheduling. Again they are seldom useful in practice because they assume that only serial tasks exist in a program and that the exact execution time for each task is known. In general, neither of the two assumptions is valid when used to schedule a parallel loop where each iteration can be considered as a task.

Even if we fully unroll a parallel loop and consider one iteration as a serial task, we still cannot satisfy the first assumption for the following two reasons. The first is that the number of iterations may not be known at compile time, and if the total number of tasks is not known, a schedule cannot be constructed. The other reason is that, even if the total number of tasks is known, it may be a very large number, say several thousands of tasks. The corresponding task graph is so large that scheduling becomes prohibitively expensive even using a linear heuristic algorithm. Also it is likely that the resulting schedule is not optimal.

Scheduling schemes can be classified as either static or dynamic. A static scheduling scheme assigns tasks to processors at compile time. The major advantage of static scheduling schemes is that they impose no run time overhead. The main drawback of static scheduling schemes is its inability to respond to an unbalanced workload among the processors. This imbalance may be caused by branch instructions, memory conflicts, cache misses, and other random delays [5].

List scheduling [1, 11, 12] is an example of static scheduling. This type of scheduling assumes that an ordered list (the priority list) of tasks is constructed beforehand. Thus, tasks are assigned to processors by repeatedly scanning the list to find the first unexecuted task that is ready for execution. List scheduling is a polynomial time algorithm that produces a suboptimal solution in the case of unlimited resources.

Dynamic scheduling schemes are designed to alleviate the problem of imbalance in workload among processors. Dynamic scheduling schemes do not determine the assignment of tasks to processors until the execution is underway. This allows dynamic scheduling schemes to balance the workload more equitably, resulting in more efficient use of processors. However, this adaptability comes at the cost of an additional run time scheduling overhead.

Graham [29] showed that in many cases of random task graphs, optimal schedules can be achieved by deliberately keeping one or more processors idle to better utilize them at a later point in time. Detecting such anomalies requires the information the entire task graph a priori. In addition, when the length of a task is known only at run time, deliberately keeping a processor idle to achieve a better utilization at a later time is not feasible because a task graph cannot be constructed.

In a dynamic scheduling scheme, a new task is assigned to a processor as soon as the processor becomes idle. Clearly, dynamic scheduling may not always produce an optimal schedule. However, it has been shown that assigning a task to a processor whenever a processor becomes idle results in an execution time which, theoretically, is never more than twice the optimal [12]. In real cases, the execution times are very close to the optimal, assuming no overhead [79]. Thus, the overhead factor is the critical optimization parameter of dynamic scheduling.

Robinson [87] gives an estimation of the expected execution time of a parallel program with a "simple" task graph when the execution time is represented by a random variable. It offers little help because it assumes no scheduling overhead and assumes that the number of processors is not limited.

Scheduling a parallel loop has characteristics different from the traditional job shopping problem. First, there is no partial order among iterations of a parallel loop. Second, the execution time of an iteration may become known only after the iteration is executed. Figure 2.1 is an example of such a parallel loop. For loops of this kind, assign an equal number of iterations to each processor may result in an unbalanced workload. Many approaches have been proposed for assigning iterations of a parallel loop to processors of a parallel machine. In next two sections we illustrate some of the well-known approaches by showing how they schedule a parallel loop with N iterations on a parallel machine with P processors.

```
x = random();
y = random();
for i = 1 to $\frac{N}{P}$ do
    if ($\sqrt{x^2 + y^2} \leq 1.0$)
      then
          {
          $picount = picount + 1;$
                                      $E_{max}$
          $count = count + 1;$
          }
      else
          {
          $count = count + 1;$        $E_{min}$
          }
```

Figure 2.1. Calculating π using the Monte Carlo method

2.2 Static Scheduling Schemes

A static scheduling scheme assumes that two processors with the same number of iterations have roughly the same execution time. If we define that a *chunk* is a set of *consecutive* iterations, then two chunks of the same size may require the same amount of execution time. However, for non-uniform parallel loops, the probability that two chunks with an equal number of iterations have the same execution time is small. Static scheduling schemes are still in use because they are simple and sometimes result in lower execution times than dynamic scheduling schemes.

A static scheduling scheme is better suited for a uniform parallel loop or when the iteration execution times of a parallel loop are known.

Static Chunk (SC)

Static chunk assigns each processor $\lceil N/P \rceil$ consecutive iterations, except the last processor which is assigned whatever iterations are left, at compile time. Except for the case when the iteration execution time is the roughly the same, such an assignment may cause an unbalanced workload. The performance obtained using this approach on a non-uniform parallel loop is unpredictable. That is, it is possible to obtain good performance when the number of processors is a certain number; however, simply increasing the number of processors by one may cause the performance to degrade significantly.

Round Robin (RR)

A modification of SC is to assign iterations to processors in a round-robin fashion, rather than assigning a processor with a consecutive block of iterations. That is, iteration i is assigned to processor $i \bmod P$. This approach is likely to produce a more balanced workload than SC. One problem with this approach is that the cache hit ratio may be low. A second problem is that the workload produced by RR is not guaranteed to be better balanced than that of SC. This is because each processor executes about N/P iterations, and if the set of iterations assigned to a processor contains many relatively long iterations, it may take a longer than that average execution time to complete, resulting in an unbalanced workload. In addition, as long as the iterations assigned to *one* processor are relatively longer than those of the other processors, the schedule is not balanced. The larger the number of processors, the higher the chance such a processor exists.

2.3 Self-Scheduling Schemes

The basic principle of a self-scheduling scheme is that when a processor becomes idle, it fetches one or more iterations and modifies shared variables such as the loop index, however, exclusively. In this way, a processor obtains more work only if it becomes idle; therefore, it does not delay the execution of the whole loop by having too much work. Also, as long as there are iterations left, an idle processor always works on them; therefore, these iterations are processed at the earliest possible time. The result is a well balanced workload.

Pure Self-scheduling

A straight forward implementation of self-scheduling of parallel loops is the *pure self-scheduling* (PSS) approach. In this approach, a processor fetches one iteration at a time during run time by incrementing the loop index in a critical section when it becomes idle. Completing the fetched iteration, the processor becomes idle again and fetches another iteration. This process repeats until all the iterations have been executed.

PSS always achieves a well balanced workload. However, this well balanced workload does not always yield good performance. This is because the amount of scheduling overhead due to the assignment of iterations to processors is proportional to the number of iterations. This amount usually is significant compared to the execution of a iteration. In addition, when the granularity of each iteration is small and the execution times of different iterations do not vary significantly, the high frequency of mutually exclusive access to shared variables, such as the loop index, may become a bottleneck, and this may seriously degrade performance. Overall, PSS may be appropriate for scheduling loops having relatively few iterations but with very long variable length execution times when compared to the scheduling overhead.

Chunk Self-scheduling

Chunk self-scheduling (CSS) is designed to overcome the problem of high scheduling overhead in PSS by allocating a fixed number, k, of consecutive iterations to each idle processor [4]. When $k = 1$, CSS becomes PSS. When $k = N/P$, CSS can be carried out in the same way as SC.

By having processors fetch more iterations at a time, PSS reduces the scheduling overhead, but it compromises load balancing. This is because the task allocation is performed with a larger granularity than that of SC.

The main drawback of CSS is the dependence on both the chunk size and the characteristics of each loop, either of which may not be known even at run time. Worse yet, even for the same loop, the execution time does not monotonically increase or decrease with chunk size [81]. Polychronopoulos and Kuck [79] proved that there cannot be an optimal value of k even for the simplest cases. Polychronopoulos further points out that CSS may even result in a slowdown, i.e., it takes a longer time to execute a parallel loop using more than one processor than to execute the loop sequentially, when the chunk size k assumes a value smaller than some threshold [81].

Guided Self-scheduling

Polychronopoulos and Kuck [79] presented the *guided self-scheduling* (GSS) scheme. In GSS, the number of iterations assigned to an idle processor is calculated dynamically. An idle processor fetches 1/P of the unscheduled iterations. When processors start executing the loop at different times, GSS produces a well balanced schedule with a low overhead for a uniform parallel loop.

A significant contribution of GSS is that it assigns reduced sized chunks to processors. By doing so, GSS is more likely to achieve a better balanced workload than CSS, with a lower scheduling overhead than that of either PSS or CSS.

When GSS is applied to a non-uniform parallel loop with N iterations, assigning close to N/P iterations to the first several fetching processors may cause a load imbalance if the iterations assigned to one processor need more than the average time to finish. In addition, near the end of the scheduling process, GSS produces many chunks of one or two iterations. This results in a large scheduling overhead since GSS acts more like PSS. Thus, Polychronopoulos proposed GSS(t), a modification which avoids the problem by allocating no less than t iterations at a time to an idle processor [79]. This approach compromises a lower scheduling overhead with a less balanced schedule. In addition, an optimal value of t that results in a well balanced schedule with minimum overhead is both application and hardware dependent [79]. Consequently, a number of schemes have been introduced to overcome these problems.

Trapezoid Self-scheduling

Tzen and Ni [109] proposed the *trapezoid self-scheduling* (TSS) algorithm to improve GSS. In their approach, 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 the programmer or the compiler. This method linearly decreases the number of iterations assigned to each processor at run time by some decreasing step δ. However, the selection of N_s and N_f suffers from the same problem as the selection of t in GSS(t) and k in CSS. Tzen and Ni proposed TSS($N/2P$, 1) as a general selection of N_s and N_f.

It is stated in [109], for a given N_s and N_f, that the total number of chunks T is

$$T = \left\lceil \frac{2N}{N_s + N_f} \right\rceil .$$

The decreasing step δ can be obtained by following formula

$$\delta = \frac{N_s - N_f}{T - 1}. \tag{2.3}$$

The basic idea of TSS is to extract the advantages of both CSS and GSS by linearly decreasing the number of iterations assigned to processors. 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 $P \times \delta$. For example, assume that TSS($N/2P$, 1) is used to schedule a parallel loop with $N = 1,000,000$ iterations on a system with $P = 256$ processors. Then

$$\begin{aligned} N_f &= \frac{N}{2P} = 1953.125 \approx 1953; \\ T &= \left\lceil \frac{2N}{N_s + N_f} \right\rceil = \left\lceil \frac{2 \times 1,000,000}{1953 + 1} \right\rceil = 1024; \\ \delta &= \frac{N_s - N_f}{T - 1} = \frac{1953 - 1}{1024 - 1} = 1.908 \approx 2. \end{aligned}$$

In this example the difference is 512 iterations. It is true that the two processors may not fetch at the same time. However, as long as the fetching times of the two processors are not too far apart, the difference in finishing times between the two processors could be significantly large. This weakness limits the usage of this scheme for problems needing a large number of processors.

Another problem of TSS shows up when the decreasing step δ calculated by Eq.(2.3) is a real number and has to be converted to an integer. Ignoring the fraction part results in the last several chunks being too large. Rounding up the real number to the next integer causes many chunks of size 1 and a large T, the total number of chunks.

For instance, in the above example, the value of δ calculated by Eq.(2.3) is 1.908. Using $\delta = 2$ results in 45,472 chunks of 1 iteration. In this case TSS achieves a balanced workload, however, with an enormous amount of scheduling overhead. When $\delta = 1$, the last chunk has 1145 iterations. Clearly, this may not balance the workload.

Factoring

Hummel et al. [21, 43] introduced *Factoring.* In Factoring, fixed sized chunks of iterations are allocated to processors in batches (P consecutive chunks form a batch), and the sizes of chunks in the same batch are the same. This size is determined using the *no-more-than-half* rule during implementation. This rule states that the chunk size of a batch is half of the chunk size of the previous batch. 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 over the uneven finishing times of the batch. The rational for this reasoning is that if a bell shaped curve for a large number of random variables (iteration execution times) with mean μ is assumed, the expected finishing time of the first P chunks of size F_0 approaches $2\mu F_0$ when P is large enough.

Factoring is based on the following analytical results in calculating a chunk size.

$$\begin{aligned} R_0 &= N \\ R_{j+1} &= R_j - PF_j \end{aligned}$$

$$\begin{aligned}
F_j &= \frac{R_j}{x_j P} \\
b_j &= \frac{P}{2\sqrt{R_j}}\frac{\sigma}{\mu} \\
x_0 &= 1 + b_0^2 + b_0\sqrt{b_0^2 + 2} \\
x_j &= 2 + b_j^2 + b_j\sqrt{b_j^2 + 4}, \; for \; j > 0
\end{aligned}$$

where σ and μ are the variance and the mean of the iteration execution times, respectively. The subscript indicates the batch.

According to the above formulae, the chunks in the first batch have a size $F_0 = N/x_0P$. Concerning the value of x_0, for the Matrix Multiplication problem tested in [43], the coefficient of variance σ/μ is 0.032. The problem size is 300×300, and the number of processors used ranges from 4 to 56. When $P = 30$, then

$$\begin{aligned}
b_0 &= \frac{P}{2\sqrt{R_0}}\frac{\sigma}{\mu} \\
&= \frac{30}{2 \times \sqrt{300 \times 300}} \times 0.032 \\
&= 0.0016 \\
x_0 &= 1 + b_0^2 + b_0\sqrt{b_0^2 + 2} \\
&= 1 + 0.0016^2 + 0.0016 \times \sqrt{0.0016^2 + 2} \\
&= 1.0022653
\end{aligned}$$

Therefore, based on the above formulae (from [43]), almost all iterations should be assigned to processors at the first batch. However, in the experiments given in [43], only half of the iterations were assigned to processors in the first batch. The authors do not explain why the analytical results were not tested.

In addition, when $2\mu > E_{max}$ the expected finishing time of the first P chunks does not approach $2\mu F_0$. This is because the execution times of chunks in the first batch cannot be greater than $E_{max} \times F_0$.

Affinity Scheduling

The benefit of processor affinity has been demonstrated in Affinity Scheduling introduced by Markatos and Leblanc [68]. Affinity Scheduling divides the N iterations of a parallel loop into P chunks within $\lceil N/P \rceil$ iterations each. The i^{th} chunk is placed on the local work queue of processor i. An idle processor removes $1/k$ of the iterations from its local work queue and executes them, where it is suggested that k be equal to P. When its work queue becomes empty, a processor finds the most loaded processor, removes $\lceil 1/P \rceil$ of the iterations from the remote processor's work queue, and executes them.

Affinity Scheduling differs from other self-scheduling schemes in two ways. One is that it does not have a shared ready task queue. Rather, each processor has its own ready task queue. Such a distributed task queue approach eliminates the bottleneck problem of other schemes. However, when we need to balance the workload, the information regarding the workload is also distributed. This makes it difficult to achieve a balanced schedule with a low cost. The second difference is that when a processor's local ready task queue becomes empty, it attempts to remove tasks from the most loaded processors. When the number of processors is large, this approach is expensive. In addition, it may not lead to a well balanced workload. This is because when a processor's local ready task queue becomes empty, the same operation is performed regardless how many other processors are also in the same situation. To see this, consider the following scenario.

Suppose several processors complete the tasks in their local task queues at the same time and all find that a processor, say P_l, is the most loaded processor. When all try to fetch more iterations from P_l, three cases could occur. The first case is that each processor obtains some iterations and there are still some iteration left in the local ready task queue of P_l. The second case is that each processor obtains some iterations and there are no iterations left for P_l. The third case is that only a few processors obtain some iterations.

The first case is what is planned and each idle processor makes a positive step toward balancing the workload. When the second case happens, P_l then has to find more iterations from another processor for itself. This obviously is more expensive than using tasks in its own ready task queue. The third case is the most expensive one. When it happens, the processors that do not obtain any iteration waste their time locating the most loaded processor and trying to fetch iterations from it. In addition, these processors and P_l have to attempt to fetch more tasks from another processor. The same thing may happen to the most recent heavily loaded processor.

2.4 Dynamic Load Balancing on Distributed-Memory Machines

Many researchers [115, 105, 112] have studied the use of dynamic load balancing for increasing processor utilization rather than scheduling. The difference between dynamic load balancing and dynamic load scheduling is that the former achieves load balance by moving processes from one processor to another while the latter achieves load balance by assigning tasks only to processors that become idle.

Many methods have been proposed to achieve load balance on distributed-memory parallel computers using dynamic load balancing. Based on how the information regarding the load of each processor is collected and used, these methods can be classified as *centralized load balancing algorithms, fully distributed load balancing algorithms,* or *semi-distributed load balancing algorithms* [117]. In addition, these methods are further classified as *sender initiated* or *receiver initiated* [117].

In these approaches, the data partitioning problem is not addressed. Since the data modified by a migrated process has to be sent back to the owner of the data, information regarding the owner of a datum has to be stored with the datum in order to have the result sent back to the owning processor. These approaches usually operate in several phases, which include determining the local load of each processor, exchanging information so each processor can check if there is a load imbalance in the system, and migrating processes if necessary [108]. These approaches are not suitable for the problem we are studying because the work load of each processor can not be estimated accurately by counting the number of unexecuted tasks.

2.5 Self-Scheduling on Distributed-Memory Machines

Due to the mismatch between the architecture of a distributed-memory machine and the basic principle of self-scheduling and a high communication cost of a distributed-memory machine, static scheduling schemes were often used in scheduling iterations to processors on a distributed-memory machine.

Rudolph and Polychronopoulos [89] reported an implementation of GSS on distributed-memory machines using a centralized scheduling technique. They attack the data distribution problem by replicating the data to every processor. To prepare for the later usage of the data, the scheduling processor tracks, for each row of the data array, the processor modifying the row. This approach has the following problems:

1. The use of a centralized scheduling technique prevents the method from scaling very well.

2. The data distribution method limits the granularity to the row level because if we allow an arbitrary assignment of array elements to processors, then the data structure describing the array distribution would have the same number of elements as the array.

3. The problem size is limited by the scheduling processor's memory because it has to store all the data.

2.6 Assumptions

To facilitate our presentation, we assume that parallel loop L's iteration execution times follow an unknown probability distribution with mean μ, standard deviation σ, maximum execution time E_{max}, and minimum execution time E_{min}. We define that a *chunk* is a set of *consecutive* iterations defined by a starting and an ending iteration number. A *fetching processor* is a processor that modifies global variables such as the loop index to obtain more work in the form of a chunk. The *critical chunk* is the chunk finished last, and the *critical processor* is the processor executing the critical chunk.

In the sequel, we assume the number of iterations $N \gg$ P; the value of N and P are known before the loop is executed; the schedule is non-preemptive; the processors of the parallel machine are homogeneous; and the parallel loop is executed in a dedicated environment.

Many methods have been proposed to parallelize a wide range of serial loops [54, 79, 114], and nested parallel loop can be coalesced to form a single parallel loop [79]. In our study, we focus on scheduling a single parallel loop.

Chapter 3

SAFE SELF-SCHEDULING

3.1 Introduction

As we saw in the previous chapter, there are several self-scheduling schemes. However, each has weaknesses. In this chapter we introduce a new self-scheduling scheme called *Safe Self-scheduling* (SSS) that takes advantages of both static scheduling schemes and self-scheduling schemes.

SSS has been developed to schedule parallel loops with variable length iteration execution times on multiprocessors. It has two phases: a static scheduling phase and a dynamic scheduling phase. SSS achieves a well balanced workload with a low scheduling overhead. In addition, SSS maintains a high cache hit ratio to further improve the performance.

The theorems that support SSS are presented. The basis for combining static scheduling and self-scheduling in SSS are explained. We also compare our scheme with Factoring [21, 43] due the similarities between the implementations of SSS in this chapter and Factoring.

The methods discussed in the chapter have been tested. SSS has been found to surpass other schemes in most cases. In the experiment on Gauss-Jordan, an application 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.

3.2 The Basic Principle of Safe Self-Scheduling

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.,

$$\sum_{i=s}^{s+m-1} e(t_i) < \frac{E}{P} \leq \sum_{i=s}^{s+m} e(t_i)$$

where $E = \sum_{i=1}^{N} e(t_i)$ and s is the starting iteration number of the chunk. We call m the *smallest critical chunk size* because adding any more iterations to this chunk further unbalances the schedule. Clearly, E/P can only be estimated using the statistical information on the execution times of the tasks, the expected execution time of tasks, the total number of tasks, and the number of processors. When executing a parallel loop on a dedicated environment, the total number of tasks and the number of processors are known before the computation.

In the implementation of SSS, when no information regarding a loop is known, every P chunks form a batch and chunks in a batch are of the same size. The size of chunks in batch i, denoted by CS_i, is $\alpha \times N_i/P$, where α is called the *static allocation factor* and $0 < \alpha \leq 1$ and N_i is the number of unscheduled iterations at the beginning of the batch. Since the size of chunks in the first batch is known, we propose that the chunks in the first batch are assigned to processors at compile time. Scheduling these chunks forms the static scheduling phase of SSS. The remaining chunks are self-scheduled. This forms the dynamic scheduling phase or self-scheduling phase of SSS. During the dynamic scheduling phase, when a processor finishes the iterations assigned to it the i^{th} fetching processor is then assigned a chunk of

$$\max((1-\alpha)^{\lceil \frac{i}{P} \rceil} \times \frac{N}{P} \times \alpha, k)$$

iterations, where k is used to control granularity. A general method for accurately calculating α is given in form of a theorem later.

After the value of α is determined, SSS can be implemented as following.

1. (a) Before starting the statically assigned iterations, one processor (say processor 0) calculates the starting iteration numbers for the chunks scheduled in the dynamic scheduling phase and stores them in an array, say `chunk_list[]`, and appends the array with P number of 0's.

   (b) Processor 0 sets the shared variable `count` to 0 and then starts to execute the chunk assigned to it statically.

   (c) All other processors perform their computation on the statically scheduled chunks.

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 `chunk_list[i+1]` > 0, then executes the chunk defined by `chunk_list[i]` and `chunk_list[i+1] -1`.

For systems such as RP3 of IBM [78] and Ultracomputer [28] 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 chunks 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 chunk sizes.

3.3 Theoretical Basis for SSS

We define the term *balanced workload* from our perspective of loop scheduling and prove the following theorems that support 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 not greater than the maximum execution time of a loop iteration.

Theorem 3.1: If (i) we assign m iterations, where $\sum_{i=1}^{m-1} e(t_i) < E/P \le \sum_{i=1}^{m} e(t_i)$ to the first fetching processor, say p_1; (ii) the remaining iterations are allocated in such a way that all other processors have the same amount of workload; and (iii) all P processors start to execute the loop at the same time (iv) the scheduling overhead is neglected, then processor p_1 finishes no later than the critical processor p_c and the difference in workload between any two processors is less than $E_{max}P/(P-1)$.

Proof: Let $E(I_1)$ be the workload for processor p_1 and E_{rem} be the average workload of the remaining $P-1$ processors; we have

$$E_{rem} = \frac{E - E(I_1)}{P-1} = \frac{E - \sum_{i=1}^{m} e(t_i)}{P-1} \le \frac{E - \frac{E}{P}}{P-1} = \frac{E}{P} \le E(I_1).$$

Since all the processors start to execute the loop at the same time, processors with the same workload finish at the same time. In addition, since $E_{rem} \le E(I_1)$, processor p_1 finishes no later than the critical processor p_c. Further, let $E/P \le \sum_{i=1}^{m} e(t_i)$ be represented as $\sum_{i=1}^{m} e(t_i) = E/P + \beta$, where $0 \le \beta <$

E_{max}, then

$$E_{rem} = \frac{E - \frac{E}{P} - \beta}{P-1} = \frac{E}{P} - \frac{\beta}{P-1}.$$

The difference in workload between processor p_1 and any other processors is

$$\frac{E}{P} + \beta - \frac{E}{P} - \frac{\beta}{P-1},$$

which is $\beta P/(P-1)$. Since $\beta < E_{max}$, the difference in workload between any two processors is less than $E_{max}P/(P-1)$.

Theorem 3.1 states that assigning m consecutive iterations to the first fetching processor, when $\sum_{i=1}^{m} e(t_i) - E/P \leq E_{max}(P-1)/P$, achieves a balanced workload with a minimum scheduling overhead since the processor only fetches once and the difference in finish times between any two processors is less than E_{max}. Since the difference in workload between any two processors is not greater than E_{max}, then by our definition, the workload is balanced. When $\sum_{i=1}^{m} e(t_i) - E/P > E_{max}(P-1)/P$, the difference in workload between any two processors is less than $E_{max}P/(P-1)$ and can be considered to be very well balanced for large P. However, it is generally not possible to determine m since $e(t_i)$ can only be known after the task t_i has been executed.

