On Multiprocessor Scheduling*

(or: Many Queues Good, One Queue Bad†)

Kirk L. Johnson
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

3 February 1994

Abstract

Large-scale multiprocessor systems offer the potential of cost-effective high-performance computing, but several key technologies—both hardware and software—are necessary before this potential can be realized. This paper examines the state of the art in one such software component: multiprocessor scheduling. After surveying several important papers and recent work in the field, I argue for a move away from traditional, centralized schemes limited to the scheduling of loop-derived parallelism.

In order to gain insight into the complexity required for good scheduler performance in large systems, the paper analyzes the behavior of a particularly simple distributed scheduling algorithm based on work stealing. This analysis indicates that although this simple algorithm has some encouraging properties, it also has one characteristic that may prove to fundamentally limit its scalability. By clearly identifying this characteristic, this paper provides insight into the kinds of mechanisms required for scheduling algorithms to perform well on large-scale systems.

1 Introduction

Large-scale multiprocessor systems offer the potential of cost-effective high-performance computing, but several key technologies—both hardware and software—are necessary before this potential can be realized. This paper examines the state of the art in one such software component: multiprocessor scheduling.

In general, multiprocessor scheduling addresses the issue of *where* and *when* to execute independent *threads* or *tasks* of one or more applications; scheduling decisions are made with an eye towards optimizing some system performance metric. In the uniprogrammed (dedicated, single-user or -application) systems discussed in this paper, this metric is typically end-to-end application running time; other metrics are possible.

*Area exam paper, February 1994.
†Apologies to George Orwell.

Scheduling can be done statically (at compile time) or dynamically (at run time). Static scheduling is appealing because it can eliminate any overhead due to scheduling that might be incurred at run time. For *regular* applications in which all threads can be identified statically and the execution times of individual threads can be accurately predicted at compile time, static scheduling can yield good performance. For *irregular* applications that don't meet these requirements, static scheduling often yields poor performance (if it can even be employed). Finally, note that some applications that appear to be regular at first glance can behave irregularly at run time due to relatively unpredictable architectural effects (*e.g.,* cache thrashing, interconnection network contention); static scheduling can perform poorly in these situations as well.

Dynamic scheduling mechanisms are required for efficient execution of irregular parallel applications. Such methods can be either *centralized* or *distributed*. In centralized systems, processors contend for exclusive access to a single system-wide scheduling resource (*e.g.,* a task or work queue) in order to make scheduling decisions [6, 7, 9, 14, 17]. Distributed approaches to scheduling typically assume each processor has a local task queue and make scheduling decisions according to some form of load balancing or load sharing algorithm [13, 15, 18]. Because of the limited applicability of static scheduling, this paper focuses on algorithms for dynamic scheduling.

In the literature, discussion of scheduling algorithms is often accompanied by descriptions of compiler techniques for exposing additional parallelism, reducing communication requirements, or otherwise improving application behavior [4, 14, 16]. For the most part, these techniques can be applied whether scheduling is done statically or dynamically, although the merit of specific techniques may depend on the scheduling model. Although this type of compiler technology is probably of fundamental importance, the focus of this paper is scheduling, not compilation, so these techniques are not discussed further.

The rest of this paper is organized as follows. Section 2 surveys algorithms (both centralized and distributed) for scheduling parallel loops and points out some of the limitations of such schemes. Section 3 discusses scheduling algorithms that are able to handle more general forms of parallelism and provides a brief overview of recent work in that area. Section 4 develops an analytical model for and characterizes the performance a particularly simple distributed scheduling algorithm; this characterization leads to the identification of a key aspect of scheduler behavior that is likely to have a large impact on scheduler performance. Finally, Section 5 summarizes the paper and suggests several directions for future research in the field.