Theorem 3.2 : 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 $E(I_j)$ be the workload of processor p_j, then

$$E(I_j) \leq (\frac{E/P}{E_{max}} - 1)E_{max} = \frac{E}{P} - E_{max}.$$

The average workload for other P - 1 processors is $E - E(I_j)/(P-1)$. In addition,

$$\frac{E - E(I_j)}{(P-1)} \geq \frac{E - (E/P - E_{max})}{(P-1)} = \frac{E}{P} + \frac{E_{max}}{(P-1)} > \frac{E}{P}$$

That is, there must exist at least one other processor that has a workload greater than $E(I_j)$, therefore processor p_j will not be the last one to finish.

According to Theorem 3.2, assigning a chunk with less than $E/P/E_{max} - 1$ iterations to a processor guarantees that this particular chunk will not unbalance the schedule. Therefore, $E/P/E_{max} - 1$ is called the *safe chunk size.* Since it is desirable to assign chunks with as many iterations as possible while maintaining load balance, chunk sizes less than $E/P/E_{max} - 1$ iterations should never be considered.

Theorem 3.3 : 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}; `(iii)` the distribution of *prob*(*then*) is uniform; `(iv)` processor p_j is assigned a chunk of size N/P and more than $N/P \times prob(then)$ iterations in the chunk have a workload E_{max}; and `(v)` $E_{max} > 2E_{min}$, 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 of the 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 an execution time 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, chunks of the same size may result

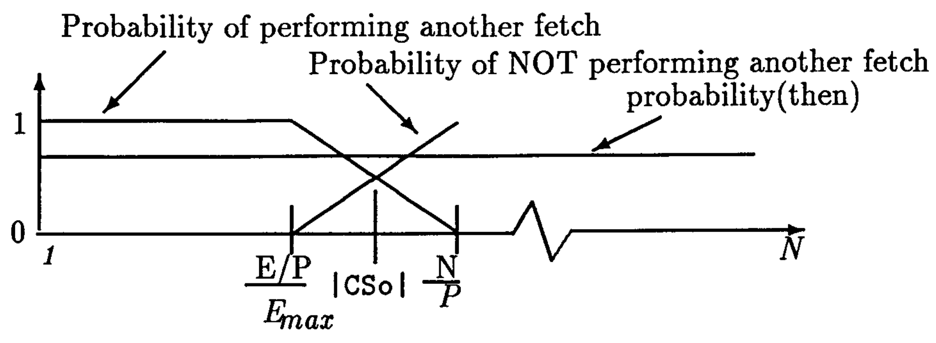

Figure 3.2. Safe self-scheduling, calculation of the first chunk size

in different finishing times. Only if iterations assigned to *one* processor happen to have more iterations having long execution times, the workload cannot be balanced. For this reason, N/P is called the *risk chunk size.*

SSS selects the first chunk size to be the point at which the probability that a fetching processor may or may not perform an additional fetch to be equal (see Figure 3.2). For loops where the execution times follow Bernoulli distribution with E_{max} having probability $prob(E_{max})$ and the probability distribution function $prob(E_{max})$ is uniform, the size of the first chunk is the average of the safe chunk size and the risk chunk size. Using $\mu \times N$ to replace its statistical equivalence E we have

$$CS_0 = \frac{\frac{N}{P} + \frac{\mu N}{E_{max} P}}{2} = \frac{N}{P} \frac{(1 + prob(E_{max}) + \frac{prob(E_{min}) E_{min}}{E_{max}})}{2} \tag{3.4}$$

$$\alpha = \frac{1 + prob(then) + \frac{prob(else) E_{min}}{E_{max}}}{2} \tag{3.5}$$

Note that, by assigning a larger number of iterations than the safe chunk size, we have accepted a moderate amount of risk of imbalance in exchange for a lower overhead. In case the overhead is small compared to the iteration execution times, a smaller value of α may be used to balance the workload.

The smallest critical chunk size can be calculated according to the theorem given below 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 3.4: A set of static allocated n-iteration chunks where n is given by

$$n = \frac{2\mu^2\frac{N}{P} + c^2\sigma^2 - \sqrt{(2\mu^2\frac{N}{P} + c^2\sigma^2)^2 - 4\mu^2(\frac{\mu N}{P})^2}}{2\mu^2} \tag{3.6}$$

with $c > \sqrt{2\ln(P)}$, will have an expected execution time less than $N\mu/P$.

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 chunks, the expected execution time is $n * \mu$ and the variance is $n * \sigma^2$. The normal distribution curve is defined as,

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

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 the chunks to finish before time t_0 is,

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

Let

$$c = \frac{t - \mu_n}{\sigma_n} \tag{3.7}$$

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

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

Let c_0 denote the value of c in Eq.(3.7) when $t_0 = N\mu/P$, we have

$$c_0 = \frac{\frac{N}{P}\mu - n * \mu}{\sqrt{n * \sigma^2}} \tag{3.8}$$

Kruskal and Weiss in [40] have shown that if each processor receives a chunk of equal size n the expected finishing time can be approximated as,

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

Let the expected finishing time to be smaller than the average processor workload $N\mu/P$. Then we have:

$$\begin{aligned} n\mu + \sqrt{2n * \sigma^2 \ln(P)} &< \frac{N}{P}\mu \\ \sqrt{2n * \sigma^2 \ln(P)} &< \frac{N}{P}\mu - n\mu \\ \sqrt{2\ln(P)} &< \frac{\frac{N}{P}\mu - n * \mu}{\sqrt{n * \sigma^2}} \end{aligned}$$

The R.H.S. is identical to c_0, so

$$c_0 > \sqrt{2\ln(P)}$$

Solving n from equation (3.8), we have,

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

3.4 Simulation Results

In this section we discuss the simulations conducted to study the effects on performance resulting from different values of α in SSS. In the simulation we assume that the loop body is an *if-then-else* statement, and the loop has 5000 iterations. The execution time of the *then* branch is set to be 4 time units and the *else* branch is set to be 1 time unit. Which branch to execute is determined by comparing two arrays `uarray[]` and `parray[]`. If `parray[i]` > `uarray[i]` then iteration i is set to execute the *then* branch, otherwise it executes the *else* branch. Elements of `uarray[]` are greater than 0.0 and smaller than 1.0 and uniformly distributed.

In order to test the effects on the selection of α in SSS on parallel loops with different characteristics, we use three groups of data and store them in three arrays `P1[]`, `P2[]`, and `P3[]` to be used as the array `parray[]`. `P1[]` contains 5000 random numbers in the range of (0.0, 1.0). `P2[]` consists of 5000 real numbers generated by using the formula

$$\frac{u'[i] + e^{-\frac{i}{100}}}{2} \times 0.8$$

where $u'[i]$ is an array of 5000 random numbers in the range of (0.0, 1.0). That is, `P2[]` is an array of 5000 real numbers in the range of (0.0, 0.08), and the values of its elements follow an exponentially decreasing curve. Similarly, `P3[]` comprises of 5000 real numbers generated by using the formula

$$\frac{u''[i] + e^{-\frac{(i-2500)^2}{2\times 0.0000001}}}{2} \times 0.8$$

`P3[i]` is a number in the range of (0.0, 0.08), and elements of `P3[]` follow a bell shaped curve.

Each simulation is implemented as following. Given two array `uarray[]` and `parray[]`, a third array `times[]` is generated where `times[i]` is 4 if `parray[i]` > `uarray[i]`, or `times[i]` is 1 if `uarray[i]` <= `parray[i]`. When this step is finished, the total amount of the workload and the frequencies of executing each of the branches are known. These pieces of information are then used to calculate the value for α. Based on the number of processors assumed to be used in executing the loop, we calculate the chunks and store the chunks in array `chunks[]`. Then the process of executing the loop using the given number of processors is simulated assuming that there is no scheduling overhead. After the loop is finished, we find the processor that has the most workload. The finish time of that processor becomes the finishing time of the simulation. In the case that there are more than one processor that all finished last, then the processor that performs the most fetches is the critical processor. For each set of value of `parray[]`, we collect the results of using 75 different sets of values for `uarray[]`.

Table I, Table II, and Table III are the simulation results of using `P1[]`, `P2[]`, and `P3[]` as `parray` to select which branch of the loop body to execute, respectively. The results shown in the tables are the average of 75 runs. The number of processors ranges from 6 to 20. What is given in the tables are the number of times the critical processor fetched and the difference between the total amount of time units assigned to the critical processor and the average workload E/P.

From Table I we can see that when $\alpha \leq 0.625$ (the one marked with †), workloads are balanced and the bigger the value for α the smaller the scheduling overhead without losing any performance. When α selects the value calculated using Eq.(3.5), the workloads are still with in 3% of the average and can be considered as well balanced. The scheduling overhead, represented by the number of fetches performed, is also small.

Table II represents the situation where the probability of an iteration executing the *then* branch decreases exponentially. This is the worst case of using fixed sized chunks in a batch because the actual amount of work represented by the first chunk and the last chunk in the same batch may very significantly. For this kind of parallel loops, chunk sizes in the same batch should increase.

From the table we can see that when $\alpha \geq 0.730$ (the one marked with *), workloads become unbalanced very quickly. Using Eq.(3.5), the calculated α is 0.73. From the table we can see that when the number of processors is greater than 10, the scheduling overhead is 0. This indicates that the statically scheduled chunks unbalances the workloads, i.e., the α is too large. Although the calculated α is larger than we would like it to be, the critical processor's workload is always with in 40 time units of a processor's average workload out of a total workload of 9204 time units. The value of α that corresponds to the safe chunk size is 0.4602.

Table I. The selection of *then* branch is uniform

α	Overhead/Amount of Time Units Over Optimal Time							
	Number of Processors & Corresponding Optimal Times							
	6	8	10	12	14	16	18	20
Opt.	2062	1546	1237	1031	883	773	687	618
0.500	8/ 2	9/ 2	8/ 2	8/ 1	7/ 3	8/ 2	9/ 2	7/ 2
0.525	8/ 1	7/ 3	7/ 3	8/ 2	7/ 2	6/ 2	6/ 2	3/ 7
0.550	7/ 2	7/ 3	7/ 3	7/ 2	6/ 4	6/ 2	8/ 2	6/ 3
0.575	7/ 2	7/ 2	6/ 3	7/ 2	6/ 3	5/ 4	5/ 4	6/ 3
0.600	7/ 2	6/ 3	5/ 3	5/ 3	6/ 2	5/ 2	5/ 4	5/ 4
0.625[†]	6/ 2	5/ 3	4/ 4	5/ 3	5/ 4	4/ 3	5/ 4	4/ 4
0.650	5/ 3	6/ 3	4/ 4	4/ 4	3/ 7	3/ 7	4/ 4	3/ 6
0.675	4/ 7	4/ 6	4/ 5	3/ 5	3/ 6	3/ 5	3/ 5	3/ 6
0.700	3/ 7	3/ 11	3/ 9	3/ 6	3/ 8	3/ 8	2/ 8	2/ 9
0.725	3/ 9	3/ 10	3/ 7	2/ 11	2/ 9	2/ 13	2/ 11	2/ 10
0.750	2/ 15	2/ 15	2/ 13	2/ 12	2/ 12	2/ 9	2/ 10	2/ 12
0.775	2/ 21	2/ 18	2/ 17	2/ 16	2/ 13	2/ 17	1/ 14	1/ 15
0.800	2/ 16	2/ 14	2/ 13	2/ 14	2/ 17	1/ 16	1/ 21	1/ 20
0.809*	2/17	2/14	2/15	2/13	1/14	1/15	1/17	1/18
0.825	2/ 14	2/ 18	1/ 17	1/ 17	1/ 22	1/ 21	1/ 22	1/ 24
0.850	2/ 11	1/ 20	1/ 19	1/ 18	1/ 22	1/ 22	1/ 19	1/ 22
0.875	1/ 23	1/ 28	1/ 29	1/ 36	1/ 33	1/ 33	1/ 28	1/ 31
0.900	1/ 41	1/ 44	1/ 39	1/ 38	1/ 36	1/ 34	1/ 32	1/ 29
0.925	1/ 55	1/ 46	1/ 37	1/ 35	1/ 33	1/ 27	1/ 27	1/ 23
0.950	1/ 42	1/ 35	1/ 29	1/ 24	1/ 24	1/ 21	1/ 22	1/ 19
0.975	1/ 25	1/ 23	0/ 24	0/ 26	0/ 28	1/ 29	0/ 31	0/ 31
1.000	0/ 55	0/ 48	0/ 52	0/ 49	0/ 48	0/ 49	0/ 47	0/ 46

Table II. The selection of *then* branch follows an exponential curve

α	Overhead/Amount of Time Units Over Optimal Time							
	Number of Processors & Corresponding Optimal Times							
	6	8	10	12	14	16	18	20
Opt.	1534	1150	920	767	657	575	511	460
0.500	8/ 2	9/ 2	8/ 2	8/ 2	7/ 3	7/ 2	8/ 3	7/ 3
0.525	8/ 1	7/ 3	7/ 3	7/ 2	7/ 3	6/ 3	6/ 3	6/ 4
0.550	7/ 2	6/ 3	6/ 3	6/ 2	6/ 3	6/ 3	6/ 3	5/ 3
0.575	7/ 2	6/ 3	6/ 3	5/ 3	5/ 3	5/ 4	5/ 4	5/ 3
0.600	7/ 2	5/ 3	5/ 4	5/ 3	5/ 3	4/ 4	4/ 5	5/ 4
0.625	5/ 3	4/ 4	4/ 4	4/ 4	4/ 6	3/ 5	4/ 6	3/ 5
0.650	4/ 5	5/ 5	4/ 6	4/ 6	3/ 7	1/ 7	3/ 7	2/ 7
0.675	3/ 9	2/ 11	2/ 11	2/ 11	2/ 13	1/ 13	1/ 12	2/ 12
0.700	2/ 16	1/ 22	1/ 20	1/ 22	1/ 22	0/ 23	1/ 23	1/ 20
0.725	2/ 18	1/ 30	0/ 34	0/ 34	0/ 35	0/ 33	0/ 34	0/ 33
0.730*	2/ 21	1/ 35	0/ 37	0/ 36	0/ 39	0/ 36	0/ 37	0/ 35
0.750	1/ 37	0/ 50	0/ 54	0/ 53	0/ 52	0/ 52	0/ 50	0/ 47
0.775	0/ 69	0/ 83	0/ 85	0/ 79	0/ 73	0/ 70	0/ 68	0/ 63
0.800	0/ 110	0/ 118	0/ 113	0/ 106	0/ 96	0/ 89	0/ 85	0/ 78
0.825	0/ 156	0/ 156	0/ 145	0/ 128	0/ 118	0/ 110	0/ 102	0/ 96
0.850	0/ 204	0/ 193	0/ 176	0/ 157	0/ 139	0/ 129	0/ 119	0/ 111
0.875	0/ 249	0/ 227	0/ 208	0/ 182	0/ 162	0/ 148	0/ 137	0/ 126
0.900	0/ 293	0/ 265	0/ 264	0/ 206	0/ 186	0/ 168	0/ 153	0/ 141
0.925	0/ 341	0/ 301	0/ 264	0/ 233	0/ 208	0/ 188	0/ 171	0/ 157
0.950	0/ 388	0/ 336	0/ 290	0/ 257	0/ 230	0/ 204	0/ 188	0/ 173
0.975	0/ 436	0/ 372	0/ 318	0/ 284	0/ 251	0/ 223	0/ 204	0/ 188
1.000	0/ 484	0/ 405	0/ 348	0/ 309	0/ 274	0/ 244	0/ 221	0/ 204

The values in P3[] follows a bell shaped curve, i.e, the closer an array element is to the middle of the array, the higher the chance that the value is a value close to 0.8, which is the largest value of the elements in P3[]. When using a set of random numbers to compare with the values in P3[], more numbers in the middle of P3[] have a value greater than the corresponding random number; therefore, more iterations near the middle of the iterations space have a longer execution time. For the results presented in Table III, the value of α for the safe chunk size is 0.55 and the value of α calculated is 0.7748. From the table we can see that when $\alpha = 0.7748$, the workloads are well balanced and the scheduling overheads are small too.

It is safe to conclude the following from this simulation. First, the smaller the α, the higher the scheduling cost. Second, the safe chunk size results in a balanced workload most of the time with a lower scheduling overhead than that of following Factoring's no-more-than-half rule [43]. Third, we observed that the value of α near the average of the calculated value and the one corresponding to the safe chunk size yields a workload within 2% of the average workload. Fourth, Factoring produces a schedule that has almost the same level of workload balance as that when using a value for α that is smaller or equal to the safe chunk size.

In general, SSS achieves a well balanced workload with low scheduling overhead most of the time. When the iterations execution times follow an exponentially decreasing curve the calculated value for α results in too many iterations being assigned to processors to start with, we argue that this represents the worst case phenomenon. In addition, the final finish times obtained using a fixed sized chunks, when a increased size chunks should be used, are within 7% of more than that of a balanced workload.

Table III. The selection of *then* branch follows a bell shaped curve

α	Exe. time (sec) & overhead in ()							
	Number of Processors							
	6	8	10	12	14	16	18	20
Opt.	1827	1370	1096	914	783	685	609	548
0.500	8/ 2	9/ 2	8/ 2	8/ 2	6/ 3	7/ 2	8/ 2	7/ 3
0.525	8/ 2	7/ 3	7/ 3	8/ 1	7/ 2	6/ 3	6/ 3	6/ 4
0.550	8/ 2	6/ 3	7/ 3	6/ 2	6/ 3	6/ 3	7/ 2	5/ 3
0.575	7/ 2	6/ 2	6/ 4	6/ 2	5/ 3	5/ 4	5/ 3	5/ 3
0.600	7/ 2	6/ 4	5/ 3	5/ 4	5/ 3	5/ 4	4/ 6	4/ 5
0.625	4/ 4	5/ 4	5/ 5	4/ 5	4/ 5	3/ 6	4/ 5	3/ 6
0.650	4/ 5	4/ 5	4/ 6	3/ 6	3/ 6	3/ 7	3/ 6	3/ 7
0.675	3/ 6	3/ 7	3/ 9	3/ 8	2/ 9	2/ 9	2/ 8	2/ 9
0.700	3/ 10	2/ 13	2/ 11	2/ 12	2/ 10	2/ 12	2/ 10	2/ 11
0.725	2/ 13	1/ 24	1/ 21	1/ 17	1/ 16	1/ 15	1/ 13	1/ 13
0.750	2/ 16	2/ 32	1/ 22	1/ 20	1/ 19	1/ 19	1/ 17	1/ 18
0.7748*	2/ 17	1/ 38	1/ 27	1/ 25	1/ 23	1/ 23	1/ 21	1/ 21
0.775	2/ 17	1/ 38	1/ 28	1/ 25	1/ 25	1/ 22	1/ 21	1/ 21
0.800	0/ 37	1/ 39	1/ 31	0/ 32	0/ 31	0/ 29	0/ 33	0/ 28
0.825	0/ 81	0/ 67	0/ 63	0/ 56	0/ 53	0/ 49	0/ 48	0/ 47
0.850	0/139	0/ 112	0/ 93	0/ 85	0/ 76	0/ 72	0/ 67	0/ 61
0.875	0/197	0/ 154	0/ 130	0/ 113	0/ 101	0/ 92	0/ 88	0/ 81
0.900	0/242	0/ 196	0/ 164	0/ 140	0/ 127	0/ 113	0/ 102	0/ 97
0.925	0/283	0/ 240	0/ 195	0/ 169	0/ 149	0/ 137	0/ 121	0/ 114
0.950	0/330	0/ 271	0/ 230	0/ 194	0/ 175	0/ 157	0/ 142	0/ 131
0.975	0/375	0/ 314	0/ 266	0/ 230	0/ 201	0/ 178	0/ 163	0/ 149
1.000	0/422	0/ 354	0/ 296	0/ 257	0/ 227	0/ 199	0/ 178	0/ 163

3.5 Comparison of SSS with Other Schemes

Comparing with GSS, SSS generates a smaller number of chunks. SSS's first several chunks are also smaller than that of GSS, and SSS finishes with chunks of small number of iterations. Comparing with TSS, SSS finishes with smaller chunks than that of TSS resulting a better balanced workload. Comparing with the Affinity Scheduling Scheme, SSS's static scheduling phase helps to maintains a high cache hit ratio.

The particular implementation of SSS given in this chapter is similar to that of Factoring [43] in the methods used to calculate the chunk sizes. Furthermore, in both schemes the chunks in the same batch have the same size. However, there are several major differences between the two schemes. The first one is that, Factoring uses the no-more-than-half rule, i.e., $\alpha \leq 0.5$ while in SSS, $0 < \alpha \leq 1$. 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 chapter assumes that little is known about the iteration execution time distribution. 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. This property improves the performance of SSS by increasing the ratio of cache hit and is proved to be extremely useful in implementing self-scheduling on distributed-memory machines [58, 92].

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 [42]. 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 chapter) 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 chunks in the first batch must not be greater than $E_{max} \times F_0$. Let further consider the following example.

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

$$\begin{aligned}
\mu &= 0.75(4.0) + 0.25(1.0) = 3.25 \\
\mu\frac{N}{P} &= 3.25(400/5) = 260.0 \\
\sigma^2 &= 0.75(4.0 - 3.25)^2 + 0.25(1.0 - 3.25)^2 = 1.6875 \\
safe\,chunk\,size &= \frac{400 \times 3.25}{5}/4.0 = 65 \\
risk\,chunk\,size &= \frac{400}{5} = 80 \\
\alpha &= \frac{0.75 + 0.25\frac{1}{4} + 1}{2} = 0.90625 \\
CS_0 &= \frac{400}{5} \times 0.90625 \approx 72
\end{aligned}$$

From the example we can see that by assigning a processor a chunk of 65 iterations (safe chunk size) cannot unbalance 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-iteration chunk is 260 time units. Hence we conclude that assigning a processor less than 65 iterations (equivalent to set $\alpha \leq 0.8125$) only results in an increased scheduling overhead. In general, for a parallel loop that has an *if-then-else* statement as it loop body, at least $N/P(prob(E_{max}) + prob(E_{min}) \times E_{min}/E_{max}$ iterations should be assigned to a processor during the first batch, where $prob(E_{max})$ is the probability of an iteration having an execution time of E_{max}. Therefore, when $prob(E_{max}) \geq 0.5$, or $E_{min}/E_{max} \geq (0.5 - prob(E_{max}))prob(E_{min})$ and $prob(E_{max}) < 0.5$, we should not used $\alpha < 0.5$.

In SSS, the value of α determines the total number of chunks produced during the execution of a given parallel loop. The larger the value of α, the smaller the number of chunks is produced, resulting in a smaller overhead. When α becomes too large, chunks with long execution times may be produced resulting in load imbalance. The smaller the value of α, the fewer the iterations that are fetched by an idle processor, therefore better the balanced workload, however, with an increased scheduling overhead. Choosing an α smaller than μ/E_{max} only causes more scheduling overhead without further balancing the workload.

The total number of chunks produced by Factoring is at least $P\,(1 + \lg(N/P))$ The total number of chunks produced by SSS is $P \lg(N/P)/\lg(1/(1-\alpha))$. For the example given above, SSS produces 10 chunks while Factoring produces at least 37 chunks. Note that a scheduling function needs to modify some global variables that have to be accessed exclusively. Frequent accessing of the shared variables such as loop index increases the time required to access them because these variables must be accessed exclusively. 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.

Finding an appropriate value of α requires some information, such as maximum and minimum execution times and prob(then) etc. We argue that it is possible to obtain approximations of these pieces of information. The execution times can be obtained through profiling utilities. The probabilities of a particular execution times can be obtained through sampling [44]. 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 IV shows the chunk sizes for several scheduling schemes on the example used above. Since the safe chunk size is 65, it is not necessary to assign a processor a chunk less than 65 iterations to start with. Note that although SSS generates total of 15 chunks, which is the smallest among all the schemes, only 10 chunks are assigned to processors during run time.

3.6 Modifications on Safe Self-Scheduling

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

3.6.1 Achieving a Higher Degree of Balanced Workload

As mentioned earlier, selecting a value of 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

Table IV. Chunk sizes for different scheduling schemes

Scheme	Chunks	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	...												

the same amount of overhead by applying a smaller value for α during the dynamic scheduling phase. Since Factoring has demonstrated its ability of producing balanced workload, using the no-more-than-half rule during the dynamic scheduling phase of SSS may improve the performance, particularly for parallel loops 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 [42] is an excellent example that exhibits such a behavior.

3.6.2 Tolerating Faulty Processors

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

We suggest the use of the GSS algorithm in the dynamic scheduling phase to make SSS also insensitive to faulty processors. The SSS-GSS scheduling can be described as given below.

1. Calculate the value for α.
2. Each processor is then assigned $N/P\alpha$ iterations statically.
3. Set the global variable `count` to be the first unscheduled iteration's number.
4. (a) begins mutual exclusion;

   (b) copies the value of `count` to `i`;

   (c) `t <- max((N - count)/p, 1);`

   (d) `count <- count + t ;`

(e) ends mutual exclusion;

(f) executes the chunk defined by `i` and `i + t` and repeats 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 chunk sizes are only slightly smaller than the new ones that would have been calculated based on the new value of P. As we already discussed, a schedule using smaller chunks, in general, results in at least as well a balanced workload as a schedule using larger chunks. Note that the step 4 above can be precalculated and stored in an array. By doing so, the critical section can be replaced by a `fetch&add` command. More discussions on scheduling under faulty processors can be found in Chapter 7.

3.6.3 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 chunks 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 enter the dynamic phase and determine the first batch chunk 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 chunk of the size

$$\max\left(\frac{N}{P}\alpha - \frac{(t_i - t_s)}{\mu}, 0\right) \tag{3.9}$$

is assigned to processor p_i. When $(\alpha N/P - (t_i - t_s)/\mu)$, the processor should then use the first available chunk in the `chunk_list[]`. The effect of this rule is that the later a processor starts, the less work it should have to complete. Following the first batch, the remaining batches are computed with the same approach previously described. Using this approach, the SSS scheme 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 $(\alpha N/P - t_x/\mu)p$ iterations can be scheduled statically by assigning to that processor with $\alpha N/P - t_x/\mu$ iterations at compile time.

3.6.4 Increasing Granularity

For fine grain parallel loop, the smallest chunk size could be more than 1. Using the similar approach as GSS(t), SSS can be modified to schedule not less than t iterations. We denoted it as SSS(t). When $E_{max} \leq h$, where h is the scheduling overhead, we have $t \geq h/E_{max}$. When $E_{min} \geq h$, $t = 1$.

3.7 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 test cases. The first test compares the SSS scheme with other well-known scheduling schemes GSS [79], TSS [109], and Factoring [42] using a parallel loop with an *if-then-else* statement as its loop body. We implement GSS as GSS(1) and TSS as TSS($N/(2P)$, 1). In the other two experiments, we apply the SSS scheme to real applications, namely matrix multiplication and Gauss-Jordan.

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

Figure 3.3. A parallel loop containing branches

3.7.1 A Parallel Loop With an *If-then-else* Statement

The first test was conducted on the loop shown in Figure 3.3. 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 executing time and the number of processors used. Clearly, a smaller cost is more desirable. The cost curves for different self-scheduling schemes executing the loop of Figure 3.3 up to 19 processors are shown in Figure 3.4. SSS outperforms the other scheduling schemes. The performance of GSS was equivalent to that for a static scheduling scheme (SC), because of uniform distribution of $prob(then)$ resulting in a small difference in the workload between any two chunks of equal size.

Figure 3.5 shows the standard deviations for the processor workload on the corresponding runs of Figure 3.4. The workload was calculated by counting DIVERSITY time units for the *then branch* and 1 time unit for the *else branch.* All the self-scheduling schemes except TSS provide balanced workload. Factoring gives the most balanced workload followed by GSS and SSS. The well balanced workload

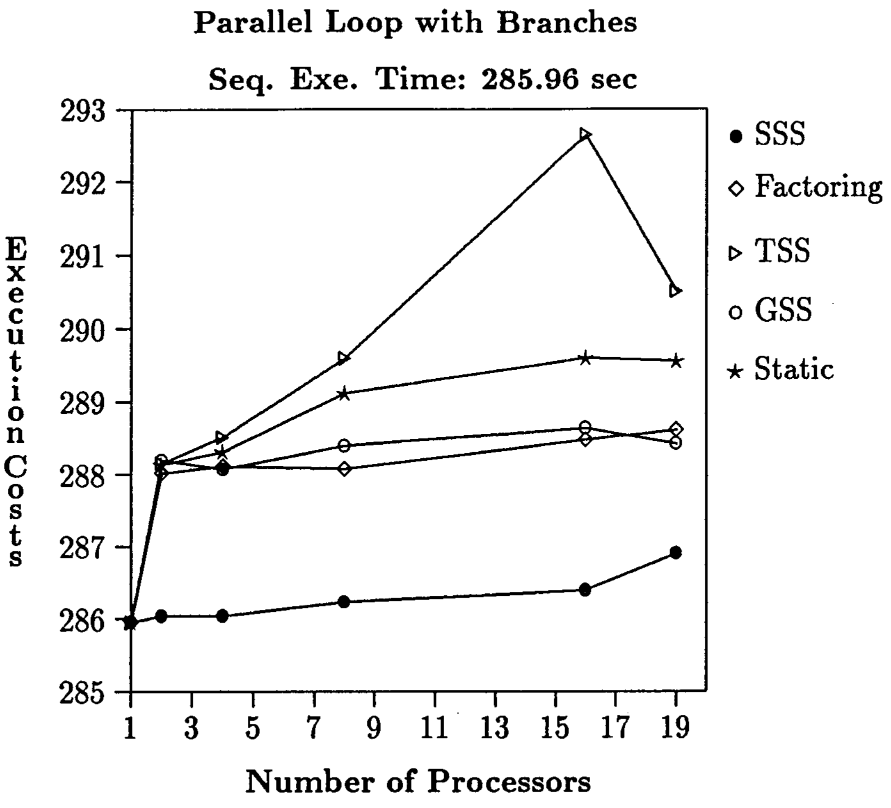

Figure 3.4. Cost curves for different scheduling schemes

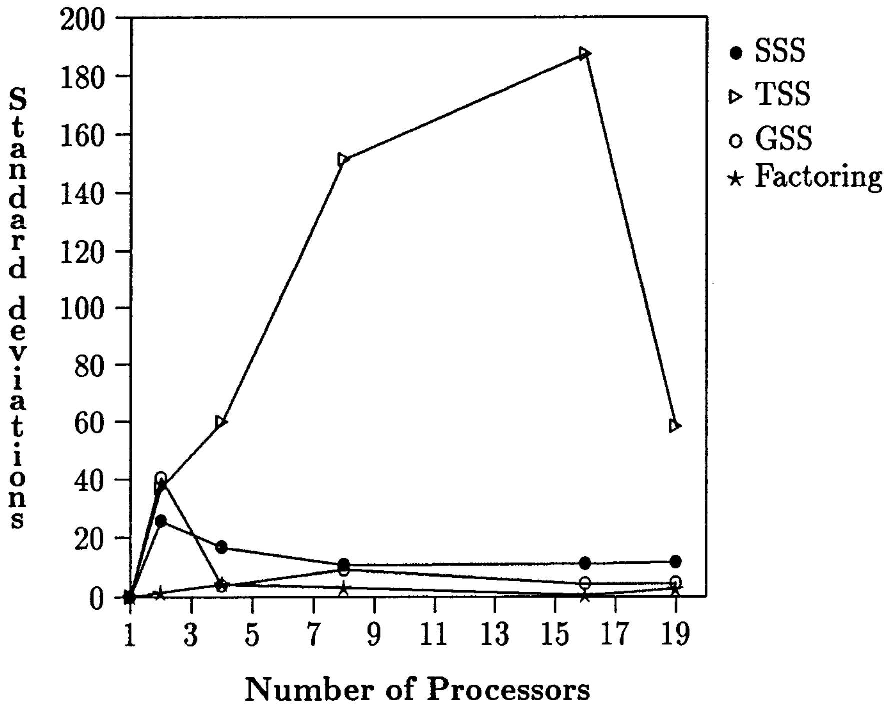

Figure 3.5. Standard deviations in workload for different scheduling schemes

of Factoring does not result in a good performance because it comes at the cost of an increased overhead in scheduling.

Figure 3.6 shows that 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. The SSS scheme surpasses other schemes in all the tests with noticeable margins. The corresponding workload balance indicated by the standard deviations is given in Figure 3.7. 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 performance than SSS because the balanced workload is achieved at the cost of a much higher scheduling overhead.

Figure 3.8 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. With increasing iteration granularity, the value of α that yields the best performance decreases. This is because (1) workload balance plays a larger and important role in the overall performance and (2) 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 balance of Figure 3.8 are indicated by the standard deviations given in Figure 3.9. 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

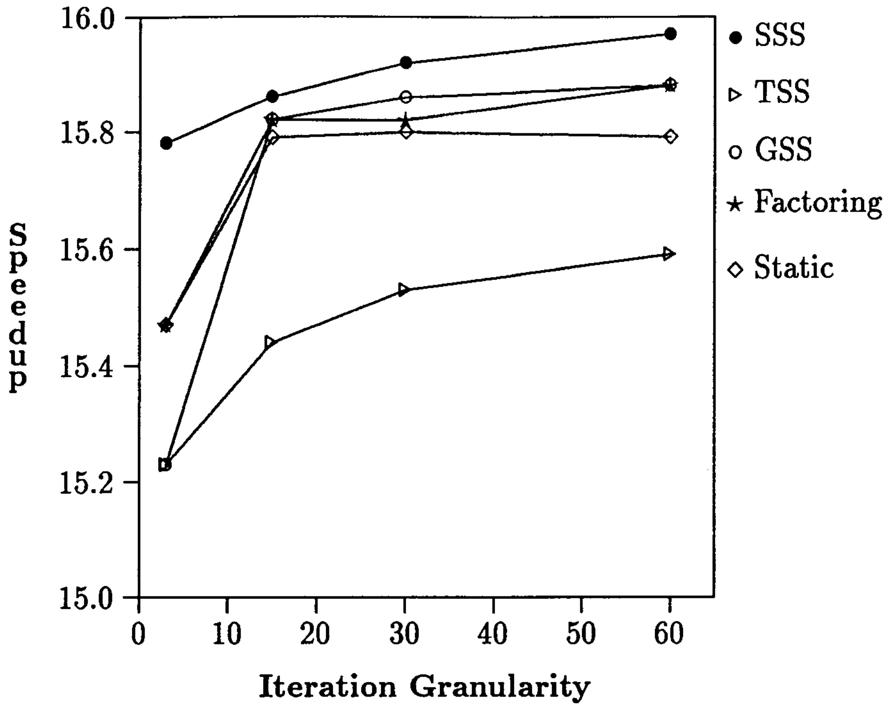

Figure 3.6. Speedup of different schemes on different granularities

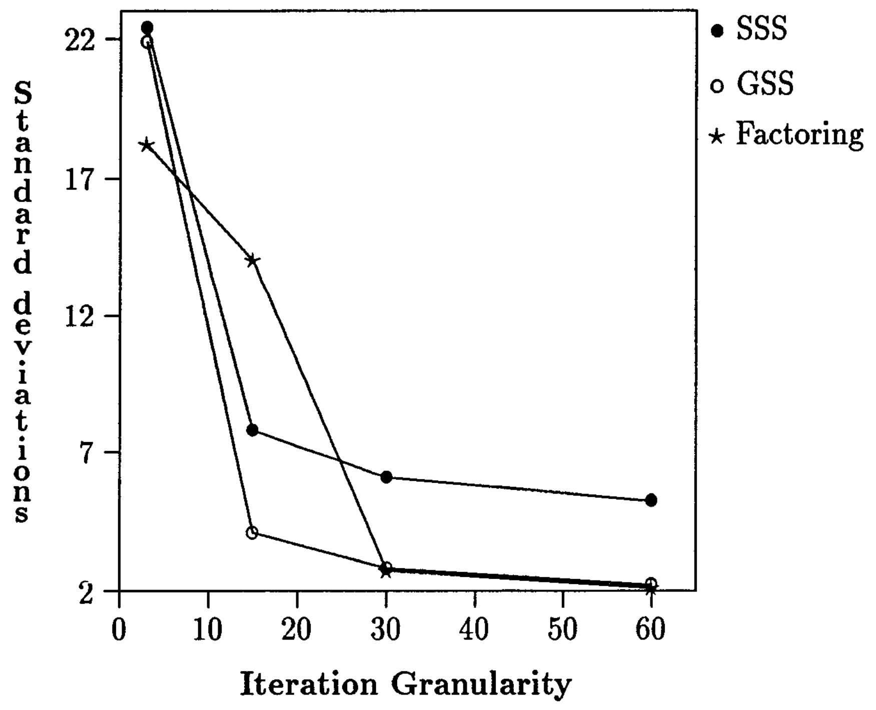

Figure 3.7. Standard deviations in workload for different schemes on different granularity

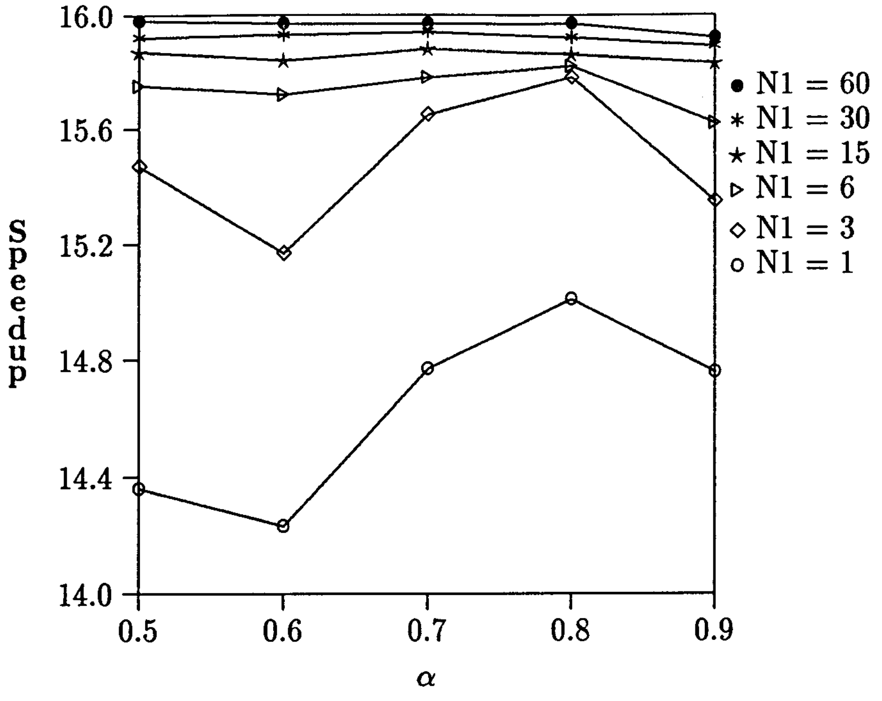

Figure 3.8. Speedup of different granularity under different α values

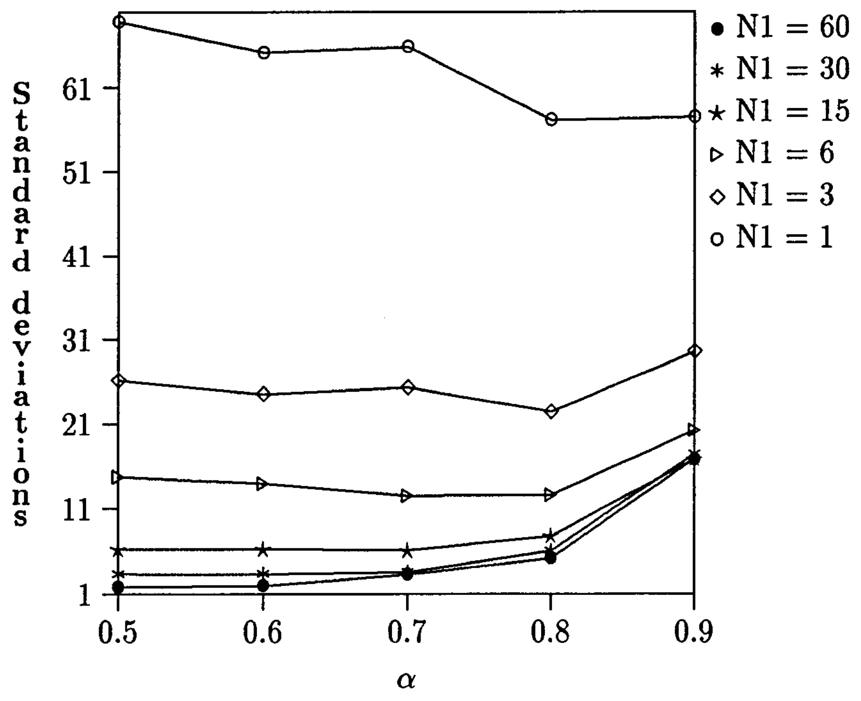

Figure 3.9. Standard deviations in workload for different granularity

```
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];
```

Figure 3.10. Matrix multiplication where many elements of matrix a are zero

too large, smaller α values do not necessarily 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.

3.7.2 Matrix Multiplication

The code in Figure 3.10 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 in the lower-triangular portion of the matrix. The outer two loops are coalesced [79]. The execution time of an iteration is between 297 μs (microseconds) and 793 μs. Using the idea of **Theorem 3.1**, 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 Figure 3.11 with the comparative results given by SS (static scheduling), TSS, GSS, and Factoring. GSS assigns too much work at the beginning. This results in a very unbalanced workload and poor performance.

3.7.3 Gauss-Jordan

Figure 3.12 shows the algorithm that performs Gauss-Jordan on an $N \times N$

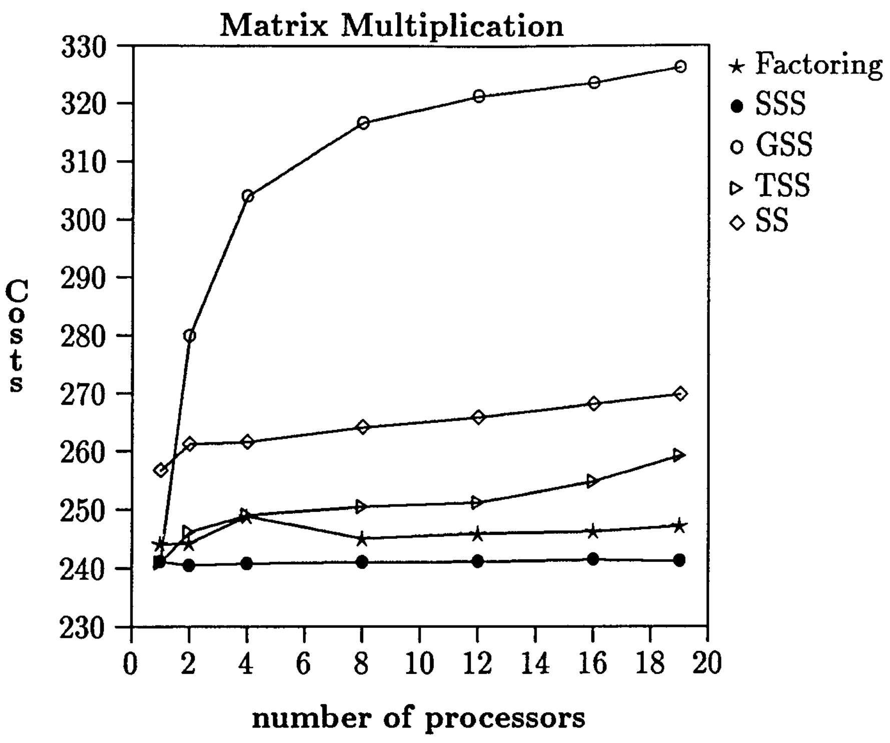

Figure 3.11. Execution cost for matrix multiply given in Figure 3.10.

```
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;
```

Figure 3.12. Gauss-Jordan

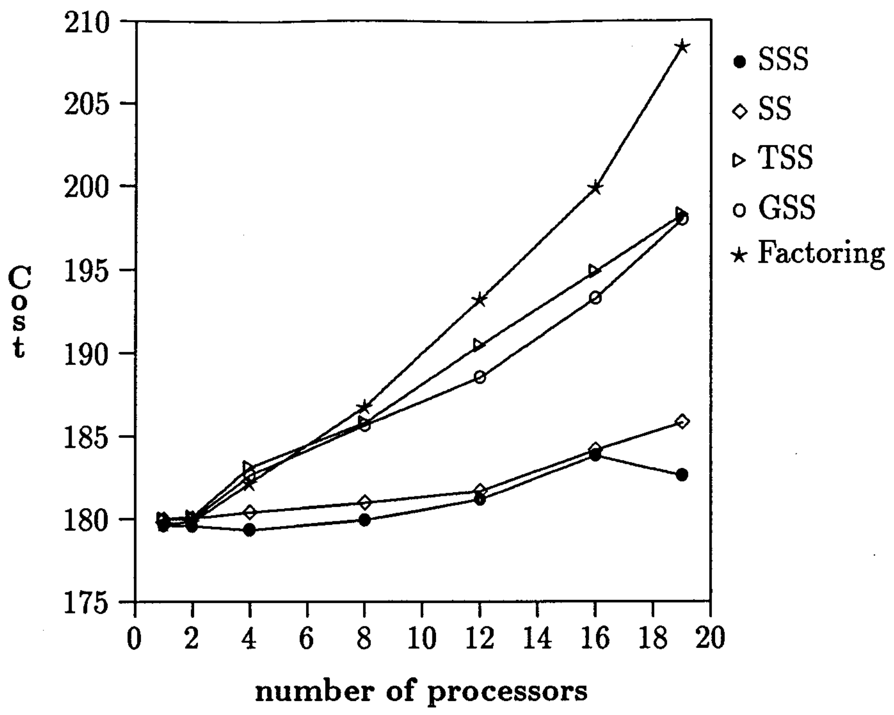

Figure 3.13. Costs of running different schemes on Gauss-Jordan.

array. Note that the iteration granularity of Gauss-Jordan is small and is independent of problem size. The amount of variance in 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 a well balanced load with a very small scheduling overhead. As shown in Figure 3.13, SSS is the only dynamic scheduling scheme that outperforms the static scheduling scheme. The reason is that SSS schedules a major portion 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

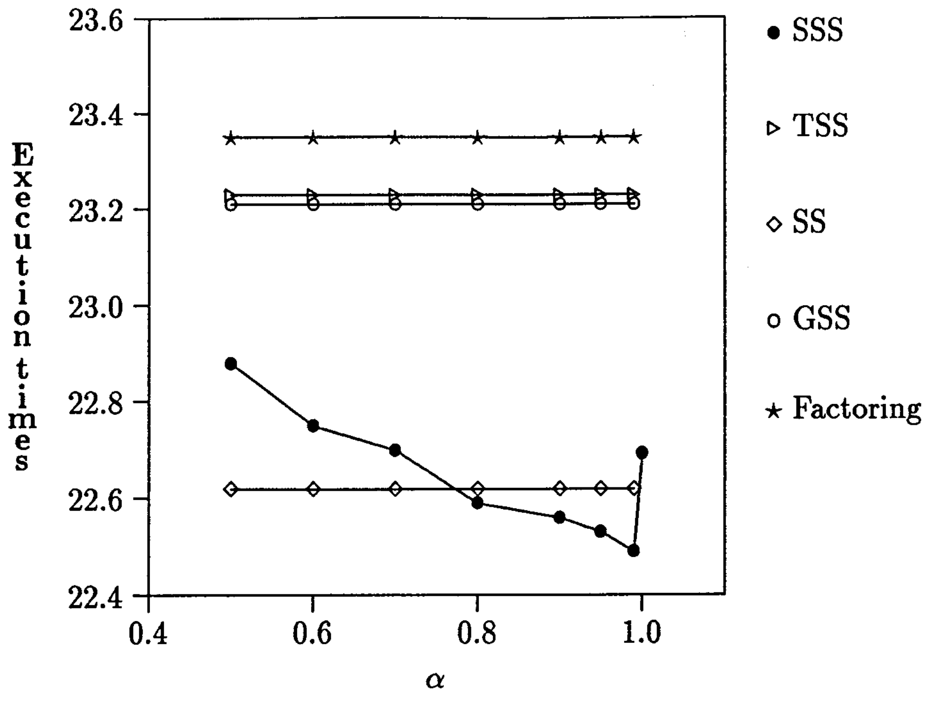

Figure 3.14. The effect of changing α on Gauss-Jordan.

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 serious when the number of processors increases.

Figure 3.14 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.

3.8 Conclusions

We have presented the Safe Self-Scheduling (SSS) scheme to schedule parallel loops with variable length iteration execution times not 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 uses self-scheduling to balance the workload at run time..

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 executing the iteration thus increases the ratio of cache hits. The other is that it reduces the scheduling overhead by assigning a large portion of iterations to processors at compile time. The importance of having a static scheduling phase is further demonstrated when self-scheduling is implemented on distributed-memory machines [58, 92].