2 Loop Scheduling

The majority of the literature on multiprocessor scheduling focuses exclusively on efficient algorithms for scheduling simple parallel loops, *e.g.*:

```
doall i,1,n
    (loop body)
enddo
```

For simple nestings of parallel and serial loops, compiler techniques (loop interchange and coalescing [14]) can often be applied to obtain loops of this form. More complex nestings are discussed in

Section 2.3.

This section surveys several important loop scheduling papers and recent work in the field. The first subsection discusses centralized loop scheduling schemes (which comprise the bulk of published multiprocessor scheduling work), points out the fundamental *scalability* problem of centralized schemes. The second subsection discusses distributed alternatives for loop scheduling. The third and final subsection wraps up the discussion of loop scheduling by pointing out the inability of the loop scheduling paradigm to handle more general (and frequently useful) models of parallelism.

2.1 Centralized Loop Scheduling

In centralized scheduling schemes, processors contend for exclusive access to a single, centralized scheduling resource in order to make scheduling decisions [1, 6, 9, 7, 14, 17]. All accesses to this resource are protected by a critical section. In general, this centralized resource might be a queue where busy processors enqueue newly created tasks and idle processors look for work, but the full generality of a queue is not required for loop scheduling schemes. Instead, these schemes use single counter indicating the number of loop iterations remaining to be scheduled. Idle processors obtain work by entering the critical section, incrementing or decrementing the counter to obtain a *chunk* of work (a range of consecutive loop indices to execute), then leaving the critical section. (Most loop scheduling schemes allow the use of atomic fetch-and-op instructions, if available, to implement exclusive counter accesses more efficiently.)

Centralized loop scheduling schemes of this sort are generically termed *self-scheduling*[1]; various self-scheduling schemes differ from one another primarily in how chunk sizes are selected.

2.1.1 Fixed-Size Chunks

Without any leading adjectives, self-scheduling is usually taken to mean a scheme where each chunk is either a single iteration (denoted SS or SS(1)) or some fixed number of iterations K (denoted SS(K)). In [7], Kruskal and Weiss analyze the behavior of such a scheme. Assuming that all processors start executing the parallel loop simultaneously, that loop iteration execution times are independent identically distributed random variables with mean μ and variance σ^2 (with some restrictions on allowable distributions), and an overhead of h time units for idle processors to schedule each chunk of work (independent of K), they derive the following expression for the expected finishing time $E(T)$ of an n iteration parallel loop running on p processors:

$$E(T) = \frac{n}{p}\mu + \frac{nh}{pK} + \sigma\sqrt{2K \ln p} \tag{1}$$

As the authors point out, each of the three terms in this expression has a simple interpretation. The first term is the time it takes p processors to finish n jobs, assuming no overhead; the second represents the overhead due to chunk scheduling; the third represents the inefficiency due to

[1]Like many interesting and recurring ideas in multiprocessor systems, self-scheduling was apparently first used on Denelcor HEP multiprocessors [2].

uneven processor finishing times. The tradeoff between the contributions of scheduling overhead (proportional to $1/K$) and uneven finishing times (proportional to $\sqrt{K}$) is readily apparent.

To find the value of K that optimizes this tradeoff, and thus minimizes expected finishing time, the authors differentiate the right hand side of Equation (1) with respect to K, set the result equal to zero, and solve for K. Having thus obtained an optimal value of K, however, they then argue that doing so may not be necessary, for choosing $K = n/p$ (assigning each processor one large, equal-sized fraction of the available work) yields acceptable performance.

In order to draw their final conclusion, the authors rely on an assumption that σ/μ—the coefficient of variation of iteration running times—is no more than one. While this assumption may be reasonable for any number of common distributions possible for a positive random variable (*e.g.,* normal, exponential, uniform), it is not hard to conceive of plausible distributions for which σ/μ is substantially larger than one.[2] Beyond this relatively minor nit, the biggest limitation of the model presented in this work is the assumption that processors can access the centralized scheduling resource (chunk counter) without interfering with one another. On this note, the authors acknowledge that their execution time estimates are lower bounds for real systems, but also point out that taking contention effects into consideration would only strengthen their contention that reasonable performance can be obtained by simply choosing $K = n/p$.