The preliminary work of adopting SSS to a distributed-memory machine can be found in [92]. We believe that scheduling parallel loops on distributed-memory parallel computers can benefit from the two phase approach in our SSS scheme since the increased communication cost for a completely self-scheduling scheme will degrade the performance.

Chapter 4

SAFE SELF-SCHEDULING ON DISTRIBUTED-MEMORY MACHINES

4.1 Introduction

The majority of self-scheduling schemes are designed to run on shared-memory machines because a self-scheduling scheme has to maintain a shared ready task queue. A static scheduling, rather than a self-scheduling scheme, is often used to schedule a parallel loop even with uneven iteration execution times.

In the last chapter we introduced self-scheduling scheme SSS to schedule parallel loops with variable length iteration execution times on shared memory parallel computers. SSS has a unique feature, i.e., it has two scheduling phases: a static scheduling phase and a dynamic scheduling phase. We show in this chapter that this feature of SSS makes it more suitable than other scheduling schemes to run on distributed-memory machines. Another advantage is that the data used by a statically scheduled iteration can be prefetched by the processor on which the iteration is assigned.

This chapter presents the method we used to implement SSS on a distributed-memory machine such as the NCUBE/7. We call this version of SSS as DSSS (Distributed Safe Self-Scheduling). We also propose a data duplication method to minimize data movement involved for bringing data to processors for iterations scheduled in SSS's dynamic scheduling phase.

DSSS and other well known self-scheduling schemes were implemented on a 64 processor NCUBE/7. Experiments show that DSSS performs well on parallel loops with different characteristics.

One possible implementation of a self-scheduling scheme on a distributed-memory machine can be as follows. Since there is no shared-memory, the ready task queue has to be stored on one processor or distributed on several processors. Let us say processor p_j stores the ready task queue, then when a processor becomes idle, it sends a message to p_j for more work. Upon receiving a request, p_j sends the requesting processor more iterations. The following issues have to be addressed before an efficient implementation is possible.

The first is that message passing on a distributed-memory machine takes a much longer time than an exclusive access of a shared variable on a shared-memory machine; therefore, the scheduling overhead is much higher than that on a shared-memory parallel computer. The result of this is that load has to be balanced without frequent access to the shared tasks queue.

The second is that, since the processor that stores the shared ready task queue has to respond requests from other processors frequently, having this processor performs computation may delay the processing of request messages, resulting in low performance. If this processor acts only as a scheduler and does not perform any computation, the maximum potential speed up of the system becomes $(P-1)$. In addition, a single scheduling processor many become a bottleneck, degrading performance further.

The third is that, to execute an iteration, a processor must store the data needed by the iteration. If we assign iterations dynamically at run time, data has to be distributed to anticipate this assignment because data movement on a distributed-memory machine at run time degrades performance significantly.

Solutions to all the above three issues are discussed in this chapter. The rest of the chapter is organized as follows. We first discuss the Distributed SSS (DSSS) which tackles the same problems as SSS but targeted on distributed-memory machines such as NCUBE/7. Section 4.3 shows the experimental results.

4.2 Distributed SSS

DSSS (Distributed Safe Self-Scheduling) is a self-scheduling scheme that schedules parallel loops on distributed-memory machines. The parallel loops scheduled by DSSS are characterized by having variable length iteration execution times not known at compile time. DSSS has two scheduling phases: static and dynamic. The static scheduling phase serves two purposes: reducing scheduling overhead and helping data distribution. The dynamic scheduling phase balances the workload.

During the static scheduling phase, the first αN iterations, where $0 < \alpha \leq 1$, are divided into P equal sized chunks. Each of the P processors is assigned one chunk. Since the assignment of iterations to processors is determined before computation starts, data required by these iterations can also be distributed to the corresponding processors.

Processors are self-scheduled during the dynamic scheduling phase. Self-scheduling schemes can be implement on distributed-memory machine by designating one processor as the scheduler to handle all requests from other processors. A processor, called *scheduling processor*, is designated to respond to other processors' request for more chunks and assigns chunks to other processors during the dynamic scheduling phase. Since no processor requests any chunk during the static scheduling phase, the scheduling processor also performs some computation in that phase. However, assigning $N/P\alpha$ iterations to the scheduling processor may overload it.

The number of iterations processed by a scheduling processor should be greater than $\alpha NE_{min}/PE_{max}$ and less than αNP.

4.2.1 Data Partitioning

For a distributed-memory machine, data should be stored on the same processor on which the task is executed. This is not a trivial problem, because data usually has to be distributed before computation. Worse yet, when data needed by an iteration is not stored on the same processor as the iteration is assigned to, either the data must be sent to the processor or the iteration has to be reassigned to the processor that owns the data.

One approach would be to replicate all the data on every processor. For applications that process large amounts of data, as many of the applications using parallel computers do, data must be distributed among processors because the amount of data may be too large to be stored on one processor.

In DSSS duplicated copies of data used in dynamic scheduling phase are distributed onto one or more processors. Whenever a scheduling processor assigns iterations to an idle processor, it always assigns iterations to a processor that has the needed data.

In DSSS the data used by a set of iterations in dynamic scheduling is grouped into a block. Each block is then stored on two or more processors but is *owned* by only one processor which is responsible for updating the data in the block. The block size t has to be determined by either the compiler or the programmer.

A table is constructed on the scheduling processor to describe the data distribution. We call this table the *data distribution table.* Each entry of the table describes one block and has the following information: the starting iteration number, the ending iteration number, the owner of the block, and the processors where the duplicate copies of the block of data are stored.

```
a[160] : integer;
......
forall i = 1 to 160 do
    if (a[i] ≠ 0)
        then a[i] = a[i]*a[i];
        else a[i] = 0;
```

Figure 4.15. Calculating the squares.

4.2.2 Task Assignment in the Dynamic Scheduling Phase

When processor p_i finishes executing its statically assigned iterations, it sends the scheduling processor a message together with its processor id requesting for more work. The scheduling processor first tries to assign p_i with iterations that need the data owned by p_i. If these iterations have already been scheduled, the scheduling processor assigns a chunk of iterations whose data is owned by some other processors but a duplicated copy exists on p_i. To cope with the idea of SSS, the number of blocks assigned to an idle processor may decrease exponentially after every P assignments, where P is the number of processors.

4.2.3 An Example

Let us assume that we are scheduling the loop given in Figure 4.15 and see how to schedule the loop onto 8 processors with α = 0.75 and N = 160.

Partitioning

The data partitioning is accomplished through the following two steps:

1. $\alpha N/P$ array elements are distributed to each of the $P-1$ processors while the scheduling processor keeps $\lceil \alpha N/(2P) \rceil$ elements. For $\alpha = 0.75$, $N = 160$,

Table V. An Example of Data Distribution Table

	Starting Iteration	Ending Iteration	Owner	1st Dup.	2nd Dup.
1	113	118	1	3	2
2	118	123	2	1	3
3	123	128	3	2	4
4	128	133	4	3	5
5	133	138	5	4	6
6	138	143	6	5	7
7	143	148	7	6	1
8	148	153	1	7	2
9	153	158	2	1	3
10	158	160	3	2	4

and $P = 8$, the first 113 array elements are distributed to the 8 processors with 15 elements on each processor except that the scheduling processor, say p_o, with 8 elements .

2. Divide the $(1-\alpha) \times N + \lfloor \alpha N/(2P) \rfloor$ elements into l blocks of maximum e elements each. Each block is stored on one or more processors and only one processor owns the block. In this example, $e = 5$ and $l = 10$. If we decide to duplicate each block on two other processors, the data distribution table is then given as Table V.

Scheduling

The scheduling processor executes the first $\lceil \alpha N/(2P) \rceil$ iterations while processor p_i, where i = 1, 2, ..., 7, execute the chunk from

$$i \times \frac{N}{P} \times \alpha - \lfloor \alpha \times \frac{N}{2P} \rfloor$$

to

$$(i+1) \times \frac{N}{P} \times \alpha - \lfloor \alpha \times \frac{N}{2P} \rfloor - 1$$

After p_i finishes the chunk, it sends a message to the scheduling processor p_0 to obtain another chunk. p_0 first tries to assign a chunk that has its data owned by p_i. If no such chunk exists, a chunk that has its data duplicated on p_i is assigned to it. When all the chunk in the data distribution table have been scheduled, p_0 sends every processor a message indicating that there are no more iterations left.

4.3 Experimental Results

In this section, we discuss experimental results for the different scheduling schemes. The first experiment is a simulation. The second experiment uses the Monte Carlo method to calculate the weight and center of mass for an object. The last experiment applies an image processing algorithm to produce a false-color image. All the three experiments are conducted on an NCUBE/7 distributed-memory computer.

4.3.1 Simulation

A 5000 iteration parallel loop with one *if-then-else* statement is used to conduct the first simulation. For CSS (Chunk Self-Scheduling), 5 iterations at a time is assigned to an idle processor. The ratio of execution time for the *then* branch to the *else* branch is assumed to be 4. The user supplies the expected frequency with which each branch is selected.

Table VI. Simulation

	Execution time (sec) & speedup in () Sequential execution time 406.334(sec)				
Schemes	4	8	16	32	64
DSSS	137.5 (3.0)	59.7 (6.8)	28.0 (14.5)	13.7 (29.6)	6.8 (59.5)
CSS(5)	154.2 (2.6)	66.3 (6.1)	31.1 (13.0)	15.2 (26.5)	7.7 (52.0)
PSS	156.7 (2.6)	68.1 (6.1)	32.6 (12.5)	16.4 (24.8)	8.7 (52.9)
GSS	154.5 (2.6)	66.8 (6.0)	31.5 (12.9)	15.3 (26.7)	7.1 (46.7)
Static	102.2 (4.0)	51.6 (7.9)	26.3 (15.5)	13.5 (30.2)	7.4 (55.1)
Factor-ing	144.6 (2.8)	62.0 (6.6)	29.0 (14.0)	14.1 (28.8)	7.0 (57.9)

The simulation results are shown in Table VI. We also implemented static chunk and other self-scheduling schemes such as PSS, GSS, and Factoring for comparison. DSSS performs well with 64 processors. Static scheduling performs better than all the self-scheduling schemes because the variance of iteration execution time is a uniform distribution.

4.3.2 Monte Carlo Integration

Monte Carlo integration is used to find the weight and the position of the center of mass for an object of complicated shape [82]. It is used when the limits of integration of the volume cannot easily be written in an analytically closed form. To evaluate the integral of a function f over the multidimensional volume V, this method selects N random points $x_1, x_1, \cdots, x_N$ over the volume and approximates $\int f dV$ with $V/N \sum_{i=1}^{N} f(x_i)$ [82].

It is not easy to sample random points within a volume with complicated shape. In that case, the volume V can be enclosed by a larger volume W of a simple shape. In our experiment, the object evaluated is put inside a rectangular volume and sample points are chosen randomly. To ensure that the same set of sample points is used in different processors, we apply a distributed random number generator [24]. The integral of the function f is estimated as the volume W multiplied by the fraction of random points that fall within volume V.

The object in this experiment is defined by three simultaneous equations:

$$z^2 + \sqrt{x^2 + y^2 - 3} \leq 1 \tag{4.10}$$

$$x \geq 1 \tag{4.11}$$

$$y \geq -3 \tag{4.12}$$

Suppose the object has a constant density ρ. We then estimate the weight by $\int \rho dxdydz$ and the linear moments by $\int x\rho dxdydz$, $\int y\rho dxdydz$, and $\int z\rho dxdydz$. The coordinates of the center of mass is then the ratio of the linear moments to the weight of the object.

The code is shown in Figure 4.16. It constitutes a parallel loop with one *if-then* conditional statement. The number of iterations is the same as the data points selected, which is 250,000.

The result of this experiment is shown in Table VII. When the number of

```
forall (i=0; i < 250000; i++)
   x = 1.0 + 3.0 * random(&R, &A, &C);
   y = (-3.0) + 7.0 * random(&R, &A, &C);
   z = (-1.0) + 2.0 * random(&R, &A, &C);
   if (sqr(z) + sqrt(sqr(x) + sqr(y) - 3.0) < 1.0)
      {
      /* estimate the weights and linear moments */
   }
```

Figure 4.16. Code for Monte Carlo Integration

processors is small, i.e., smaller than 16, static scheduling performed better than other self-scheduling schemes. DSSS gives a better performance with more than 16 processors. The better performance results from a well balanced workload and a low scheduling overhead. The expected frequency of executing the *then branch* is estimated to be 0.75 and α is calculated to be 0.88.

4.3.3 Generation of False-Color Image

The color image produced by translating a monocolor image into a color presentation is called false-color. This technique is often used to display data that is not inherently imaging in nature.

In this experiment, the monocolor image is stored in an array. Each element of the array contains an integer ranging from 0 to 1000 to represent different gray levels of a pixel. Each gray level is then mapped to a color which is represented by a number ranging from 0 to 255. The computations among different pixels are different because the execution time of the function that maps a gray level to its corresponding color differs according to its input. We only implemented static scheduling scheme and DSSS because other scheduling schemes do not include data

Table VII. Monte Carlo Integration

	Execution time (sec) & speedup in () Sequential execution time 118.962(sec)				
Schemes	4	8	16	32	64
DSSS	41.0 (2.9)	17.6 (6.7)	8.3 (14.3)	4.2 (28.0)	2.2 (55.3)
CSS	42.3 (2.8)	18.8 (6.3)	9.3 (12.8)	5.0 (23.8)	3.1 (38.9)
GSS	40.7 (2.9)	17.5 (6.7)	8.3 (14.3)	4.4 (27.0)	2.4 (49.7)
Factoring	41.8 (2.8)	17.5 (6.8)	8.1 (14.6)	4.4 (27.3)	2.3 (52.0)
Static	30.9 (3.9)	15.5 (7.6)	7.9 (15.1)	4.4 (26.8)	2.3 (52.4)

partitioning. The image tested has 512×512 pixels.

For DSSS, data needed in the static scheduling phase is prefetched. Data needed in the dynamic scheduling phase is grouped into blocks of 16 elements and duplicated copies are stored on all other processors. Table VIII shows the performance of the DSSS with $\alpha = 0.55$ and the static scheduling. The improvement in speedup by DSSS comes from better utilization of processors. The time for 1 processor is estimated, since the data array is too large to be stored on one processor.

Table VIII. Generation of a False-Color Image

	Execution time (sec) & speedup in () Sequential execution time 81.707(sec)				
Schemes	4	8	16	32	64
DSSS	24.1 (3.4)	11.1 (7.4)	5.5 (14.9)	2.8 (29.5)	1.5 (54.1)
Static	23.9 (3.4)	13.7 (6.0)	8.5 (9.6)	5.0 (16.2)	2.609 (31.3)

4.4 Conclusions

We demonstrated a successful attempt in applying SSS to schedule parallel loops with variable length iteration execution times on distributed-memory machines. The iteration execution times may not be known at compile time.

The approach introduced in this chapter makes good use of the two phases approach of SSS. The advantage of applying SSS's static scheduling phase is that, first, scheduling overhead is reduced, and, second, a major portion of data is distributed during this phase.

The dynamic scheduling phase balances the workload. The data needed in the dynamic scheduling phase is grouped into small blocks. Each block is then stored on one processor and that processor is designated as the owner of the block. The same block of data is then duplicated on limited number of other processors. In this way, an iteration can be assigned to a processor that either owns the data needed by the iteration or has a duplicated copy of the data needed by the iteration.

We showed that DSSS offers better performance than other self-scheduling schemes. Compared with static scheduling, DSSS surpasses static scheduling scheme when the number of processors is large. As much as 79% of improvement over static scheduling has been achieved by using DSSS. The same techniques used by DSSS can also be applied to other self-scheduling schemes.

Chapter 5

A GENERAL APPROACH FOR SELF-SCHEDULING ON DISTRIBUTED-MEMORY MACHINES

5.1 Introduction

In the last chapter we discussed DSSS, an implementation of SSS on a distributed-memory machine. Enlightened by the techniques used in DSSS, in this chapter we present a general approach that supports the implementation of a given self-scheduling scheme on distributed-memory machines.

This chapter discusses self-scheduling of non-uniform parallel loops on distributed-memory machines. The chapter focuses on both workload balance and data distribution, the two main issues in scheduling non-uniform parallel loops on distributed-memory parallel computers.

The approach again has two phases: a static scheduling phase and a dynamic scheduling phase. The static scheduling ameliorates the high scheduling overhead in a distributed-memory machine and also makes it possible to prefetch the data needed by the statically scheduled iterations. The workload is balanced in the dynamic scheduling phase.

We classify data distribution methods into four categories and present *k-duplication of partial array*, a method that allows the problem size to grow linearly in the number of processors. We also present a multilevel scheduling scheme that alleviates the problem of the scheduling processor being a bottleneck.

Combining the new data distribution methods with the general approach for self-scheduling of parallel loops, a user can expect to solve larger problems efficiently by employing more processors.

The rest of the chapter is organized as following. Section 5.2 discusses a general approach for implementing self-scheduling schemes on distributed-memory machines. Section 5.3 presents the data distribution policies needed by different methods of assigning iterations to processors. We propose a multilevel scheduling scheme in Section 5.4. Experimental results are presented in Section 5.5. We conclude the chapter in Section 5.6.

5.2 Self-Scheduling on Distributed-Memory Machines

On a shared-memory machine, the ready task queue is stored in the shared-memory where each processor has access to it; although, exclusive access is required to guarantee that every iteration is executed once and only once. On a distributed-memory machine, unlike shared-memory machine, the ready task queue needs to be stored on one or more processors' local memory. We call these processors the *scheduling processors.* The other processors, which we call the *working processors*, then have to request for a task by sending a message to a scheduling processor. In this section we discuss a general method for implementing a self-scheduling scheme on a distributed-memory machine with only one scheduling processor. This restriction is then relaxed in later sections.

Let $C_1, C_2, \cdots, C_T$ be the T chunks generated by a self-scheduling scheme S on the iteration space $I = \{t_1, t_2, \cdots, t_N\}$ of parallel loop L. We propose the following two-phase approach to implement the self-scheduling scheme S on a distributed-memory machine. In the first phase, a chunk C_i is statically assigned to processor i, where $1 \leq i \leq P$. Since the assignment of iterations to processors is determined at compile time[2] data required by these iterations is also distributed to the corresponding processors at compile time. For the self-scheduling scheme that generates a decreased sized chunks, the first P statically allocated chunks usually account for a major portion of the iterations.

In the second phase, processors are self-scheduled to balance the workload. The processor that finishes first[3] in static scheduling phase becomes the scheduling processor. The scheduling processor responds to other working processors' request for more iterations and assigns iterations to other processors during this self-scheduling phase. Chunk C_j, where $p < j \leq T$ is assigned to a processor that stores the data needed by the iterations in C_j. How to distribute data is discussed in next section.

Figure 5.17 (a) shows the flow chart for a *scheduling processor.* The processor starts with executing its chunk of iterations allocated at compile time. It then calculates the chunks scheduled in dynamic scheduling phase according to the scheduling scheme S to generate a ready task queue. Upon receiving a request from a *working processor*, it removes a chunk from the queue and assigns the chunk to the working processor. When the list becomes empty, the scheduling processor broadcasts a message to all the working processors to indicate that there is no more tasks.

Figure 5.17 (b) is the flow chart for a *working processor.* After finishing the

[2]Here *compile time* means the time the values for parameters N, P, and the scheduling scheme become available.

[3]In practice, identifying the first finished processor is non trivial problem. In our implementation we always select processor 0 as the scheduling processor by assigning smaller number of iterations during the static scheduling phase.

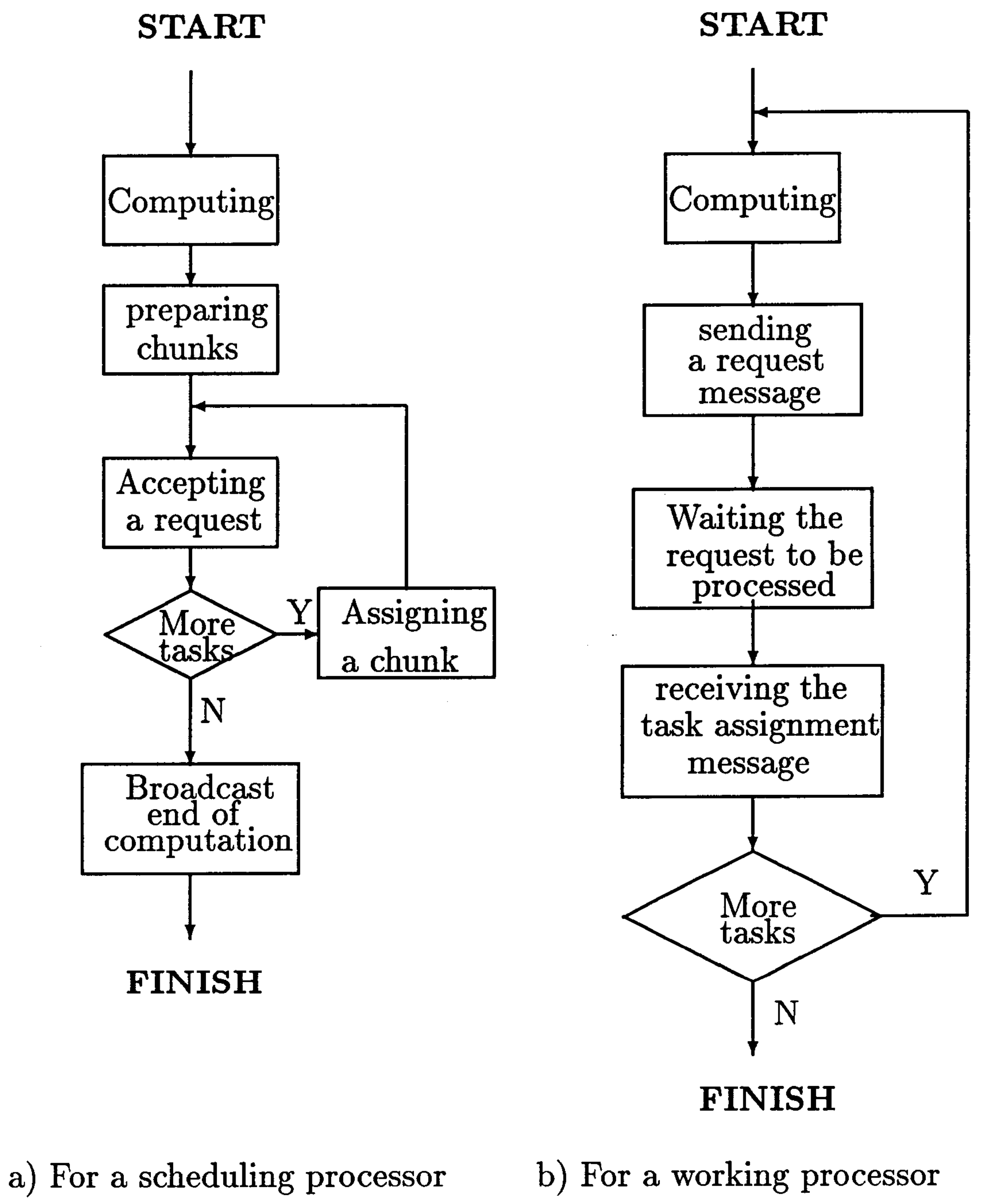
START
Computing
preparing chunks
Accepting a request
More tasks
Y
Assigning a chunk
N
Broadcast end of computation
FINISH
a) For a scheduling processor
START
Computing
sending a request message
Waiting the request to be processed
receiving the task assignment message
More tasks
Y
N
FINISH
b) For a working processor

Figure 5.17. The execution process of a parallel loop

statically assigned chunk, it sends a message to the scheduling processor requesting another chunk of iterations. It then waits for a message from the scheduling processor. If the message contains a chunk, it performs the computation defined by the chunk. For systems that have communication co-processors, the request messages for additional tasks may even be sent before the completion of a chunk. By doing so, the overhead in waiting for additional work from the scheduling processor can be greatly reduced.

Note that designating a processor as the scheduling processor is not necessary during the static scheduling phase since working processors do not make any requests for iterations during this phase. However, a scheduling processor is necessary during the dynamic scheduling phase to store the shared information such as the loop index. To store the shared information on more than one processor incurs additional overhead in maintaining the consistency of the information. Having the scheduling processor also perform computation as other *working* processors in the dynamic scheduling phase may result in a delay in processing the requests for additional work from other processors.