2.1.2 Variable Chunk Sizes

At the end of [7], Kruskal and Weiss discuss several possible extensions of their work, observing that "an optimal scheme would presumably have K decrease as the process evolves" and raising the question of how chunk sizes should be chosen in the case that all processors may not start executing the parallel loop simultaneously. In [14], Polychronopoulos and Kuck present *guided self-scheduling* (GSS), a simple extension of self-scheduling that addresses both of these issues.

In this work, the authors assume a model in which a parallel loop is to be executed on p processors, but all p processors may not start simultaneously. Within this model, their goal is to provide a scheme in which all processors terminate at approximately the same time, arguing that "the best possible solution is that which guarantees that all p processors will terminate with at most B units of time difference from each other" (where B is execution time of a single loop iteration). To achieve this goal, they offer GSS, which uses the following rule for selecting x_i, the size of the i-th chunk to be scheduled:

$$x_i = \left\lceil \frac{R_i}{p} \right\rceil \tag{2}$$

where R_i is the number of loop iterations remaining to be scheduled before the i-th chunk size has been selected (thus $R_{i+1} = R_i - x_i$).

Assuming all loop iterations have equal execution times but allowing essentially arbitrary processor starting times, Polychronopoulos and Kuck prove a theorem stating that by using this rule to select chunk sizes, all processors finish executing the parallel loop within B time units of one another. In addition, they claim that their proof continues to be valid if individual loop iteration

[2] For example, see Figure 1 in [9].

execution times are drawn, presumably at random and independently, from some finite number of possible values $\{b_1, b_2, ..., b_k\}$.[3]

Once again assuming equal execution times for all loop iterations, Polychronopoulos and Kuck claim a second theorem, part of which states that GSS "uses the minimum possible number of synchronization points necessary to achieve optimal load balancing" (where a "synchronization point" is an atomic access to the centralized scheduling resource). The proof of this fact is either omitted without comment, cleverly hidden, or non-existent. The statement of the second theorem is followed by a proof that GSS requires $O(p \ln(n/p))$ synchronization points in the worst case, but any argument that this is the minimum possible number appears to be missing.

Finally, the authors describe two possible shortcomings of GSS and suggest heuristic modifications of GSS to address them. The first shortcoming occurs when loop iteration execution times are both non-constant and correlated. For example, consider a loop in which most iterations have small, nearly-equal running times, but all iterations in some consecutive range require substantially longer to complete. For chunks drawn from the range of iterations with longer running times, assigning one p-th of the remaining iterations to each chunk (per Equation (2)) may compromise load balancing guarantees—some processors may end up with substantially more than one p-th of the remaining work. To address this problem, the authors suggest making p artificially large when scheduling such loops; this has the effect of decreasing chunk sizes and thus the likelihood that any chunk ends up with more than one p-th of the remaining work. Of course, as Kruskal and Weiss observed previously, any benefits of improved load balancing due to smaller chunk sizes must be traded off against increased scheduling overhead.

The second shortcoming of GSS discussed by Polychronopoulos and Kuck is that under some situations (*e.g.,* when p is large and B is small), scheduling overhead may be unacceptably large because of contention for the centralized scheduling resource. To address this problem, they suggest a modification of GSS in which all chunks with fewer than k iterations are eliminated; this new scheme is denoted GSS(k). They proceed to show how GSS(2) is obtained by replacing Equation (2) with:

$$x_i = \left\lceil \frac{R_i + p}{p} \right\rceil \tag{3}$$

and taking some care to ensure that $x_i = R_i$ in the final chunk. Without further exposition or example, Polychronopoulos and Kuck suggest that the same process can be applied to derive GSS(k). Presumably this is accomplished by selecting chunk sizes according to:

$$x_i = \left\lceil \frac{R_i + (k-1)p}{p} \right\rceil \tag{4}$$

On the other hand, some researchers (*e.g.,* Eager and Zahorjan [1]) interpret GSS(k) as meaning chunk sizes are selected according to:

$$x_i = \max\left(\left\lceil \frac{R_i}{p} \right\rceil, k \right) \tag{5}$$

[3] One assumes that when this is the case, the theorem must be modified to state that all processors are guaranteed to finish within $\max_i b_i$ time units of one another.

Whatever the correct interpretation of GSS(k) may be, Polychronopoulos and Kuck conclude their discussion of GSS(k) by pointing out that the best value of k is machine and application dependent.

Despite the shortcomings of this work (the apparently (?) missing proof for part of the second theorem; the lack of exposition as to what the authors really mean by GSS(k)), it has proven to be very influential. In fact, GSS has become the *de facto* standard against which all new loop scheduling schemes must be compared. The staying power of GSS can be attributed to three properties: (i) simplicity, (ii) its ability to produce schedules that are significantly better than self-scheduling and often competitive with or better than the static-division-of-work ($K = n/p$) approach advocated by Kruskal and Weiss, and (iii) robustness in the face of uneven processor starting times.

2.1.3 Variations on a Theme

Several researchers have proposed variations that attempt to improve upon GSS, proposing new solutions to the two basic shortcomings pointed out by Polychronopoulos and Kuck (poor behavior for some workloads with correlated iteration execution; excessive synchronization requirements under some conditions). In general, these schemes are all very similar to GSS; they all use decreasing chunk sizes and a single, centralized scheduling resource. In addition, their exact behavior can typically be varied somewhat from that described here using one or more parameters, á la GSS(k). Most differ from one another primarily in exactly how chunk sizes are selected. This section provides a brief overview of some of that work.

Tzen and Ni propose *trapezoid self-scheduling* (TSS) [17]. In TSS, the size of the first chunk is assigned $\frac{n}{2p}$ iterations and each successive chunk is assigned $\frac{n}{8p^2}$ fewer iterations than the previous chunk. By using this scheme, TSS is less likely to perform poorly on correlated workloads than GSS—the largest chunk size assigned by TSS is half the size of the largest chunk assigned by GSS. In addition, TSS schedules at most $4p$ chunks, independent of n, compared to the $O(p \ln(n/p))$ chunks scheduled by GSS, possibly reducing synchronization requirements for large values of n/p. Perhaps more important, however, is the fact that TSS tends to schedule fewer very small chunks towards the end of a parallel loop than GSS does, which may reduce contention for the centralized scheduling resource. In practice, TSS appears to perform about as well as GSS, except in those cases where the first chunk assigned by GSS would contain significantly more than one p-th of the total work in a loop. In these situations, TSS performs significantly better than GSS [10, 17].

Hummel, Schonberg, and Flynn propose *factoring* [6]. In factoring, chunks are assigned in batches of p equal-sized chunks. The total number of iterations per batch is a fixed ratio of those remaining, typically one-half.[4] Since the largest chunks assigned by factoring contain $\frac{n}{2p}$ iterations (like TSS), factoring is less likely to perform poorly on correlated workloads than GSS. Using this scheme, factoring schedules approximately $p \lg n$ chunks. This is typically at least twice as many chunks as GSS would schedule, thus contention for the centralized scheduling resource may be more of a problem for factoring than GSS. When this isn't a problem, factoring appears to perform about as well as GSS; for correlated workloads that are problematic for GSS, factoring appear to work about as well as TSS [6, 10].

[4]When the coefficient of variation σ/μ is known for loop iteration execution times, this ratio can be adjusted to improve performance.