5.3 Data Distribution Policies for Self-scheduling

In the last section we presented a method of assigning iterations of parallel loops to processors of a distributed-memory parallel computer. To allow an efficient implementation of the method, data has to be distributed to anticipate the assignments of iteration. In this section we discuss the data distribution policies that are suitable for different scheduling schemes.

To schedule a parallel loop on a distributed-memory machine, an iteration must be assigned to a processor that stores the data needed to execute the iteration. Otherwise, the iteration has to be re-assigned (ideally) to a processor that stores the data, or message passing has to be invoked to bring in the data to the processor to which the iteration is assigned. Both of these methods are expensive to be performed frequently and should be avoided.

Different scheduling schemes require data being distributed differently. For example, if the parallel loop is scheduled statically, the data needed by an iteration can be prefetched because the processor on which the iteration is assigned is predetermined. On the other hand, if the loop is scheduled using PSS, the data needed by an iteration has to be stored on every processor so that a processor can carry out the iteration immediately. This is because the iteration can be assigned to any processor. We classify the data distribution policies into four categories:

1. Total Replication of Full Array (TRFA)
2. Total Replication of Partial Array (TRPA)
3. K-Duplication of Partial Array (KDPA)
4. No Duplication

Replication refers to a piece of data that is stored on all the processors. *Duplication* refers to a piece of data that is stored on a fixed k number of processors. This fixed number k is independent of the total number of processors and $1 < k < P$. Table IX lists the four data distribution categories and some of the scheduling schemes that use them. Most of the self-scheduling schemes discussed in Chapter 2 require a total replication of data because an iteration can be assigned to any processor. SSS for distributed-memory (DSSS) statically assigns the first P chunks; therefore, the data associated with these chunks can be prefetched. Since data needed by self-scheduled iterations can either be replicated or duplicated, data for DSSS can be either partially replicated or partially duplicated, depending on the amount of data needed by the loop.

Table IX. The data distribution categories and the corresponding loop scheduling schemes

Data distribution	Scheduling schemes
total replication of full array	GSS, TSS, FACTORING, PSS, CSS
total replication of partial array	DSSS
k-duplication of partial array	DSSS
no duplication	Static scheduling schemes

For the general approach discussed in the last section, the data used by a statically scheduled iteration does not need to be duplicated or replicated. Only the data used in the dynamic scheduling phase needs to be either duplicated or replicated. If a scheduling scheme uses no static scheduling phase then the total replication of full array is used. In the absence of the dynamic scheduling phase no data needs to be duplicated. Whether to use k-duplication of partial array or total replication of partial array in the self-scheduling phase depends on many factors that we discuss below.

5.3.1 Total Replication of Full Array

In the total replication of full array, the data used by an iteration is stored on all the processors. This method should be used when a fixed amount of data is needed by the entire loop. In addition, this amount is independent of the problem size and the data is usually not modified. An example of this kind of applications is using Monte Carlo integration to find the weight and the position of the center of mass of a complicated shaped object [82].

The total replication of full array method is also used when all the iterations are assigned to processors at run time. For example, if a parallel loop is scheduled using GSS, then the data has to be replicated to all the processors because an iteration can be assigned to any processor. In this case, the largest problem solvable using total replication of full array is limited to the problem solvable on a single processor. For example, for a particular machine, if we can store 1 million integers on one processor, then regardless of the number of processors the machine has, the problems solvable on this machine cannot use more than 1 million integers. This data distribution method has minimum scalability. Clearly, this is not acceptable because many scientific computations often scale with the available processing power. In addition, maintaining the consistency of the P copies of a piece of datum distributed on all the processors may also severely degrade the performance.

5.3.2 Total Replication of Partial Array

In total replication of partial array, only a part of the data is stored on all the processors and the rest of the data is partitioned into pieces and each piece is stored on a different processor. For example, DSSS has two phases: a static scheduling phase and a self-scheduling phase. A piece of data used by a statically scheduled iteration needs only be stored on the processor to which the iteration is assigned. The data used by a dynamically scheduled iteration is replicated to all the processors.

Theorem 5.1 Let N be the problem size that can be solved using the total replication of full array and N' be the problem size that can be solved using total replication of partial array. Assuming that $N' \times \beta$ amount of data is stored without duplication or replication and evenly distributed, where $0 < \beta \leq 1$, then we have

$$N' = \frac{N}{\frac{\beta}{P} + 1 - \beta}.$$

Proof: For all the data stored on a processor, $N'/P \times \beta$ is stored on this processor only and $N'(1-\beta)$ is replicated on this and other processors. Since the amount of memory used by both of these methods are the same, i.e.,

$$N = \frac{N'}{p}\beta + N'(1-\beta) \tag{5.13}$$

By solving N' from Eq.(5.13) we have

$$N' = \frac{N}{\frac{\beta}{P} + 1 - \beta} \tag{5.14}$$

□

Since $\beta/P + 1 - \beta < 1$, it is always true that $N' > N$, i.e., we can always solve larger problem by using total replication of partial array than using the total replication of full array .

For example, if $\beta = 0.9$, then 90% of iterations are scheduled statically and have their data stored on only one processor. Suppose that the rest of data is replicated. For a particular machine, if we can store 1 million integers on one processor, then for a machine with 16 processors, by using Eq.(5.14) with $N = 1,000,000$, $\beta = 0.9$, and $P = 16$, the similar problems solvable can use as many as 6.4 million integers of data. This method can be used when the problem size, i.e., the number of iterations in the parallel loop, does not grow linearly in the number of processors.

5.3.3 K-Duplication of Partial Array

K-duplication of partial array is similar to total replication of partial array except that the replicated data is now duplicated to a *fixed number* of processors. In k-duplication of partial array, no data (needed by only a particular iteration) is stored on all the processors.

In order to implement a given scheduling scheme, chunk sizes are calculated according to the scheme. The data used by a chunk of iterations is grouped together and called a block. The block of data is stored on some fixed k number of processors, where $1 < k < P$. The reason for $k < P$ is that we assume one of the processors is the scheduling processor and it does not perform any computation during the dynamic scheduling phase. Each of these data blocks, although duplicated on more than one processor, is *owned* only by a designated processor which is responsible for updating its values. Every $P - 1$ consecutive chunks of iterations form a batch. Each of the $P - 1$ chunks in a batch is assigned to a processor that owns the data needed by that chunk. A block of data is also duplicated on other $k - 1$ processors. Note that a chunk may not necessarily be assigned to the designated processor for execution during the dynamic scheduling phase. Rather, any processor that stores the data needed by the chunk can execute the iterations in the chunk. The information of this mapping of blocks of data to the $P - 1$ processors is stored in a table on the scheduling processor.

During the self-scheduling phase, an idle processor sends a request to the scheduling processor for additional work. The scheduling processor first tries to assign the requesting processor a chunk whose associated data block is owned by the requesting processor. If that chunk has already been scheduled to another processor, the scheduling processor then assigns a chunk *within the same batch* for which the data is duplicated on the requesting processor. If all such chunks in that batch are scheduled, the scheduling processor then attempts to assign a chunk from the next batch in a similar fashion. For the reason of achieving a balanced workload, a larger chunk should always be assigned before a small chunk. Since chunk sizes decrease in the later batches, the scheduling processor always tries to schedule larger chunks in the current batch before assigning chunks from the next batch.

Assigning chunks of iterations

To assign chunks to processors efficiently, we propose a method below that uses three arrays. The array `chunks[]` stores the chunks of iterations assigned to processors during the dynamic scheduling phase. A chunk is defined by a starting iteration number and an ending iteration number. Chunk i's starting iteration number is stored in `chunks[i]`. The ending iteration number is `chunks[i + 1] - 1`. The array `flags[]` is used to record if a chunk is scheduled or not. `flags[i]` is set to `false` if chunk i has been scheduled and set to `true` otherwise. The sizes of array `chunks[]` and `flags[]` are the same but vary for different scheduling schemes. The array `table[][]` is a two-dimensional array that has $P-1$ rows. Row i contains the indices of the array elements of `chunks[]` that have their data stored on processor $i+1$. The size of a row is proportional to the size of `chunks[]`. `table[i][0]` is used to index the next chunk, in row i, to be assigned to processor $i+1$. Before a chunk is actually assigned, the scheduling processor checks the corresponding element in the array `flags[]` whether the chunk has been assigned to another processor already, and if so, `table[i][0]` is incremented by one and the chunk indicated by the new value of `table[i][0]` is checked. This process continues until either an unassigned chunk is found or all the chunks in row i have been scheduled.

For a simple implementation of above scheduling policy, we logically view the processors connected in a ring. The data owned by a processor is duplicated on its two neighboring processors. If a chunk whose data is owned by processor $i + 1$, then the chunk number is stored in `table[i][j]` where j mod k is equal to 1. If a chunk that has its number stored in `table[i][j]` where j mod k is not equal to 1, it is duplicated on processor $i + 1$. Given a chunk c, its data is owned by processor $(c \bmod (P-1)) + 1$.

The above method describes a simple method of deciding on which processors a block of data should be duplicated. The data is not necessarily duplicated on the neighboring processors. Other assignments can be chosen as well. In addition, the data may also be duplicated on more than $k = 3$ processors. However, as k increases there is more flexibility in assigning iterations to processors at run time; but the storage requirement also increases proportionately.

An Example of K-Duplication

Let us study the scheduling of a parallel loop with 160 iterations on, for the purpose of illustration, a 5-processor distributed-memory machine, i.e., $N = 160$ and $P = 5$. Assume that the chunks are calculated using SSS with $\alpha = 0.8$; the processors are logically connected in a ring; the data associated with the chunks in the dynamic scheduling phase is stored on three processors, i.e., $k = 3$. The data associated with a chunk is duplicated on its owner's two neighboring processors.

Static scheduling phase

In static scheduling phase, there are 5 processors performing computation. A working processor executes

$$\lceil N/P \times \alpha \rceil = \lceil \frac{160}{5} \times 0.8 \rceil = 26$$

iterations. Assuming the scheduling processor only executes half of what other processors execute, then 117 iterations are scheduled statically. The data associated with these iterations is prefetched and stored in the local memory of each processor.

Dynamic scheduling phase

When the scheduling processor P_s finishes the chunk of iteration assigned to it during the static scheduling phase, it fills in the array `chunks[]` according to the specific self-scheduling scheme (SSS in this example). Table X shows the elements of the array `chunks[]`. There are 8 chunks in dynamic scheduling phase. Chunk

Table X. The values for elements of array `chunks[]`

	values								
indices	0	1	2	3	4	5	6	7	8
values	117	126	135	144	153	155	157	159	160

i's starting iterations number and ending iterations number are given by `chunk[i]` and `chunk[i + 1] - 1`, respectively.

Information regarding the data distribution is stored in the array `table[][]`. Table XI shows the elements of the array `table[][]`. When an idle processor, say processor 1, sends a request to P_s for more iterations, P_s checks the first row of `table[][]`. If `table[0][0]` is 1, the scheduling processor then attempts to schedule the chunk indexed by `table[0][table[0][0]]` which is `table[0][1]` and has a value of 0 according to Table XI. If chunk 0 has not been scheduled, then iterations from `chunk[0]` to `chunk[1] - 1`, i.e. iteration from 117 through 125 are assigned to processor 1; `table[0][0]` is incremented; and `flags[0]` is set to `false`. However, if chunk 0 is already scheduled (when `flags[0]` is `false`), then the scheduling processor increments `table[0][0]` and checks the chunk indicated by `table[0][table[0][0]]`. The scheduling processor repeats the process until an unscheduled chunk is found or all the chunks indicated in the row are checked.

Clearly, k-duplication of partial array greatly increases the size of problems solvable in terms of the data that can be stored. The actual amount of data can be stored is given by the theorem below.

Theorem 5.2 Let N be the problem size that can be solved using the total replication of full array and N' be the problem size that can be solved using k-duplication of partial array. Assuming that $N' \times \beta$ amount of data is not

Table XI. The values for elements of array `table[][]`

	values						
Columns	0	1	2	3	4	5	6
`table[0]`	1	0	3	1	4	7	5
`table[1]`	1	1	0	2	5	4	6
`table[2]`	1	2	1	3	6	5	7
`table[3]`	1	3	2	0	7	6	4

duplicated, where $0 < \beta \leq 1$. The rest of data is divided into blocks according to chunks, and the data owned by a processor is duplicated on other $k-1$ processors, then we have

$$N' \approx \frac{N \times P}{1 + k \times (1-\beta)}$$

Proof: For all the data stored on a processor, $N' \times \beta / P$ amount of data is stored on the processor only, $N'(1-\beta)/(P-1)$ amount of data is owned by and stored on the processor, and $N'(1-\beta)/(P-1) \times (k-1)$ amount of data is stored on but not owned by the processor. Since

$$N = \frac{N'}{p}\beta + \frac{N'(1-\beta)}{(P-1)} \times (k-1) + \frac{N'(1-\beta)}{(P-1)}$$

we have

$$N = \frac{N' \times \beta}{P} + \frac{k \times N'(1-\beta)}{(P-1)}$$

If we approximate (P -1) with P, we then have

$$N \approx \frac{N' \times \beta}{P} + \frac{k \times N'(1-\beta)}{P}$$

By solving N' we have

$$N' \approx \frac{N \times P}{k - \beta \times (k-1)} = \frac{N \times P}{1 + k \times (1 - \beta)}$$

□

Theorem 5.2 reveals two important properties about k-duplication of partial array. First, assuming that each iteration needs a fixed amount of data, the problem size solvable using this data distribution method is linear in the number of processors. This property is proved as a Corollary below. This is an important property if we want to solve larger problems by increasing the number of processors.

Corollary 5.1 For a parallel loop, if each iteration needs a fixed amount of data and the data is distributed using the k-duplication of partial array method, then the size of problems solvable is linear in the number of processors.

Proof: From theorem 5.2 we have that the size of problems solvable on P processors N' is approximated to be $N \times P(k - \beta \times (k-1))$, where N is the problem size solvable on one processor. Since both k and β are constants, so we have $O(N') = O(P)$. □

The second property is that the selection of k is the result of considering the trade-off between the problem size solvable and the possibility of an idle processor having the data owned by a busy processor. A large k limits the size of problems solvable because duplicated copies of data need additional memory space, while a small k decreases the chance of an idle processor having the data owned by a busy processor.

Note that chunks may not be assigned to processors by the order listed in `chunk[]`. Instead, when a processor becomes idle, the biggest chunk whose data is stored on the idle processor is assigned to the processor. When all the chunks whose data stored on the idle processor have been assigned, the idle processor remains idle until all the iterations are executed.

5.3.4 No Duplication

No duplication refers to the situation where data used by one iteration is stored only on one processor. The only cases where this data distribution method should be selected over others is when the parallel loop is statically scheduled. In this case, the processor to which an iteration is assigned is known at compile time so the data needed by the iteration can also be distributed to the processor before computation. For applications where the execution times between iterations do not vary much, static scheduling schemes are used in conjunction with this data distribution method.

As mentioned before, when an iteration is assigned to a processor that does not have the required data, the iteration can be re-assigned to a processor that has the data or message passing can be invoked to bring in the data to the processor. If any of the above two approaches of ensuring a processor having the data needed to execute an iteration is used, data does not need to be duplicated. However, unless loop size is small and the grain size is so large that the message passing overhead can be neglected, the scheduling overhead will be prohibitively high to demonstrate the benefit of balancing the workload. We believe that these approaches should not be recommended as general approaches.

Can the Affinity Scheduling scheme [68] use no duplication as its data distribution policy? The fact is that in Affinity Scheduling the data may not need be duplicated or replicated initially. However, when a processor executes iterations in another processor's work queue, the data needed by these iterations has to be sent to the processor on which the iterations are executed. This requires the data to be at least partially duplicated. In addition, the overhead incurred by such a scheme is higher than that of the method described earlier in k-duplication of partial array. This is because, first, the messages are much larger because they contains data; second, the message preparation takes a longer time because it has to pack and unpack data; third, for each of these messages there are two processors spending time on non-computation tasks at the message sending and receiving end. In addition, Affinity Scheduling requests an idle processor to obtain more iterations from the "most loaded" processor. This is an expensive operation on a distributed-memory machine and requires all the processors to participate.

5.4 Multilevel Scheduling

In a straightforward implementations of self-scheduling schemes on a distributed-memory machine, task distribution is centralized. Unscheduled iterations are grouped into chunks and stored in a queue on a designated scheduling processor. An idle processor sends a request message to the scheduling processor for additional work.

When the number of processors increases, having only one scheduling processor results in sequentialized task assignment to idle processors. Also, frequent request messages sent by the idle processors to the scheduling processor may cause a bottleneck due to the increased traffic.

To solve the sequentialized task assignment problem, more than one processor may have to participate in scheduling. If these processors are spread evenly across the system, then they may also alleviate the problem of bottleneck. This can only be achieved at an increased cost in managing the scheduling. We discuss a method that decentralizes the control by dividing recursively the processors into two or more groups of equal size until each group has only e processors. We call the resulting groups of e processors *leaf groups.* For each leaf group, there is one scheduling processor and $e-1$ working processors. The scheduling processor is responsible for

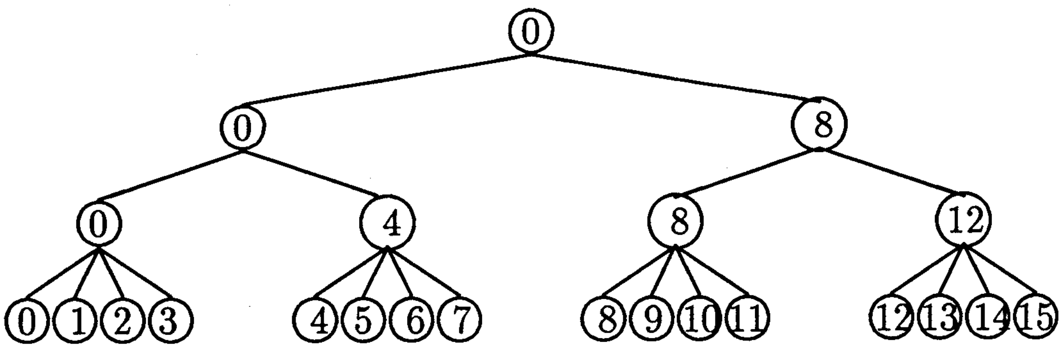

Figure 5.18. Grouping processors for decentralized control

two tasks: one, to assign tasks to the working processors in the same leaf group to keep them busy, and two, to communicate with the scheduling processors of neighboring groups to balance the global workload.

For example, if the target machine is a hypercube of dimension d, it is then divided into $2^d/e$ subcubes of size e (assume e is also power of two). The optimal value for e depends on both the hardware and the problem itself.

If we assume that the underlying architecture is a hypercube, then the division of processors into two groups of equal size can be performed by dividing the processors along a dimension. For example, for two leaf groups, the processors having most significant bit 0 of their binary address form the first group, and the rest form the second group. In general, if $e = 2^l$ then all the processors with the rightmost l digits being 0 are scheduling processors. A scheduling processor's parent processor can be obtained by flipping the right most bit with a value of 1 to 0. Figure 5.18 gives an example of such a grouping for a 4 dimensional hypercube where $e = 4$. A scheduling processor's sibling scheduling processor at level l can be obtained by flipping the l's most significant bit. Figure 5.18 shows that there are four scheduling processors.

By employing a larger number of processors as scheduling processors, less processors are available for computation. This weakness of using multilevel scheduling can be partially relieved by assigning some iterations to the scheduling processors during the static scheduling phase when working processors do not request for work.

Scheduling Protocol

Several different types of communication messages are sent, both by the working processors and the scheduling processors during the dynamic scheduling phase, to ensure that working processors do not idle.

When a working processor, say WP_0, finishes its task and becomes idle, it sends a request message to its parent scheduling processor SP_0 for more iterations. For example, in figure 5.18, processor 5 sends a request to processor 4 when processor 5 becomes idle. If SP_0 has unscheduled iterations, it sends a chunk of iterations to the working processor WP_0. If SP_0 does not have any unscheduled iteration, it tries to find work by communicating with other scheduling processors in the system. In which case SP_0 sends a request message to its sibling scheduling processor SP_1 for additional work. For example, when processor 8 runs out of tasks, it sends a message to processor 12 for additional work.

If SP_1 receives a request from SP_0 for additional work, it shares its unscheduled iterations, if any, with SP_0. However, if $SP1$ itself does not have any work, it sends a message to SP_0 reporting unavailability of work if the $SP1$'s id number is larger than that of SP_0. Upon receiving this message, SP_0 then attempts to find work from the sibling processor at a level above the current level. If $SP1$'s id number is smaller than that of SP_0, then SP_1 finds additional work from the sibling processor at a level above the current level. The same protocol is used at the upper levels of this multi-level scheduling scheme.

If no work is found on the way up the tree until the root is reached, the root processor then sends a message to its children processors indicating that there is no more work to be found in the loop. When a scheduling processor receives such a message it informs all its children working processors that there is no more work in the system.

5.5 Experimental Results

The fact that self-scheduling can be used on a distributed-memory machine to improve performance has been demonstrated in the previous chapter. In this section we use again the image processing algorithm that produces a false-color image to illustrate the feasibility of the data distribution methods proposed.

The problem used in this section is similar to the one used in previous chapter. The difference is that data distribution methods proposed in this chapter are used to distribute data used by dynamically scheduled iterations. We implemented the static scheduling and SSS schemes. For SSS we tested using total replication of partial array as well as k-duplication of partial array to distribute the data. For the static scheduling scheme, no data is duplicated or replicated. The image tested has 512×512 pixels.

Table XII shows the performances of different schemes. For SSS, a value of 0.55 was used for α and the data in self-scheduling phase is replicated in SSS(TRPA) and duplicated in SSS(KDPA), i.e., total replication of partial array policy and k-duplication of partial array policy are used to distribute the data in SSS(TRPA) and SSS(KDPA), respectively. The block size is calculated using SSS. A speedup of 54 is achieved using total replication of partial array on an NCUBE/7 with 64 processors. The improvement in speedup in SSS(TRPA) and SSS(KDPA) come from better utilization of processors. The overhead of k-duplication of partial array of data is higher than that of using total replication of partial array of data. This is expected because, comparing with total replication of partial array, a larger

Table XII. Generation of a False-Color Image

	Execution time (sec) & speedup in () Sequential execution time 81.707(sec)				
Schemes	4	8	16	32	64
SSS(TRPA)	24.0 (3.4)	11.1 (7.4)	5.5 (14.9)	2.8 (29.5)	1.5 (54.1)
SSS(KDPA)	24.1 (3.39)	11.3 (7.2)	6.0 (13.6)	2.95 (27.7)	1.7 (45.4)
Static	23.9 (3.4)	13.7 (6.0)	8.5 (9.6)	5.0 (16.2)	2.609 (31.3)

scheduling overhead is involved in assigning a chunk to an idle processor when data is partial duplicated. The experiments were conducted on an NCUBE/7 with 64 processors. We assume that the times of loading and distributing the data depends on hardware and are not considered.

5.6 Conclusions

Self-scheduling schemes are used, essentially on shared-memory machines, to schedule parallel loops with variable length iteration execution times not known at compile time. With an increase in the number of processors in a parallel computer, memory tends to be distributed. In this chapter we have studied the problem of implementing the concept of self-scheduling non-uniform parallel loops on a distributed-memory environment.

The general approach introduced in this chapter is an extension of DSSS. In addition to the advantages of using two phases in DSSS, the methods discussed provide a systematic way to implement a given self-scheduling scheme on a distributed-memory machine. We have classified the data distribution methods into four categories based on the amount of data being replicated or duplicated. We also present *k-duplication of partial array*, a method that permits problem size to grow linearly in the number of processors. Using the method discussed in this chapter, a user can expect to solve larger problems efficiently by employing more processors.

We also show that the k-duplication of partial array method of distributing data allows the system to self-schedule parallel loops with much greater data size without significant loss in efficiency. To ease the bottleneck of a single processor as the scheduling processor, we have proposed and implemented a multi-level scheduling scheme for parallel loops.

Chapter 6

INTEGRATING SSS INTO CHARM

6.1 Introduction

In this chapter we present techniques for automatic generation of code for user selected self-scheduling scheme and the code of a data distribution policy that is suitable for the selected scheduling scheme. The techniques are implemented in a machine independent parallel programming environment, namely CHARM [19]. We have developed high-level abstractions for distributing data and abstractions for a variety of self-scheduling schemes on shared as well as distributed-memory machines. These abstractions provide flexibility and machine independence for programs that are easily portable across a variety of parallel computers.

Many parallel languages or environment support some form of parallel loops but few support self-scheduling of parallel loops [39, 88]. Self-scheduling schemes are often used to improve the processor utilization. However, implementing self-scheduling requires extensive coding that is often left to the programmer. Different self-scheduling schemes on a distributed-memory parallel computer may require that data being distributed to processors differently. This makes implementing a self-scheduling scheme even more difficult. In the case when different schemes are used for different loops in the same program, data may need to be redistributed at run time to support different schemes efficiently. In addition, the code for scheduling schemes is often embedded and interspersed with the code for the underlying algorithm. This makes the program more complicated, more difficult to port from one machine to another, and harder to debug.

There are two approaches to deal with the distribution of data to processors. First, a compiler may analyze the program and generate a distribution at compile time for the program [23, 31, 56, 85]. The second approach is that the programmer may provide some abstractions to indicate to the compiler on how the user want the data to be distributed [39, 86, 88]. We have taken the second approach where the programmer provides both a high level data distribution abstractions and parallel loop scheduling abstractions. The compiler then uses these pieces of information to insert code to realize the user's specifications. We provide abstractions to specify how to distribute a data array initially and how to schedule the iterations of a parallel loop. If the data distribution does not result an efficient realization of the user specified scheduling method on a parallel loop, functions are called automatically at run time to redistribute the data. The abstractions are developed in CHARM, a parallel programming language.

Analysis is presented to assist the user in determining which scheme to use in scheduling a parallel loop. Experiment results show that the newly added features greatly increase the usability of CHARM without sacrificing efficiency. Although the studies are conducted using CHARM, the same techniques apply to other parallel programming languages as well.

The rest of the chapter is organized as following. We give an overview of CHARM in Section 6.2 and present the approach of supporting different scheduling schemes and data distribution policies in CHARM in Section 6.3. Experimental results obtained on two applications, namely false-color image and subgraph isomorphism are discussed in Section 6.4. We conclude this chapter in Section 6.5.

6.2 Overview of CHARM

CHARM is a message-driven machine independent parallel programming system [19]. It allows parallel programs to run efficiently on different MIMD systems without any modification to the code. The system can be a shared-memory machine, distributed-memory machine, or a network of workstations. It supports an explicitly message-passing parallel language and helps control the complexity of parallel programs by imposing a separation of concerns between the user program and the system. The programmer is responsible for the static or dynamic creation of tasks or processes and exchanging messages between them. The processes can be allocated to processors statically or CHARM can also assume the responsibility of scheduling the processes dynamically.

Conceptually, CHARM maintains a pool of work consisting of processes and messages for existing processes. The system assigns processes in the pool non-deterministically at run time to processors for execution. The programmer may also specifically assign processes to processors. A message is sent to a process called a *chare*. A chare has several entry points and a message must be sent to one of the entry points. Processing a message involves jumping to the entry point specified in the message and executing the code sequentially. Once a message is processed, it always voluntarily relinquishes the processor, returning control to CHARM. The execution of a message may result in new processes or new messages. These new pieces of work are then put into the work pool for execution.

Although CHARM language provides machine-independent high level abstractions for information sharing it does not provide data partitioning and distribution abstractions or abstractions for scheduling loops. The rest of the chapter describe the abstractions we have developed for CHARM to support the execution of parallel loops and automation of data distribution.

6.3 Abstractions for Data Distribution and Loop Scheduling

We provide two sets of abstractions. One set specifies initial data distribution policies when a global array is declared. The other set specifies methods of assigning iterations to processors. We assume that the scheduling methods all have two phases: a static scheduling phase and a dynamic scheduling phase. As discussed in the earlier chapters, in the static scheduling phase, each processor is assigned an equal amount of work. In dynamic scheduling phase, chunks are determined according to SSS for the sake of discussion.

6.3.1 Initial Data Distribution Abstractions

The initial data distribution abstractions specifies, for a global array, the data type, dimension, the size on each dimension, the distributed dimension, and the initial data distribution policy. The syntax is as follows:

```
    distribute data_type name[size]* [size:d_method][size]*
```

or

```
    distribute data_type name[size]+
```

The keyword `distribute` indicates that the array is a global array and must be distributed according to the programmer's specification by the `d_method` field. `data_type` defines the data type of an array element and it can be of any type legal in CHARM. `name` is an identifier and `size` specifies the number of elements in that dimension. Finally, `d_method` indicates the initial data distribution policy. The `d_method` can be any one of the following policies: `BLOCK`, `CYCLIC`, `TRPA`(α), `KDPA`(α) or `REPLICATE`. In both `BLOCK` and `CYCLIC`, no data is duplicated. TRPA is for total replication of partial array and KDPA is for k-duplication of partial array. The α in `TRPA`(α) and `KDPA`(α) is a number between 0 and 1 and indicates the fraction of the total amount of data distributed with no duplication or replication. We assume that these two data distribution policies are used only if the user plans to schedule the parallel loops in the program using self-scheduling and SSS is used to calculate the chunks. The concept remains the same if other self-scheduling schemes are used to calculate the chunks. The default data distribution policy is `REPLICATE` where all data elements are replicated on all the processors.

Example 1

The declaration

```
    distribute int a[1024:BLOCK];
```

declares an array of 1024 elements with the elements from $i \times \lceil 1024/P \rceil$ to $(i+1) \times \lceil 1024/P \rceil - 1$ resident on processor i where P is the total number of processors.

Example 2

The declaration

```
    distribute int a[1024][1024:BLOCK];
```

declares a two dimensional array of 1024×1024 elements with the elements on column from $i \times \lceil 1024/P \rceil$ to $(i+1) \times \lceil 1024/P \rceil - 1$ resident on processor i. That is, each processor has an array of integers a[1024][$\lceil 1024/P \rceil$] resident on it.

Example 3

The declaration

```
    distribute int a[1024:TRPA(0.75)];
```

declares a one dimensional array of 1024 elements. Elements from $i \times \lceil 1024/P \times 0.75 \rceil$ to $(i+1) \times \lceil 1024/P \times 0.75 \rceil - 1$ resident on processor i only. Elements from

Table XIII. Supported loop scheduling schemes and the corresponding data distribution policies

SCHEDULING_SCHEME	Meaning	Data Distribution
GSS, Factoring, PSS, CSS	Using GSS	Replicate
SSS(α)	Using SSS	TRPA
DSSS(α)	Using DSSS	KDPA
BLOCK	Using static chunk	No duplication

$P \times \lceil 1024/P \times 0.75 \rceil$ to 1023 are replicated on all the processors.

6.3.2 Parallel Loop Scheduling Abstractions

A parallel loop scheduling abstractions specifies, for a `forall` loop, the starting iteration number, ending iterations number, and the loop scheduling scheme. It has the following format:

```
forall(var = starting; var < ending; var++; SCHEME) {

    Loop body
}
```

The keyword `forall` indicates that the loop is a parallel loop and needs to be executed on all the processors. SCHEME, when replaced by a key word listed in the left most column of Table XIII, specifies a scheduling scheme, according to which the iterations of the loop is assigned to processors.

Table XIII also lists the required data distribution method for each of the scheduling schemes. The current implementation of the self-scheduling schemes uses one processor as the scheduling processor on which the main *chare* executes.

```
distribute int a[128:TRPA(0.8)];
chare main{
   ...
entry CharеInit: {
   ...
   bound("num.dat", a);
   }
} /* end of main */
```

Figure 6.19. User's CHARM code for initial data distribution

6.4 Implementation of Abstractions in CHARM

6.4.1 Implementation of Data Distribution Abstractions

The data is initially distributed to processors in the main chare's `DataInit` entry point by calling the function `Bound(array_name, file_name)`, where `file_name` is the name of the file that contains the data for `array_name`. The data is then distributed to the processors according to the data distribution method defined in the declaration of `array_name`.

Figure 6.19 shows a segment of the CHARM code using abstractions for initial data distribution. This segment of code is then translated into the code given in Figure 6.20 which creates a distributed array in a *Branch Office Chare*, called _CK_LP_BOC in Figure 6.20. A *Branch Office Chare* is similar to an ordinary *chare* except it is created on every processor in the initialization stage of a CHARM program's execution. Inside _CK_LP_BOC, we declare a pointer variable that points to the first element of the distributed array.

Once this _CK_LP_BOC chare is created, CHARM executes the function

```
chare main{
    ...
entry DataInit: {
    Create BOC for data distribution;
    }
entry CharеInit: {
    Create chares that read in the data from file;
    }
} /* end of main chare */

BranchOffice _CK_LP_BOC {
int *_ck_usr_data
entry RECEIVING_DATA: {
    allocate memory for _ck_usr_data and/or initialize it using the passed in message
    }
public GetIntDataPtr(ptr) {
    assign ptr with the memory address of data;
    }
}
```

Figure 6.20. The resulting CHARM code for Figure 6.19.

```
chare working {
    ...
    forall (i = 0; i < 128; i++; SSS(0.8))
        A(a[i]);
}
```

Figure 6.21. User's CHARM code for loop scheduling

bound in the main chare's `CharelInit` entry point. In the function a file is randomly accessed to read out values for only one processor at a time and the values are packed into a message and sent to the processor's _CK_LP_BOC chare's RECEIVING_DATA entry point. The pointer variable then points to the first data element. If the data is not read from a file, then memory is allocated for the distributed array and is pointed by the pointer variable. In Figure 6.20, the pointer variable is `_ck_usr_data`. To access the distributed array, a process calls the public function `GetIntDataPtr()` to obtain the address of the array.

6.4.2 Implementation of Scheduling Abstractions

Finger 6.21 shows a `forall` loop. After translation, the loop body becomes a chare with two entry points. The chare `_CK_LP_CHARE` in Figure 6.22 is the place where the loop is actually executed.

The `forall` loop is replaced by a message that is sent to the main chare's `SchedulerInit` entry point, which results in a chare named _CK_LP_SCHEDULER being created on the scheduling processor. The _CK_LP_SCHEDULER chare has different entry points for different scheduling schemes supported. Figure 6.22 shows that chare _CK_LP_SCHEDULER has two entry points: SSS and SSS_REQUEST. The first message to the chare is sent to the SSS entry point where the static

```
chare main{
entry _CK_SchedulerInit: {
    CreateChare(_CK_Scheduler, _CK_Scheduler@SSS, ··· )
    } } /* end of main chare */
chare working {
SendMsg(main@SchedulerInit ··· )
} /* end of user working chare */
chare _CK_Scheduler {
entry SSS: {
    for (i = 1; i < num_proc; i++)
        CreateChare(_CK_LP_CHARE, _CK_LP_CHARE@EXECUTE ··· )
    }
entry SSS_REQUEST: {
    SendMsg(_CK_LP_CHARE@EXECUTE1 ··· )
    }
} /* end of chare Scheduler */
chare _CK_LP_CHARE {
int *_ck_dist_a;
entry EXECUTE: {
    BranchCall(_CK_LP_BOC@IntDataPtr(&_ck_dist_a));
    for (i = msg->lo; i < msg->hi; i++) A(_ck_dist_a[i - msg->lo]);
    SendMsg(_CK_Scheduler@SSS_REQUEST);
}
entry EXECUTE1 {
    for (i = msg->lo; i < msg->hi; i++) A(_ck_dist_a[i - msg->starting]);
    SendMsg(_CK_Scheduler@SSS_REQUEST);
}
} /* end of the _CK_LP_CHARE chare */
```

Figure 6.22. The resulting CHARM code for Figure 6.21.

Table XIV. Loop scheduling schemes and the corresponding data distribution policies supported

Initial policies	Redistributed to
BLOCK	TRPA, KDPA, CYCLIC
CYCLIC	TRPA, KDPA, BLOCK
TRPA	BLOCK
KDPA	BLOCK

scheduling phase of SSS is enforced. This results in the chare _CK_LP_CHARE being created on every processor. In _CK_LP_CHARE, a processor first obtains the memory address of the distributed array on that processor by calling the public function `GetInitDataPtr()` of chare _CK_LP_BOC. The processor then executes the parallel loop on the iterations assigned to it. After finishing the iterations of the first chunk, a processor sends a message to chare _CK_Scheduler's entry point SSS_REQUEST requesting for more iterations. If there are unscheduled iterations, a message is sent by chare _CK_Scheduler to the requesting processor's EXECUTE1 entry point, which is an entry point of chare `_CK_LP_CHARE`

6.4.3 Data Redistribution

Before a parallel loop is scheduled, data used by the loop needs to be distributed according to the scheduling scheme. If the data has been distributed for another loop in the program, the data may need to be redistributed to ensure an efficient execution of the current loop. We currently support data redistribution for schemes listed in Table XIV. The reason that REPLICATE is excluded from the data redistribution is that, first, replicating the data may not be feasible due

```
for every processor do
    1. calculates the amount of memory need by the new data distribution policy;
    2. allocates memory;
    3. divides the elements stored on itself into three groups
        3.a. elements staying on itself
        3.b. elements needed to be sent to other processors
        3.c. elements needed to be broadcasted to all the processors
    4. sends out elements in 3.b. to the corresponding processors
    5. broadcasts elements in 3.c.
    6. reads the message queue and copy the elements into correct
locations allocated in stem 2.
```

Figure 6.23. Algorithm for re-distribution

to the limitation provided by the amount of memory; second, if replication is used then programs usually use small and fixed amount of data. Redistribution between TRPA and KDPA is not supported for similar reasons. Changing the data distribution from BLOCK or CYCLIC to TRPA or KDPA may not always be feasible due to the limitation imposed by the amount of memory needed by the resulting data distribution policies.

The data redistribution is achieved by having each processor executing a segment of code specified in Figure 6.23

6.5 When to Use What

A user often has to make decisions about how to distribute the data. This is affected by two factors. One is the amount of data that needs to be processed, and the other is the scheduling scheme used. For algorithms that process large amount of data, the data has to be distributed using either no duplication or k-duplication of partial array. If a program has more than one parallel loop, the user may choose to use different schemes for different loops. Following are three theorems that can be used in assisting a user select a scheduling scheme(s).

Let L_1 be an uniform parallel loop and L_2 be a non-uniform parallel loop. Let t_1 and t_2 be the total sequential execution time of L_1 and L_2, respectively. Let t_{ST} and t_{SSS} be the execution time when both L_1 and L_2 are scheduled using a static scheduling scheme and SSS, respectively. Let t_{RES} be the execution time when one of the loops, say L_1, is scheduled using static scheduling and the other is scheduled using SSS. That is,

$$\begin{aligned} t_{ST} &= \frac{t_1 + t_2}{P} + t_{imb} && (6.15) \\ t_{SSS} &= \frac{t_1 + t_2}{P-1} + t_s && (6.16) \\ t_{RES} &= \frac{t_1}{P} + \frac{t_2}{P-1} + t_s + t_{red} && (6.17) \end{aligned}$$

where t_{imb} is the delay in static scheduling caused by unbalanced workload; t_s is the scheduling cost for a balanced workload; and, t_{red} is the overhead for redistributing the data.

Theorem 6.1: $t_{ST} > t_{SSS}$ when $t_{imb} > (t_1 + t_2)/(P \times (P-1)) + t_s$

Proof: From Eq.(6.15) and Eq.(6.16) we have

$$\begin{aligned} t_{ST} > t_{SSS} &\Longrightarrow \frac{t_1 + t_2}{P} + t_{imb} > \frac{t_1 + t_2}{P-1} + t_s \\ &\Longrightarrow t_{imb} > t_1 + t_2 \times \left(\frac{1}{P-1} - \frac{1}{P}\right) + t_s \\ &\Longrightarrow t_{imb} > \frac{t_1 + t_2}{P \times (P-1)} + t_s \qquad \square \end{aligned}$$

Theorem 6.2: $t_{ST} > t_{RES}$ when $t_{imb} > t_2/(P \times (P-1)) + t_s + t_{red}$

Proof: From Eq.(6.15) and Eq.(6.17) we have

$$\begin{aligned} t_{ST} > t_{RES} &\implies \frac{t_1+t_2}{P} + t_{imb} > \frac{t_1}{P} + \frac{t_2}{P-1} + t_s + t_{red} \\ &\implies t_{imb} + \frac{t_2}{P} > \frac{t_2}{P-1} + t_s + t_{red} \\ &\implies t_{imb} > \frac{t_2}{P \times (P-1)} + t_s + t_{red} \qquad \square \end{aligned}$$

Theorem 6.3: $t_{SSS} > t_{RES}$ when $t_1 > t_{red} \times P \times (P-1)$

Proof: From Eq.(6.16) and Eq.(6.17) we have

$$\begin{aligned} t_{SSS} > t_{RES} &\implies \frac{t_1+t_2}{P-1} + t_s > \frac{t_1}{P} + \frac{t_2}{P-1} + t_s + t_{red} \\ &\implies \frac{t_1}{P-1} > \frac{t_1}{P} + t_{red} \\ &\implies t_1 > t_{red} \times P \times (P-1) \qquad \square \end{aligned}$$

According to Theorem 6.1, self-scheduling should be used when the number of processors is large. Since $t_1 + t_2$ is fixed for a given loop, increasing the number of processors decreases the overhead of using self-scheduling schemes almost quadratically assuming that t_s remains the same. In theorem 2, since the uniform parallel loop L_1 is scheduled statically, it needs not be considered in selecting a particular scheduling scheme. Therefore, the non-uniform parallel loop L_2 determines which scheme to select. Theorem 3 suggests that when t_1 is large, one may consider scheduling L_1 statically.

The above theorems can be used to help the user determining how a given loop should be scheduled. We understand it may be difficult to calculate the values of some parameters such as t_{imb} and t_s. However, since the method presented in this chapter allows a user to schedule a given loop using different schemes by simply changing one parameter, the user can make a good use of the theorems to eliminate some of wrong choices. Again, since a program runs many times in its life span, after several execution of the program, the user may have a better estimation of the parameters.

6.6 Performance

The techniques discussed in the previous sections are implemented in CHARM and tested using a 16 node Intel hypercube iPSC/2 and a 20 processor Sequent Symmetry. We present below the results of two experiments: false-color image and subgraph isomorphism.

6.6.1 False-Color Image

The false-color image tested here is similar to the one discussed in the previous chapters. We show the results of scheduling the loop using static scheduling scheme, SSS(TRPA), and SSS(KDPA). In both SSS(TRPA) and SSS(KDPA), the chunk sizes are calculated using SSS with α being 0.8. The image tested has 512×512 pixels. Note that since one processor is used for scheduling, the potential speedup cannot exceed $P - 1$.

Table XV shows the performance of different schemes running on a Sequent Symmetry. The improvements in speedup by both SSS(TRPA) and SSS(KDPA) come mainly from better processor utilization. The sequential execution time is obtained by running a C program that executes the same algorithm as the CHARM code.

Table XVI shows the results of executing the same code on an Intel hypercube iPSC/2. Clearly, self-scheduling schemes achieve better performance when the number of processors is reasonably large. Both SSS and DSSS perform roughly the same.

Table XVII shows the performance of the same problem except that the problem size is increased to 1024×1024. The sequential time is estimated because the problem size is too large to run on a single processor. Again, self-scheduling schemes enjoy better performance when the number of processors is large.

Table XV. Generation of a False-Color Image on the Sequent Symmetry on 512 × 512 pixels

	Execution time in seconds & speedup in () Sequential execution time 24.780 sec		
Schemes	4	8	16
SSS(TRPA)	8.32 (2.98)	3.69 (6.72)	1.73 (14.32)
SSS(KDPA)	8.28 (2.99)	3.62 (6.85)	1.80 (13.77)
Static	7.63 (3.33)	4.75 (5.20)	3.00 (8.26)

Table XVI. Generation of a False-Color Image on the iPSC/2 on 512 × 512 pixels

	Execution time in seconds & speedup in () Sequential execution time 16.895 sec		
Schemes	4	8	16
SSS(TRPA)	5.703 (2.96)	2.547 (6.63)	1.396 (12.1)
SSS(KDPA)	5.671 (2.98)	2.744 (6.16)	1.563 (10.8)
Static	4.907 (3.44)	2.948 (5.73)	1.865 (9.06)
GSS	5.678 (2.75)	2.915 (5.80)	1.833 (9.22)

Table XVII. Generation of a False-Color Image on the iPSC/2 on 1024 × 1024 pixels

	Execution time (sec) & speedup in () Estimated sequential execution time 65.602(sec)		
Schemes	4	8	16
SSS	21.656 (3.03)	9.40 (6.98)	4.609 (14.23)
DSSS	21.618 (3.03)	9.33 (7.03)	4.454 (14.73)
Static	16.878 (3.89)	8.91 (7.36)	4.943 (13.27)

6.6.2 Subgraph Isomorphism

Two graphs $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$ are *isomorphic* to each other if there is a one to one mapping ϕ between V_a and V_b so that $\forall_{x,y}$ if $(x, y) \in E_a$ then $(\phi(x), \phi(y)) \in E_b$. The subgraph isomorphism problem is to find that if the number of nodes in G_a and G_b are not the same, it may be that the smaller of the two graphs is isomorphic to a subgraph of the larger one. The problem we are solving is to find all the isomorphisms for the two given graphs.

There are many possible algorithms for subgraph isomorphism. We report here the performance of a brute-force approach. The algorithm used is summarized in Figure 6.24. The number of leaf nodes in our test is 262144. Clearly, developing an efficient algorithm for this problem is out of the scope of this chapter.

The execution time of first step, which makes up more than 75% of the total computation time, is roughly the same for all the iterations. The time needed for the second step differs from iteration to iteration because the checking is terminated on the first finding of a non-matched edge. This makes the problem a good candidate for data redistribution.

```
for every leaf node in the search tree do
    1. find the corresponding mapping based on the location of the node
    2. check if the mapping is isomorphic
```

Figure 6.24. Algorithm for isomorphism

Table XVIII. Subgraph isomorphism on the Sequent Symmetry

	Execution time in seconds & speedup in () Sequential execution time 20.480 sec		
Schemes	4	8	16
SSS	7.05 (2.90)	2.95 (6.94)	1.50 (13.65)
Static	5.59 (3.66)	2.90 (7.06)	1.68 (12.19)

We first measured the performance on both a Sequent Symmetry and an Intel hypercube iPSC/2 *without* data redistribution. The results for the Sequent Symmetry are shown in Table XVIII and for iPSC/2 are shown in Table XIX. The results given in the table show that the efficiency of SSS increases while the efficiency of static scheduling decrease. When the number of processor is 16, SSS surpasses static scheduling.

Table XX shows the performance when data is redistributed. The parallel loop is split into two parallel loops. First, static scheduling is used to carry out the first loop. The data is then redistributed to processors. Finally, SSS is employed to schedule the second loop. The results shows that this application does not have enough computation to offset the overhead of data redistribution. However, the

Table XIX. Subgraph isomorphism on the iPSC/2

	Execution time seconds & speedup in () Sequential execution time 22.932 sec		
Schemes	4	8	16
SSS	7.683 (2.98)	3.324 (6.90)	1.59 (14.42)
Static	6.158 (3.72)	3.098 (7.40)	1.68 (13.67)

Table XX. Subgraph isomorphism on the iPSC/2 with data redistribution

	Execution time & difference with that of SSS in Table XIX in () Sequential execution time 22.932 sec		
Schemes	4	8	16
Static + SSS	11.805 (4.122)	7.476 (3.156)	5.601 (3.01)

results show that as the number of processors increases, the cost of redistribution decreases. This is because the larger the number of processors, the smaller the amount of data a processor receives. We also noticed that due to the characteristic of CHARM, each data element is copied twice; once from a sending processor's memory to the message and then from the message to receiving processor's memory. This cost can be eliminated by implementing data redistribution at a lower level. The same code, when running on a Sequent Symmetry, takes much longer to finish mainly due to bus contention.

6.7 Conclusions

In this chapter we have presented techniques supporting automating self-scheduling of parallel loops in a parallel programming language. Although the studies are conducted using CHARM, the same techniques apply to other parallel programming languages as well.

We have developed abstractions for self-scheduling parallel loops and data distribution. These new abstractions have been added to the CHARM language. A CHARM program with these new abstractions is translated to an ordinary CHARM program. Data distribution methods that allow efficient execution of scheduling schemes for parallel loops, both static and dynamic scheduling schemes, are supported. Even when the data distribution does not match with the desired scheduling scheme, the system can detect this difference and automatically redistribute the data. Analysis is presented to help user selects a suitable scheduling scheme. The experimental results indicate that the newly added features greatly increase the usability of CHARM without sacrificing efficiency.