In [9], Lucco suggests yet another variation on GSS: TAPER. Assuming the coefficient of variation σ/μ for loop iteration execution times is known, Lucco uses a probabilistic approach to derive the following rule for selecting chunk sizes:

$$x_i = \left\lceil T_i + \frac{v_\alpha^2}{2} - v_\alpha\sqrt{2T_i + v_\alpha^2/4} \right\rceil \tag{6}$$

where $T_i = R_i/p$ and $v_\alpha = \alpha\sigma/\mu$. α is a parameter controlling the accuracy of the probabilistic estimate; Lucco suggests a value of $\alpha = 1.3$. (Note that this equation yields GSS as a special case when $\sigma = 0$.) Since the σ/μ can be difficult to determine accurately at compile time, Lucco suggests using either an estimate obtained through profiling of previous application runs or a run-time technique that starts with a conservative estimate and refines that estimate by sampling iteration execution times on-the-fly. To handle correlated workloads, Lucco suggests developing a *cost function* that estimates iteration execution times as a function of iteration number, again using profiling or sampling information. Given such a cost function, chunk sizes are scaled such that the expected execution time of the i-th chunk is approximately $x_i\mu$ (where x_i is as given in Equation (6) and μ is the global mean of iteration execution times).

Even though the process of scheduling a chunk in TAPER appears to be relatively expensive, Lucco reports impressive performance results for several applications running under an implementation of TAPER on a 512-processor Ncube. On the other hand, he also includes data indicating that a large fraction of iterations in an important loop nest in one of the applications have execution times in excess of 15 milliseconds. This is enough to mask quite a bit of scheduling overhead; one surmises that the other applications may be equally coarse-grained. In addition, the Ncube results he reports are for a distributed version of TAPER, not a centralized scheme in which the computation suggested in the previous paragraph would have to be done within a single critical section all processors are contending for. The distributed version of TAPER is discussed further in Section 2.2.

Finally, Eager and Zahorjan suggest *Adaptive GSS* (AGSS) [1]. AGSS is identical to GSS except for two modifications. To handle the problem of excessive contention for the centralized scheduling resource, AGSS detects excessive contention on-the-fly (by estimating the number of processors waiting on the critical section) and dynamically increases the chunk size, thus achieving an effect similar to GSS(k) without having to statically determine an optimal value of k. To address the problem of correlated iteration execution times, AGSS provides the option of using a "wrapped" assignment scheme in which the iterations in a chunk are scattered throughout the iteration index space instead of being consecutive. Permuting the index space in this manner tends to decrease any correlation between iteration execution times in a chunk, thus reducing the likelihood that GSS will assign any chunks with substantially more than one p-th of the total work in the loop [12].

2.1.4 Scalability of Centralized Schemes

Centralized implementations of various self-scheduling schemes are appealing in part because they have proven to be sufficient for exploiting loop parallelism on small multiprocessor systems (typically up to a few tens of processors; the exact number depends loop average iteration execution times, total problem size, and the cost of accessing the centralized scheduling resource). Their appeal also derives from their simplicity. In addition to easing the task of implementation, such simplicity affords the ability to reason about these systems and make fairly strong statements about their performance; witness Polychronopoulos and Kuck's proof of GSS load balancing optimality.

Despite this appeal, a fundamental limitation of centralized loop scheduling schemes is their lack of *scalability*. Because scheduling decisions require exclusive access to a single, centralized scheduling resource, this resource can rapidly become a bottleneck as the number of processors is increased; this effect can only be countered by increasing the problem size, thus increasing chunk size and decreasing the rate at which individual processors require exclusive access.

The tradeoff between problem size and synchronization requirements can be captured by the following simple model. Assuming p processors, an average chunk execution time of T time units, and a chunk scheduling bandwidth of b chunks per time unit, the centralized scheduling resource is not a bottleneck as long as the following inequality holds:

$$\frac{p}{T} < b \tag{7}$$

This inequality can be rewritten as:

$$T > \frac{p}{b} \tag{8}$$

Assuming each processor executes at least one chunk, this inequality makes it clear that in order to prevent the centralized scheduling resource from becoming a bottleneck, total problem size must scale as $\Omega(p^2)$. Requiring problem sizes that are four times larger in order to get the same efficiency out of a system with twice as many processors is clearly not a desirable property.