Chapter 7

SELF SCHEDULING UNDER FAULTY PROCESSORS

7.1 Introduction

Most of the self-scheduling schemes assume that the number of processors does not change during the execution of a parallel loop. This chapter introduces methods which tolerate the loss of processors during loop execution. We consider two cases. The first is a hardware failure; the second is when the operating system reassigns processors from one job to another. We refer to the first situation as a *hard fault* and the second as a *soft fault.* An example of a soft fault can be found in the Intel Paragon XP/S system: nodes can be partitioned so there are less computational nodes during day time than during night time. In the case of a soft fault, we assume the processor is reassigned by the operating system only *after* it finishes the currently assigned task.

The main consideration of soft fault is performance since the correctness is not affected. The soft fault is dealt with by taking advantage of the two phase (static and self-scheduling) approach of SSS. We propose using a scheme which is less sensitive to processor faults than SSS during its self-scheduling phase. The second case is handled by adding a third phase to SSS. This phase performs self-scheduling on the iterations of unfinished chunks due to processor failure. Both methods are implemented, and benchmarks are given.

Chou and Abraham [9] discuss load redistribution in distributed systems given failures. They assume that each processor in the system, when it fails, has the capability of buffering jobs for later execution and that only one processor is down at any time. These are not the assumptions in this chapter.

DAWGS (a Distributed Automated Workload balancinG System) [10] is a fault-tolerant, load-balancing system. It guarantees that the job will be run at some point in the future. However, it does not guarantee a minimum response time.

The rest of the chapter is organized as follows. Section 7.2 discusses soft faults, and Section 7.3 discusses hard faults. Section 7.4 concludes the chapter.

7.2 Soft Fault

A soft fault denotes the case of a processor that has been working on a parallel loop being reassigned by the operating system to run some other tasks not part of the parallel loop. We assume the processor finishes all its assigned iterations beforehand. In addition, it is difficult to detect when a soft fault occurs without continually polling each processor. Therefore, in our discussions below, we will assume the value of P, the number of available processors, remains constant when we determine the chunk sizes even though the actual number of available processors may be less.

Soft faults have minimum effect on workload balance for GSS. This is because GSS calculates the i^{th} chunk as N_i/P, where N_i is the number of unscheduled iterations, and both hard and soft faults reduce P. Keeping the value of P constant results in slightly smaller chunk sizes than if P were updated whenever a processor is lost. This leads to a more balanced workload.

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

Figure 7.25. A parallel loop containing branches

For Factoring, a processor reassigned at the i^{th} batch, $i > 1$, results in at least one other processor executing more than one chunk in the $(i+1)^{th}$ batch. That is, one processor has to execute another $N/(2^{i+2}P)$ iterations when it should have executed only $N/(2^{i+3}P)$ iterations. Since the processor executing the extra chunk is the one which finished its chunk before the other processors, executing the extra chunk does not affect performance as badly as it might first appear. For Factoring, the chunk size decreases as the batch number increases; therefore, a soft fault during an early stage of loop execution hurts workload balance more than it does at a later stage.

Figure 7.26 shows an experiment conducted on the parallel loop given in figure 7.25. A similar figure example has been used in Chapter 3. The difference is that, in this experiment the number of soft faults was one.

Given a set M of P processors, $p_0, p_2, \cdots, p_{p-1}$, and a set P_f $(P_f \subset M)$ of f faulty processors, the processor usage is defined as

$$\sum_{p_i \in P_f} (T(p_i)) + \max_{p_j \notin P_f} \{T(p_j)\} \times (|P| - |P_f|),$$

where $T(p_i)$ is the time p_i spent on the parallel loop. That is, processor usage is the sum of the execution times of the faulty processors and the product of the number of non-faulty processors and the execution time of the critical processor. In this definition we assume that the reassigned processors are used by some other task

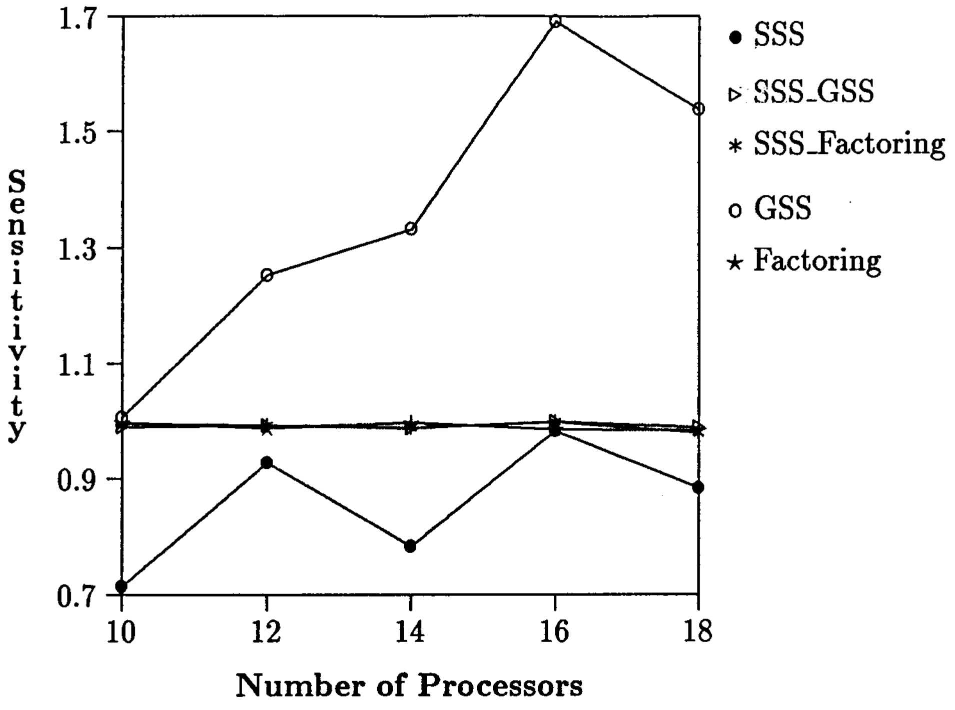

Figure 7.26. Sensitivity of processor usage of SSS, GSS, and Factoring with 1 soft fault

immediately and that the remaining non-faulty processors finishing early are idle while the last processor finishes the loop.

Sensitivity, as shown in figure 7.26, is defined as PU_{nf}/PU_f where PU_{nf} is the processor usage with no faults and PU_f is the processor usage with faults. It can be seen from the figure that both SSS and GSS are sensitive to soft faults.

This can be explained for SSS by the following. The chunk size of the i^{th} batch ($i \geq 0$) is $(N \times (1-\alpha)^i \times \alpha)P$. Therefore, the ratio of the number of iterations between two chunks in consecutive batches is $1/(1-\alpha)$. When a processor drops out after finishing its chunk in the i^{th} batch, there must be at least one processor which fetches a chunk in the $(i+1)^{th}$ batch that is $1/(1-\alpha)$ times larger than it was supposed to fetch. For example, when $\alpha = 0.8$, $1/(1-\alpha) = 5$.

As figure 7.26 shows, when the number of processors increases, the sensitivity of GSS increases. The main reason of this is that (1) as P increases, PU_{nf} increases

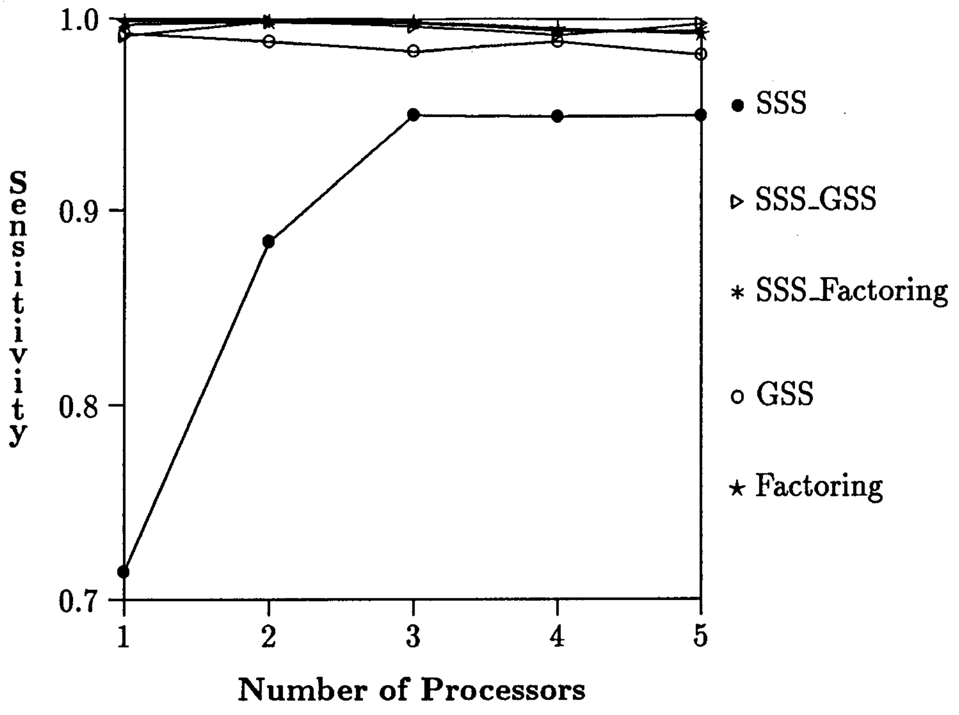

Figure 7.27. **Sensitivity of SSS, GSS, and Factoring to soft fault on 10 processors with up to 5 faulty processors**

and (2) as P increases, PU_f increases too, but at a slower rate than PU_{nf}.

The primary cause of (1) is workload imbalance. When P increases, the first several processors' workloads do not decrease proportionally. The explanation of (2) is that a soft fault actually reduces the ratio of chunk size to number of processors, resulting in a better balanced workload. This ratio is reduced because the value of P is larger than the actual number of processors available.

In addition to testing sensitivity for one soft fault, the loop given in figure 7.25 was tested with up to 5 soft faults. Figure 7.27 gives the results of the experiment with 10 processors. As can be seen, SSS continues to demonstrate sensitivity to soft faults while the other schemes do not. The results obtained by using 12, 14, 16, and 18 processors and 1 to 5 soft faults are very similar to figure 7.27 and have not been included because of space considerations.

To reduce the sensitivity of SSS to soft faults, modifications were made to takes advantage of its two phases. Rather than using the same approach as SSS during the dynamic scheduling phase, we suggest using GSS or Factoring. The SSS using GSS in its dynamic scheduling phase is called SSS_GSS and is described in figure 7.28. In our implementation, we calculate a chunk's boundary before execution to reduce the time spent in the critical section. Chunk boundaries are stored in an array called `chunk_array`. To fetch a chunk, a processor only needs to `fetch&add` the array's index. Factoring can be used as the dynamic scheduling phase of SSS simply by setting α to a number less than or equal to 0.5. This scheme is called SSS_Factoring.

The sensitivity to a soft fault of SSS_GSS and SSS_Factoring is shown in figure 7.26. Clearly, by using either GSS or Factoring in SSS's dynamic scheduling phase, SSS offers better, more stable processor performance.

Another benefit of using GSS or Factoring in SSS's dynamic scheduling phase is less processor cost. Figure 7.29 shows the cost of different scheduling schemes. The cost of a schedule is defined in a similar manner as processor usage except that the number of faulty processors is zero. Figure 7.29 shows that the cost of GSS increases as the number of processors increases. For Factoring, SSS, SSS_GSS, and SSS_Factoring there is no significant change in cost as the number of processors increases. SSS_GSS and SSS_Factoring perform about the same and always offer the best cost performance.

Performance improvement, defined as a lower cost, comes from a better balanced workload as indicated by figure 7.30. The figure plots the standard deviation of processor workload for the corresponding runs of figure 7.29. The workload was calculated by counting DIVERSITY time units for the *then* branch and 1 time unit for the *else* branch. In figure 7.30 the workload of GSS becomes less balanced as the number of processors increases; in figure 7.29 the performance of GSS decreases as

1. Calculate the value for α.
2. Assign each processor $\alpha \times N/P$ iterations statically.
3. Set the global variable `count` to be the first unscheduled iteration's number.
4. When a processor becomes idle, it performs the following
   (a) begins mutual exclusion;
   (b) copy the value of `count` to local variable `i`;
   (c) `t <- max((N - count)/p, 1);`
   (d) `count <- count + t ;`
   (e) end mutual exclusion;
   (f) execute the chunk defined by `i` and `i + t` and repeat step 4 if `i` > `N`;

Figure 7.28. SSS_GSS algorithm

the the number of processor increases. For SSS a better balanced workload in figure 7.30 always results in a better performance in figure 7.29. The only exception is Factoring which has a well balanced workload curve but not a correspondingly good performance. This is because Factoring's well balanced workload comes with great scheduling overhead. SSS_Factoring reduces the scheduling overhead significantly by statically scheduling a major portion of iterations.

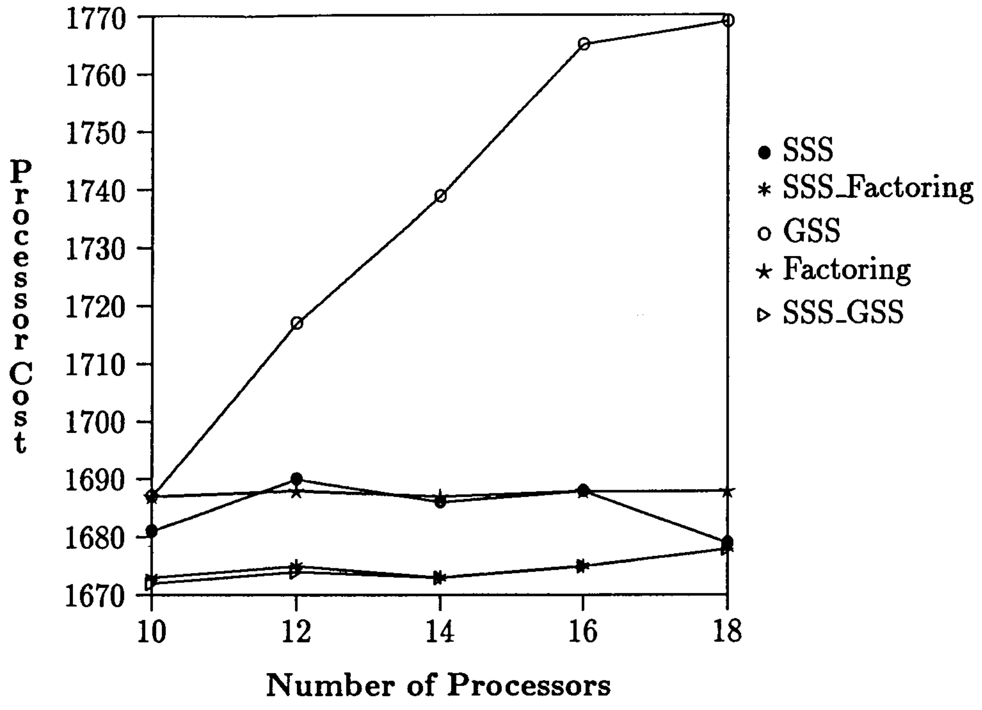

Figure 7.29. Processor cost of different scheduling schemes

7.3 Hard Fault

A hard fault happens when a processor fails physically, e.g. a power failure. The main difference between a soft and hard fault is that in a soft-fault, a faulty processor finishes its currently assigned tasks (in our case the chunk of iterations) before it "drops out"; this is not true if a processor fails physically. In additional, the assumption that the failed processor can set a particular global variable indicating its failure is unrealistic in a hard fault.

A common method of dealing with hardware failure is to issue checkpoints. We propose to do the same. In our approach, a checkpoint is set whenever a processor finishes a chunk. We assume that the computation in a chunk of iterations is performed in a copy-in-copy-out fashion, i.e., the data associated with a chunk is modified only if the entire chunk of iterations is executed. Under this assumption,

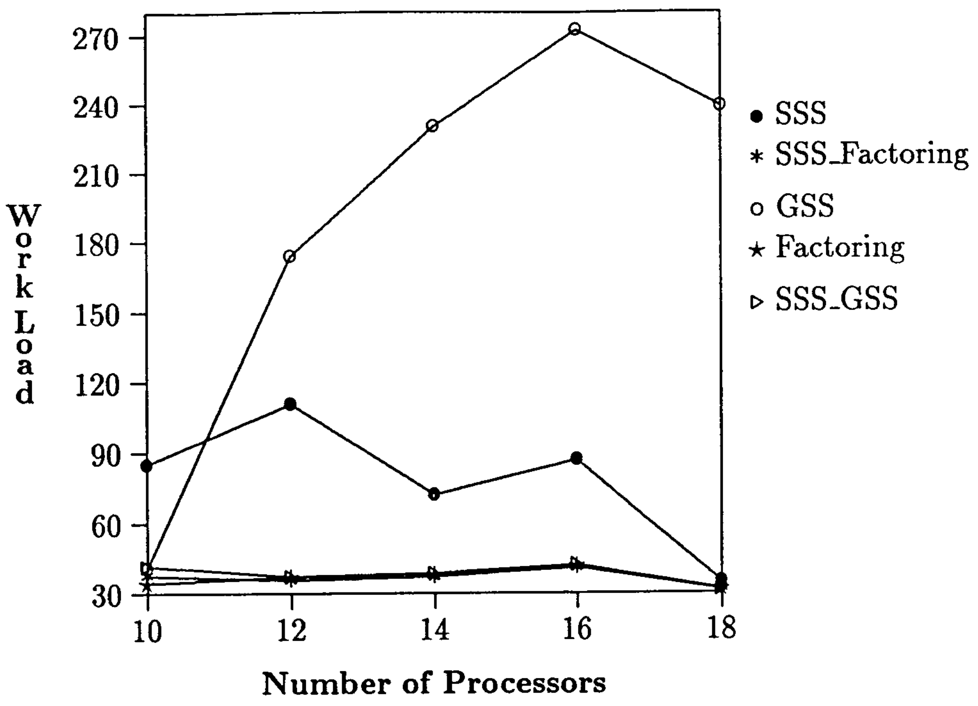

Figure 7.30. Standard deviation in workload for different scheduling schemes

no computation needs to be "undone" when a failure occurs. However, the chunks in which a processor failed must be re-executed.

The above indicates two issues relating to hard faults. One is to re-execute the chunk of iterations, and the other is to balance the workload. Workload balancing can be performed in the same way as soft faults. We propose using Factoring in SSS's dynamic scheduling phase. GSS is not recommended for reasons stated later. The rest of this section discusses the problem of re-executing a chunk when a hard fault occurs.

Assuming that the starting and ending iteration numbers of a chunk are stored in the array `chunk_array`. To fetch a chunk, a processor only needs to copy, using `fetch&add`, the current array index into, say, `temp_index`. The processor then executes the iterations from `chunk_array[temp_index]` to `chunk_array[temp_index + 1] - 1`. To set checkpoints dealing with hard faults, we propose using another array in parallel with `chunk_array` which we call `flag_array`. Before a

```
1. begin mutual exclusion;

2. if flag1 is true then copy-out;

3. end mutual exclusion;
```

Figure 7.31. Pseudocode for a processor to copy-out its results

processor fetches a new chunk, it writes to `flag_array[temp_index]` a value, say `-1`. This assignment to `flag_array[temp_index]` indicates that the chunk from `chunk_array[temp_index]` to `chunk_array[temp_index + 1] - 1` has been executed. The processor can then update its value of `temp_index` by fetching a new chunk.

When an element of `flag_array` has *not* been written back by a processor, it is interpreted as a hard fault. However, it may also mean that the chunk is still being executed. It is important, then, that the chunk size in the dynamic scheduling phase not be too large. For this reason, when hardware failure is a consideration, the Factoring scheduling scheme is recommended for use in SSS's dynamic scheduling phase rather than GSS.

The approach we propose can best be described using the pseudocode of figure 7.31. The dynamic scheduling phase of SSS is now divided into two subphases: the execution subphase and the checking subphase. `Flag1` and `flag2` are global boolean variables which are true during the execution and checking subphases respectively.

The execution subphase begins with the dynamic scheduling phase of SSS. As long as `flag1` is true, a processor writes back the result of executing a chunk of iterations. The first processor, say p_i, which tries to fetch a chunk and discovers that all have been assigned performs the actions described by figure 7.32. This ends the execution subphase and begins the checking subphase.

The first action taken by p_i is to set `flag1` to false. This prevents any processor still working on a chunk from writing back its results so that these iterations can be reassigned to other processors. P_i then finds the first chunk whose corresponding element in `flag_array` is not marked as completed. The iterations of this chunk are redistributed by p_i using the underlying scheduling scheme. P_i then sets `flag2` to true, starting the checking subphase.

A processor working in its execution subphase, finishing after p_i, and finding `flag1` set to false, discards its results and waits on `flag2`. When `flag2` becomes true, p_i has finished the redistribution process, and processors may now begin their checking subphase in a manner similar to figure 7.31. The difference is that now `flag2` is used instead of `flag1`. This procedure can be generalized and the checking subphase performed repeatedly until no faulty processor is found. This would be done in case more than one processor fails.

Figure 7.33 shows the result of simulating the parallel loop of figure 7.25 with one processor hard-faulting. In the simulation a processor fails during the second batch. Figure 7.33 also contains the results of a sequential execution and a non-faulty execution for the purpose of comparison. It can be seen that processor usage increases with the number of processors. This is expected: the larger the number of processors, the higher the overhead incurred by a processor's exclusive access to `flag1`. The increase, however, remains within 2% of processor usage in the sequential case.

A drawback to this approach appears when a processor reaches the end of its execution subphase. At this point, since Factoring is being used, a processor is working on a small chunk. An excessive amount of overhead accrues because of the frequent need for checking `flag1` and because the results from the late finishing

```
1. begin mutual exclusion;
2. if flag1 is true then
    (a) flag1 = false;
    (b) for all elements in flag_array not set by a processor executing the corresponding chunk, redistribute the chunk.
    (c) flag2 = true;
3. end mutual exclusion;
4. while there are chunks
   fetch and execute
```

Figure 7.32. Pseudocode for re-distributing iterations left by a faulty processor

processors are lost and must be re-calculated.

To overcome this, two modifications are made to the procedure above. First, a processor working on a chunk in the last two batches need not check `flag1` before writing back its results. Second, the first processor finished in the checking subphase, p_i, does not redistribute chunks in the last two batches to the task queue; instead, it executes any chunk from these batches whose corresponding entry in `flag_array` is not marked. Using these two modifications may cause some chunks in the last two batches to be executed twice. However, there should be few such chunks, and they should be small. This should not add significantly to the overhead.

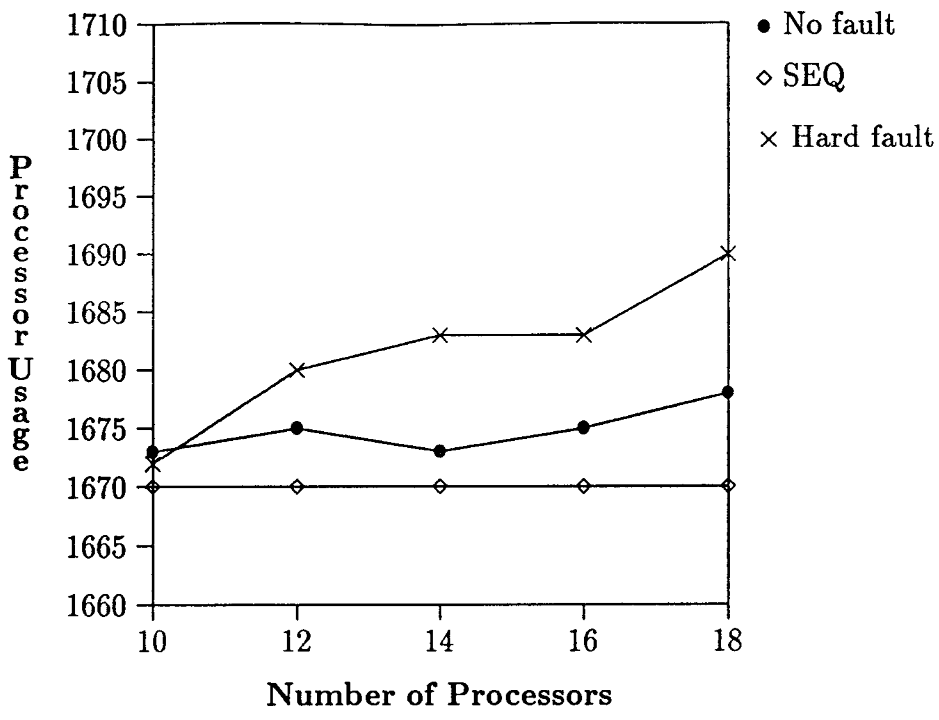

Figure 7.33. Processor usage of SSS_Factoring with 1 processor hard-fault

7.4 Conclusions

In this chapter we have considered the problem of scheduling a parallel loop in the presence of processor faults. We have defined two types of faults: a hard fault results when a processor fails physically, and a soft fault results when a processor is reassigned by the operating system to another task. This chapter presents a SSS with a modified dynamic scheduling phase to tolerate processor loss.