2.2 Distributed Loop Scheduling

Distributed scheduling algorithms are required to avoid bottlenecks due to the saturation of centralized resources. In general, fully distributed scheduling algorithms assume each processor has a local task queue; upon becoming idle, processors initially look for work in this local queue. If there is no work available in the local queue, an idle processor may examine queues on other processors looking for work to *steal*. Some algorithms also provide (or rely exclusively on) mechanisms that allow busy processors to *push* work onto other processors when a load imbalance is detected. Before delving into a discussion of fully distributed loop scheduling algorithms, however, we first discuss a simple distributed extension of centralized loop scheduling that overcomes some of the scalability problems of truly centralized implementations.

2.2.1 Centralized Scheduling Using Distributed Counting

In essence, the basic scheduling process in centralized loop scheduling schemes is one of assigning new (consecutive) chunk indices to idle processors looking for work; scalability problems arise from the use of an inherently non-scalable counting mechanism. To circumvent these problems, one could replace the centralized scheduling resource with a distributed counting mechanism that provides bandwidth which scales with the number of processors. In such a scheme, instead of obtaining exclusive access to a centralized scheduling resource to make scheduling decisions, idle processors would obtain new a chunk index using the distributed counting mechanism. Given a chunk index, the range of loop indices represented by that chunk (or the possibility that execution of the parallel loop has completed) is easily determined with a small amount of computation.

One distributed counting mechanism that could be used in such a scheme is the *software combining tree*, which can provide $O(p/\log p)$ counting bandwidth (p requests in $\Theta(\log p)$ time) [3]. Assuming a chunk scheduling bandwidth of $cp/\log p$ chunks per time unit (c being an arbitrary constant of proportionality), Equation (8) becomes:

$$T > \frac{\log p}{c} \tag{9}$$

implying that total problem size only has to scale as $\Omega(p \log p)$ in order to prevent scheduling from becoming a bottleneck, a significant improvement over the $\Omega(p^2)$ bound obtained for traditional centralized scheduling schemes.

Unfortunately, while distributed counting mechanisms provide scalable counting bandwidth, they often do so at the expense of increased per-operation latency ($\Theta(\log p)$), even when contention is low. Increased counting latency leads increased scheduling overhead, which in turn decreases system efficiency. An ideal counting mechanism would provide low-latency ($O(1)$) counting when contention is low and only cause processors to suffer high-latency ($O(\log p)$) operations when contention is high; Lim describes a *reactive synchronization* algorithm with these characteristics in [8]. Employing such an algorithm would not improve the asymptotic scalability of a loop scheduling scheme based on distributed counting, but it may improve constant factors somewhat.

2.2.2 Fully Distributed Loop Scheduling

This section provides a brief overview of two fully distributed loop algorithms found in the literature. At a high level, the two algorithms are quite similar: both employ per-processor work queues; both begin execution of a parallel loop by dividing the loop into p equal sized pieces and placing one piece in each processor's work queue. In both cases, processors schedule work from their local queue using self-scheduling with decreasing chunk sizes. Beyond this, the algorithms differ somewhat.

In the distributed version of TAPER described by Graham and Lucco [4, 9] (henceforth simply referred to as TAPER), processors are logically connected in a binary tree (some processors act both as leaves and internal nodes of the tree). Each time a processor finishes executing a chunk of loop iterations, it passes a token up the binary tree; internal nodes propagate such tokens towards the root, possibly combining them with other tokens along the way. The node at the root of the tree tracks the rate of progress for each processor. When a heavily-loaded processor appears to be falling behind more lightly-loaded peers, the root initiates the transfer of some of the work remaining in the heavily-loaded processor's queue to that of a lightly-loaded peer. Idle processors do not query other processors looking for work.