Maintaining a high efficiency is the main concern for soft faults. To tolerate soft faults we replaced the dynamic phase of SSS with either GSS or Factoring. Our experiments suggest that both SSS_GSS and SSS_Factoring offer lower sensitivity to soft faults and greater workload balance than either SSS, GSS or Factoring alone.

To deal with hard faults, we split the dynamic phase of SSS into two subphases, an execution subphase and a checking subphase. Both subphases continue to use either Factoring as the underlying scheduling algorithm; however, together they allow us to implement a method to mark completed chunks. In this way, SSS is able to tolerate hard faults. Our experiments showed that the loss of processor usage due to a hard fault is within 2% of processor usage in the sequential case.

Although our present results come from a small test suite, we believe our method will scale well. From our analysis, however, we conclude that SSS along with the modifications described in this chapter can be used to schedule a non-uniform parallel loop given the presence of processor faults. These scheduling schemes offer performance comparable to SSS but tolerate both hard and soft faults.

Chapter 8

CONCLUSIONS AND FUTURE WORK

The problem studied in this research is how to increase the performance of scientific applications containing parallel loops. Loops in such applications are a rich source of parallelism. This problem is, to a certain extent, what parallel processing is all about. This study seeks its goal by devising methods to achieve high processor utilization with low cost.

We demonstrated a technique of combining a static scheduling scheme with a dynamic scheduling scheme. This combination of the schemes reduces the scheduling overhead while achieving a balanced workload, makes data distribution easier, makes it easier to employ other well known scheduling schemes to utilize their advantages, and increases the affinity of iterations to processors. This combination also simplifies self-scheduling of a parallel loop on a distributed-memory machine and dramatically increases the size of the problems solvable on such architectures.

We also developed methods to duplicate data on a number of processors. This method eliminates data movement during computation, thus reducing the communication cost and increasing the size of problems solvable. We evolved a systematic approach to implement a given self-scheduling scheme on a distributed-memory computer. We also studied using of a multilevel scheduling method to self-schedule parallel loops on a distributed-memory machine with large number of processors.

We introduced the use of abstractions to incorporate self-scheduling methods and data distribution methods in parallel programming environments. The abstractions were implemented using CHARM, a real parallel programming environment. Methods were developed to tolerate processor faults caused by both physical failure and reassignment of processors by the operating system during the execution of a parallel loop.

The techniques introduced in this dissertation have been tested using simulations and real applications from different fields. Good results have been obtained on both shared-memory and distributed-memory parallel computers.

The following are some interesting problems related to the problems discussed in this dissertation. The first one is how to self-schedule and perform data distribution on a network of workstations. Differing from a processor in a homogeneous parallel computer, a workstation in a network, compared with other workstations in the network, may not have the same configuration of amount of memory, amount of storage (hard disk space), CPU speed, and number of users using the workstation. Since the workload of a workstation changes dynamically, some forms of dynamic scheduling must be used to assign iterations to a lightly loaded workstation. Again, we would not assign an iteration to a processor that does not store the data needed by the iteration. It would be interesting to modify the techniques discussed in this dissertation to develop self-scheduling schemes suitable for scheduling parallel loops on networks of computers.

An advantage of SSS is its utilization of the information regarding a parallel loop. This information includes the minimum and maximum iteration execution times, the mean execution times, etc. We did not elaborate on how to collect this information. One idea is to insert segments of code capable of learning. The insertion can be turned on and off. When turned on, it collects the above mentioned information and, at the end, calculates a suitable value for α and stores it in a file.

Multilevel scheduling allows concurrent assignment of iterations to processors, and this requires a sophisticated policy to distribute data to processors. Another interesting issue is that the tree structure of multilevel scheduling can be embedded into a hypertree. The question is, if an internal node of a hypertree can also assign iterations to its children, can we modify our scheme to take advantage of this?

Overall, we believe that we studied a realistic problem and have achieved significant results. Many applications can benefit from our research, and we are interested in seeing our approach being used on a large application requiring many hours of execution time and running on a large system with thousands of processors.

BIBLIOGRAPHY

[1] Adam T.L., Chandy, K.M., and Dickson, J.R, "A Comparison of List Schedules for Parallel Processing Systems," Communication of ACM, vol. 17, no. 12, Dec. 1974, pp. 685-690.

[2] Aho A. Sethi R, and J.D. Ullman *Compilers, Principles, Techniques, and Tools,* Adison-Wesley Publishing Company, Reading, MA, 1986.

[3] Aho A. and J.D. Ullman *Foundations of Computer Science,* Computer Science Press, New York, 1992.

[4] BBN Advanced Computer Inc., Cambridge, MA., "March 1000 Fortran Compiler Reference," revision 1.0 ed., Nov. 1988.

[5] Beckman C. J. and Polychronopolos C., "The Effect of Barrier Synchronization and Scheduling Overhead on Parallel Loops," 1989 International Conference on Parallel Processing, vol. II, 1989, pp. 200-204.

[6] Ben-Asher Y., Cohen A., Schuster A., and Sibeyn J.F, "The Impact of Task-Length Parameters on the Performance of the Random Load-Balancing Algorithm," in the Proceedings of IPPS92, 1992, pp. 82-85.

[7] Bertsekas D.P., Özveren C, Stamoulis G.D., and Tsitsiklis J.N., "Optimal Communication Algorithms for Hypercubes," Journal of Parallel and Distributed Computing 11, 1991, pp. 263-275.

[8] Chen W.K. and Gehringer E.F., "A Graph-Oriented Mapping Strategy for a Hypercube," in the Proceedings of Hypercube, Concurrent Computer and Applications, vol I,, 1988, pp. 200-209.

[9] Chou T.C.K and Abraham J.A. "Load Redistribution Under Failure in Distributed Systems" IEEE Trans. on Computers, vol. C-32, no. 9, Sept. 1983, pp. 799-808.

[10] Clark H., "DAWGS-A Distributed Compute Server Utilizing Idle Workstations," Journal of Parallel and Distributed Computing 14, 1992, pp. 175-186.

[11] Coffman E.G., Jr and Graham R.L., "Optimal Scheduling on Two Processor System," Acta Infromatica, vol. 1, no. 3, 1972, pp. 200-213.

[12] Coffman E.G. (ed), "Computer and Job-shop Scheduling Theory," John Wiley and Sons, New York, 1976.

[13] Crovella M., Das P, Dubnicki C, and LeBlanc T.J. Markatos E.P., "Multiprogramming on Multiprocessors," The University of Rochestor, Computer Science Department, TR 380, 1991.

[14] Crowl L.A. and LeBlanc T.J., "Control Abstraction in Parallel Programming Languages," in the Proceedings of the 1992 International Conference on Computer Languages, Apr. 1992, pp. 44-53. 1992.

[15] Cytron R., "Doacross: Beyond Vectorization for multiprocessors," in the Proceeding of 1986 International Conference on Parallel Processing, Aug. 1986, pp. 836-844.

[16] Eager D.L. and Zahorjan J., "Adaptive Guided Self-Scheduling," Department of Computer Science and Engineering, University of Washington, Technical report 92-01-01, January 1992.

[17] El-Rewini H., Lewis T.G. and Ali H., *Task Scheduling in Parallel and Distributed Systems* Prentice-Hall, New York, 1993.

[18] Fang Z., Tang P., Yew P., and Zhu C., "Dynamic Processor Self-Scheduling for General Parallel Nested Loops," IEEE Transactions on Computers, vol. 39, no. 7, July 1990, pp. 919-929.

[19] Fenton W, Ramkumar B., Saletore V.A., Sinha A.B., and Kale L.V., "Supporting Machine Independent Programming on Diverse Parallel Architectures," in the Proceedings of 1991 International Conference on Parallel Processing, 1991, pp. 193-201.

[20] Fisher J.A. "The VLIW machine: a multiprocessor for compiling scientific code," IEEE Transactions on Computers, vol. 17, no. 7, July 1984, pp. 45-53.

[21] Flynn L.E. and Hummel S.F. "Scheduling Variable Length Parallel Subtasks," IBM Research Report RC15492, Feb. 1990.

[22] Fowler R. and Kontothanassis L., "Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing," The University of Rochester, Computer Science Department, TR 411, June 1992.

[23] Fox G.C, and Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C., and Wu M., "Fortran D Language Specification," Dept. of Computer Science, Rice University, TR90-141, Dec., 1990.

[24] Fox G.C. et al., *Solving Problems on Concurrent Processors,* Prentice-Hall, Englle Cliffs, N.J., 1988.

[25] Garey M.R., and Johnson D.S., *Computer and Intractability - A guide to the Theory of NP-Completeness,* Freeman and Company, New York, 1979.

[26] Gibbons A. and Rytter W., *Efficient Parallel Algorithms,* Cambridge University Press, Cambridge, 1989.

[27] Gottlieb A., Grishman R., Kruskal C.P., McAuliffe K.P., Rudolph L., and Snir M., "The NYU ultracomputer – Designing an MIMD shared-memory parallel computer," IEEE Transactions on Computers, vol. C-32, no. 2, pp. 175–189, Feb. 1983.

[28] Graham R.L., "Bounds on Multiprocessor Scheduling Anomalies and Related Packing Algorithms," in the Proceedings of Spring Joint Computer Conference, 1972, pp.205-217.

[29] Grunwald D.C., Nazief B.A.A., and Reed D.A., "Empirical Comparison of Heuristic Load Distribution in Point-to-Point Multicomputer Networks," in the Proceedings of The Fifth Distributed Memory Computing Conference, April 1990, pp. 984-993.

[30] Gupta M. and Banerjee P., "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 2, March 1992, pp. 179-193.

[31] Gupta R. "Synchronization and Communication Costs of Loop Partitioning on Shared- Memory Multiprocessor Systems," 1989 International Conference on Parallel Processing, vol. II, 1989, pp. 23-30.

[32] Hatcher P.J. and Quinn M.J., *Data-Parallel Programming on MIMD Computers,* The MIT Press, Cambridge, MA, 1991.

[33] Hu T.C., "Parallel sequencing and assembly line problem," Oper. Res., vol 9, Nov. 1961, pp. 841-848.

[34] Kaufman M.C., "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem" IEEE Transactions on Computers, vol. c-23, 11, Nov. 74, pp. 1169-1174.

[35] Kasahara H. and Narita S., "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Transactions on Computers, vol. c-33, no. 11, Nov. 84, pp. 1023-1029.

[36] Kishor S.T., *Probability and Statics with Reliability, Queuing, and Computer Science Applications,* Prentice-Hall, Englewood, NJ, 1982.

[37] Kimura K. and Ichuyoshi N., "Probabilistic Analysis of the Optimal Efficiency of the Multi-Level Dynamic Load Balancing Scheme," in the Proceedings of The Sixth Distributed Memory Computing Conference, 1991, pp. 145-152.

[38] Koelbel C. and Mehrotra P., "Compiling Global Name-Space Parallel Loops for Distributed Execution," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 4, October 1991, pp. 440-451.

[39] Kruskal C.P., Weiss A., "Allocating Independent Subtasks on Parallel Processors," IEEE Transactions Software Engineering, vol. SE-11, no. 10, October 1985 pp. 1001-1016.

[40] Hatcher P.J., Quinn M.J., Lapadula A.J., Seevers B.K, Anderson R.J, and Jones R.R, "Data-Parallel Programming on MIMD Computers, " in IEEE Transactions on parallel and distributed systems, vol 3. no. 2 , pp. 377–383, July 1991.

[41] Hummel S.F., Schonberg E., and Flynn L.E., "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," Supercomputing 91, 1991, pp. 610-619.

[42] Hummel S.F., Schonberg E., and Flynn L.E., "Factoring: A Method for Scheduling Parallel Loops," Communications of ACM, vol. 35, no.8, Aug. 1992, pp. 90-101.

[43] Kant K., "Introduction to Computer System Performance evaluation," McGraw-Hill Inc., New York, 1992.

[44] King C. and Ni L.M., "Grouping in Nested Loops for Parallel Execution on Multicomputers," 1989 International Conference on Parallel Processing, vol. II, 1989, pp. 31-38.

[45] Kohler W.H., "A preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," IEEE Transaction on Computer, vol. C-15, no. 12, Dec. 1975, pp. 1235-1238.

[46] Krone M. "Heuristic Programming applied to scheduling models," in the Proceedings of Fifth Anneal Conference on Information Science and Systems, Department of EE, Princeton University, 1971, pp. 841-848.

[47] Kruskal C. and A. Weiss, "Allocating independence subtasks on parallel processor," IEEE Transaction on Software Engineering, vol. SE-11, no. 10, October 1985, pp. 1001-1015.

[48] Kuck D.J. et al, "The Effect of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," in the Proceedings of 1984 International Conference in Parallel Processing, Aug. 1984, pp. 129-138.

[49] Lam B.Y., Saletore V.A., and Liu J., "Analysis of SSS Scheme," Presented at Permian Basin Supercomputing Conference, Permian Basin, Tx, March 13-15, 1992.

[50] Lam Y.B., Saletore V.A., and Liu J., "Conjugate Gradient Method Using CHARM on Parallel Computers," to appear in the Proceedings of PARALLEL CFD93.

[51] Lee S.Y, and Aggarwal J.K., "A Mapping Strategy for Parallel Processing," IEEE Transaction on Computer, vol. C-36, no. 4, April 1987, pp. 433-442.

[52] Lewis T., Curry R. and Liu J., "Data Parallel Program Design," in the Proceedings of the First International Conference of ACPC, 1991, pp. 37-53.

[53] Lewis T.G. and El-Rewini H., *Introduction to Parallel Computing*, Prentice-Hall, New York, 1992.

[54] Li G. and Wah B.W., "The design of optimal systolic arrays," IEEE Transactions on Computers, Jan 1985, pp. 66-77.

[55] Li J. and Chen M., "Compiling Communication-Efficient Programs for Massively Parallel Machines," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 3, July 1991, pp. 361-377.

[56] Liu J, Saletore V.A, and Lewis T.G., "Scheduling Parallel Loops Containing If-Then-Else statements," in the Proceedings of ISMM Fifth International Conference on Parallel and Distributed Computing and Systems, Pittsburgh, Pennsylvania, October 1-3, 1992, pp. 83-88.

[57] Liu J. and Saletore V.A., "Self-Scheduling on Distributed-Memory Machines," to appear in the Proceedings of Supercomputing 93, Portland, OR, Nov., 1993.

[58] Liu J., Saletore V.A., and Lewis T.G., "Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors," submitted to International Journal of Parallel Programming, 1993.

[59] Liu J., Saletore V.A., and Lam B., "Partitioning of Arrays for High Performance," to appear in Proceedings of Sixth International Conference on Parallel and Distributed Computing Systems, 1993.

[60] Liu J., Marsaglia J., Broeg B, and Saletore V.A., "Scheduling Parallel Loops Under Faulty Processors," to appear in the Proceedings of Sixth International Conference on Parallel and Distributed Computing Systems, 1993.

[61] Lo S.P. and Gligor, V.D., "Properties of Multiprocessor Scheduling Algorithms," in the Proceedings of ICPP87, pp. 867-870.

[62] Lo V.M., "Heuristic Algorithm for Tasks Assignment in Distributed Systems," IEEE Transactions on Computers, vol. 37, no. 11, Nov. 1988, pp. 1384-1397.

[63] Lu L.C. and Chen M., "New Loop Transformation techniques for Massive Parallelism," YALEU/DCS/TR-833, October, 1990.

[64] Markatos E.P., Crovella M., Das P, Dubnicki C, and LeBlanc T.J., "The Effects of Multiprogramming on Barrier Sychronization," The University of Rochestor, Computer Science Department, TR 380, 1991.

[65] Markatos E.P. and LeBlanc T.J., "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors," The University of Rochestor, Computer Science Department, TR 399, 1991.

[66] Markatos E.P. and LeBlanc T.J., "Using Processor Affinity in Loop Scheduling on Shared-Memory," The University of Rochestor, Computer Science Department, TR 410, 1992.

[67] Moldovan, Dan I., *Parallel Processing – From Application to Systems,* San Mateo, California: Morgan Kaufmann Publishers, 1993 .

[68] Miranker, W.L., and Winkler, A., "Spacetime Representations of Computational Structures," Computing, pp. 93-114, 1984.

[69] Misicano A. J., "Efficient Dynamic Scheduling of Medium-Grained Tasks for General Purposing Parallel Processing," in the Proceedings of ICPP88, vol. 1, 1988, pp. 166-175.

[70] Muntz R.R. and Coffman E.G., Jr "Optimal Preemptive Scheduling on Two Processor systems," IEEE Transactions on Computers, vol. c-18, no. 11, 1969, pp. 1014-1020.

[71] Ni L.M., Xu C.W., and Gendreau T.B., "A Distributed Drafting Algorithm for Load Balancing," IEEE Transactions on Software engineering, vol. SE-11, no. 10, October 1985, pp. 1153-1161.

[72] Ni L.M. and Wu C. E., "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," IEEE Transactions on Software Engineering, vol. 15, no. 3, March 1989, pp. 327-334.

[73] Nicol D.M. and Saltz J.H., "An Analysis of Scatter Decomposition," IEEE Transactions on Computers, vol. 39, no. 11, Nov. 1990, pp. 1337-1345.

[74] Papadimitiou C. and Yannakakis M., "Scheduling Interval-ordered Tasks," SIAM Journal of Computing , vol. 8, 1979, pp. 405-409.

[75] Parberry I., "Parallel Complexity Theory," Pitman, London, John Wiley and Sons, NewYork, Toronto, 1987.

[76] Pfister G.F., Brantley W.C., George D.A., Harvey S.L., Kleinfelder W.J., McAuliffe K.P., Melton E.A., Morton V.A, and Weiss J., "An Introduction to The IBM Research Parallel Processor Prototype (RP3)," in Experimental Compution Architechtures, Ed. J.J. Dongarra, New York: Horth Holland, 1987.

[77] Polychronopoulos C. and Kuck D.J., "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," IEEE Transactions on Computers, vol. 36, no. 12, Dec. 1987, pp. 1425-1439.

[78] Polychronopoulos C., "Compiler Optimization for Enhancing Parallelism and Their Impact on Architecture Design," IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 991-1004.

[79] Polychronopoulos C., "Toward Auto-Scheduling Compilers," TR, Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1988.

[80] Press W.H., *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, New York, 1988.

[81] Quinn M.J., "Designing Efficient Algorithms for Parallel Computers," McGraw-Hill Book Company, New York, 1987

[82] Ramamoorthy C.V., Chandy K.M, and Gonzalez M.J. Jr. "Optimal Scheduling Strategies in a Multi-processor System," IEEE Transactions on Computer vol. C-21, no. 2, Feb. 1972, pp. 137-146.

[83] Ramanujam J. and Sadayppan P. "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 4, October 1991, pp. 472-482.

[84] Rosing M, Schnabel R.B, and Weaver R.P., "Dino: Summary and Examples," in the Proceedings of Third Conference on Hypercube Con current Computer, 1988, pp. 312-316.

[85] Robinson J.T., "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms," IEEE Transaction on Software Engineering, vol. SE-5, no. 1, January 1979, pp. 24-31.

[86] Rosing M and Weaver R.P., "Mapping Data to Processors in Distributed Memory Computations," in the Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, SC, 1990.

[87] Rudolph D.C. and Polychronopoulos C., " An Efficient Message-Passing Scheduler Based on Guided Self scheduling," in the Proceedings of the 3rd International Conference of Supercomputing, 1989, pp. 50-60.

[88] Saletore V.A., "A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks," in the Proceedings of The Fifth Distributed Memory Computing Conference, April 1990, pp. 994-999.

[89] Saletore V.A., Liu, J., and Lam B.Y., "Scheduling Non-uniform Parallel Loops on Distributed Memory Machines," in the Proceedings of Hawaii International Conference on System Sciences," vol. II, pp. 516-525, January 1993.

[90] Saletore V.A. and Liu J., "Combining Self-scheduling and Data-Distribution Schemes for Parallel Computations," submitted to Eighth International Parallel Processing Symposium 1993.

[91] Saletore V.A. and Liu J., " Scheduling and Data Distribution for Non-uniform Parallel Loops on Distributed Memory Parallel Machines," submitted to IEEE Transactions on Parallel and Distributed Systems, 1993.

[92] Saltz J., Crowley K., Mirchandaney R., and Berryman H., "Run-Time Scheduling and Execution of Loops on Message Passing Machines," Journal of Parallel and Distributed Computing no. 8, 1990, pp. 303-312.

[93] Saltz J., Mirchandaney R., and Crowley K., "Run-Time Parallelization and Scheduling of Loops," IEEE Transactions on Computers, vol. 40, no. 5, May 1991, pp. 603-612.

[94] Sarker V., "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Research Monographs in Parallel and Distributed Computing, London: Pitman, 1989.

[95] Shang W. and Fortes J.A.B., "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," IEEE Transactions on Computers, vol. 40, no. 6, June 1991, pp. 723-742.

[96] Shirazi B. and Wang M., "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," Journal of Parallel and Distributed Computing 10, 1990, pp. 222-232.

[97] Skillicron D.B., "Architecture-Independent Parallel Computation," IEEE Computer, Dec. 1990, pp. 38-50.

[98] Smith Justin R., *The defign and Analysis of Parallel Algorithms,* Oxford, NewYork: Oxford University Press, 1993.

[99] Snyder L. and Socha D., "An Algorithm Producing Balanced Partitionings of Data Arrays," in the Proceedings of DMCC5, 1990, pp. 867-875.

[100] Stallings W, *Computer Origination and Architecture,* MaCmillan Publishing Company, New York, 1990.

[101] Taieb F.Z., Rami G.M., and Kirk R.P., "Dilation Based Bidding Schemes for Dynamic Load Balancing On Distributed Processing Systems," in the Proceedings of Distributed Memory Computing Conference, 1991, pp. 129-136.

[102] Tang P., Yew P., and Zhu C, "Impact of self-scheduling on performance of multiprocessor systems," in the Proceeding of 1988 ACM International Conference on Supercomputing, July 1988, pp. 593-603.

[103] Tang P., Yew P., and Zhu C, "Compiler Techniques for Data Synchronization in Nested Parallel Loops," in the Proceedings of International Supercomputing Conference, 1990, pp. 177-186.

[104] Tel G., *Topics in Distributed Algorithms,* Cambridge University Press, 1991.

[105] Tzen T.H. and Ni L.M., "Trapezoid Self-Scheduling: A practical Scheduling Scheme for Parallel Compilers," IEEE Transactions on parallel and distributed systems, vol. 4, no 1, Jan. 1993, pp. 87-98.

[106] Ulman J.D., "NP-Complete Scheduling Problems," Journal of Computer and System Science 10, 1975, pp. 384-393.

[107] Wang C. and Wang S, "Efficient Processor Assignment Algorithms and Loop Transformations for Executing Nested Parallel Loops on Multiprocessors," IEEE Transactions on parallel and distributed systems, vol. 3, no. 1, January 1992, pp. 71-82.

[108] Willbeek-LeMair M. and Reeves A. P., "A localized Dynamic Load Balancing Strategy for Highly Parallel Systems," Frontier 90, 1990, PP380-383.

[109] Wolf M.E. and Lam M.S., "A Loop Transformation Theory and an Algorithm to Maximize

Parallelism," IEEE Transactions on parallel and distributed systems, vol. 2 no. 4, October 1991, pp. 452-471.

[110] Wolfe M.J., " Massive Parallelism through program Restructuring," Frontier 90, pp. 407-415.

[111] Xu J. and Hwang K., "Heuristic Methods for Dynamic Load Balancing in A Message-Passing Supercomputer," in the Proceedings of Supercomputing, 1990, pp. 888-897

[112] Zima H., Bast H. -J, and Gerndt M., "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization," in Parallel Computing, vol. 6., 1988, pp. 1-18.

[113] Znati T.F., Melhem R.G., and Pruhs K.R., "Dilation based bidding schemes for dynamic load balancing on distributed processing systems," in the Proceedings of The Sixth Distributed Memory Computing Conference, 1991, pp. 129-136.