In the *affinity scheduling* (AFS) algorithm suggested by Markatos and LeBlanc [10], processors schedule chunks of work from their local queue until it is empty, at which point they switch to a mode in which new chunks of work are obtained by stealing from the processor that has the most work remaining (using essentially the same self-scheduling rule for selecting chunk sizes).[5] Finally, when an idle processor steals a chunk of work from another processor, it immediately begins executing the entire chunk; no work is left on the previously idle processors queue for other peers to steal at some later time.

Interestingly, in addition to providing load balancing, both algorithms also attempt to reduce communication overhead as well. For parallel loops that are executed many times in succession within the body of a sequential loop, both schedulers ensure that the same groups of loop iterations are assigned to the same processors for each execution of the parallel loop[6], increasing the likelihood that the working set of any given iteration will already be in cache (or local memory) on successive passes through the parallel loop. Optimizations of this sort are probably useful in any system; for systems with high communication overheads or limited communication bandwidth, as appears to be the case for the systems studied in these papers, they are of fundamental importance.

Given that both of these algorithms avoid the problem of using a centralized mechanism to schedule every chunk of work, one expects that both of these schemes should scale significantly better than centralized loop scheduling schemes. However, because TAPER requires the root processor to be involved in all load balancing operations, it could become a bottleneck when there are significant load imbalances in the initial division of work among the processors. To address this, Lucco argues that in many applications, successive executions of the loop will have similar work distributions. When this is true, since TAPER initially assigns each processor the same set of iterations executed by that processor in the previous pass, the need for load balancing is likely to be significantly lower after a few passes through the loop.

2.3 More General Models of Parallelism

Beyond scalability issues, the major limitation of all loop scheduling schemes is their inability to support more general models of parallelism. As mentioned previously, standard loop scheduling schemes can only be used with individual parallel loops or simple nestings of parallel and serial loops. More complex situations, such as those in which a parallel loop is executed conditionally within the body of another parallel loop, are troublesome. Consider, for example:

```
doall i,1,n
    (some computation)
    if (test) then
        doall j,1,m
            (inner loop body)
        enddo
    endif
    (more computation)
enddo
```

[5] A policy requiring idle processors to examine the state of all other processors before finding work is sufficient for the small-scale systems discussed in this paper; the authors acknowledge that more efficient schemes would be necessary in large-scale systems.

[6] TAPER actually behaves slightly differently; at the start of a parallel loop, each processor is assigned the set of iterations executed by that processor during the previous pass through the loop.

Simple loop scheduling schemes might be able to handle such scenarios by executing the inner parallel loop serially, but doing so may needlessly limit application parallelism.

In order to handle nested parallelism of this form, several researchers have proposed two-level scheduling schemes [2, 5]. In these schemes, a high-level scheduler is used to manage *control blocks* for active instances of parallel loop nests and assign processors to execute particular loop nests; new control blocks are created dynamically as necessary. Processors assigned to a specific loop use simple, centralized self-scheduling techniques to execute the loop they are assigned to; loop control blocks contain the centralized state required to do so.

Although two-level scheduling schemes address some limitations of simple loop scheduling schemes, they seem somewhat *ad hoc*, subject to other limitations that also plague the simpler schemes. For example, loop scheduling schemes invariably assume that individual loop iterations will be able to run to completion once scheduled. Loops with iterations that might block (perhaps to synchronize with another iteration from the same loop) are, for the most part, strictly verboten. Similarly, loop scheduling schemes are unable to handle parallelism derived from other than parallel loops (*e.g.,* control parallelism expressed in a fork/join or future style). Allowing for such possibilities requires more sophisticated scheduler mechanisms (*e.g.,* the ability to unload blocked threads and reschedule them when they are ready to run again), something wholly beyond the limited world view of most loop scheduling schemes.

3 General Scheduling

This section discusses scheduling algorithms that are capable of handling general parallelism, unlike the simple loop scheduling algorithms discussed in the previous section. For the purposes of this paper, *general parallelism* is taken to mean a scheme with primitives for creating, suspending, rescheduling, and terminating threads. Further, threads are allowed to block (and possibly need to be suspended) at any time. Note that parallel loops can be expressed easily in such a model (*e.g.,* using syntax that desugars a parallel loop into a mechanism that creates the specified threads and then blocks until their completion; this can be done very efficiently using techniques such as lazy task creation [11]).

Given of the possibility that threads may block, scheduling algorithms for general parallelism, henceforth referred to as *general scheduling* algorithms, must operate at the level of individual threads. This is in contrast the loop scheduling algorithms, where a single scheduling decision can, in essence, commit a large block of threads (loop iterations) to execute indivisibly as a single unit.

As was the case for loop scheduling, centralized algorithms for general scheduling do not scale well to large numbers of processors. Thus this section discusses exclusively distributed algorithms for general scheduling; the remainder of the section provides a brief survey of recent work describing practical general scheduling algorithms.

Rudolf, Slivkin-Allalouf, and Upfal [15] describe a simple distributed load balancing algorithm. In their scheme, individual processors periodically select a peer at random and transfer tasks to or from the chosen peer such that the queue lengths on the two processors are equalized. The frequency with which load balancing operations are attempted is inversely proportional to local queue length. The authors prove that for a system that is already balanced, their algorithm preserves that balance. Unfortunately, they provide no indication of how long it takes their algorithm to reach a balanced configuration from initially unbalanced configuration. Without knowing the duration of such transients (and, more importantly, how that duration depends on p), it is difficult to draw any firm conclusions about how well this algorithm scales for use in large systems.

Willebeek-LeMair and Reeves [18] provide experimental results comparing the effectiveness of five simple distributed load balancing algorithms drawn from the literature on a 32-node Intel iPSC/2 hypercube. From their results, they conclude that a simple scheme in which idle (or underloaded) processors steal work from neighboring processors (in the hypercube) performs nearly as well as or better than any of the other algorithms. Unfortunately, they provide no meaningful analysis or argument about the expected scaling behavior of any of algorithms.

In [13], Nussbaum describes XTM, a sophisticated distributed scheduling algorithm in which idle processors looking for work to steal are guided by information maintained in a distributed search tree. By using this tree, if work is available for stealing anywhere, idle processors are able to find it in $O(\log p)$ steps of the searching process. Nussbaum proves two properties of XTM: (i) the expected time required to update the search tree when the state of a processor (*i.e.* whether or not that processor has work available for stealing) changes is constant, independent of the number of processors; and (ii) when the information in the search tree is accurate and communication latency dominates any computation associated with the search process, the time required to find work is no worse than a factor of four more than would be possible if the exact state of the entire machine was known to all processors. In addition, he presents simulation results indicating that XTM performs well, even for systems with thousands of processors and applications with relatively fine-grained parallelism.

4 An Analysis of Random Work Stealing

Distributed general scheduling algorithms offer the promises of improved scalability and broad applicability. Among other things, however, these promises come at the cost of losing the ability to reason and make strong statements about scheduler performance—contrast the proof of GSS load balancing optimality with what researchers have been able to prove about even simple distributed general scheduling algorithms. In addition, although some sophisticated distributed scheduling algorithms appear to perform well (*e.g.,* Nussbaum's XTM), the capabilities and limitations of simpler algorithms remain unclear.

In an attempt to make headway on these problems, this section describes RWS (**R**andom **W**ork **S**tealing), a simple distributed general scheduling algorithm similar to that suggested by Rudolf, Slivkin-Allalouf, and Upfal. It presents an analytical model for the behavior of RWS and uses that model to characterize the performance of RWS. This analysis indicates that except during an application's termination transient, when few processors have work available for stealing, RWS performs very well. In addition, the analysis indicates quite clearly what algorithmic properties are necessary for improved performance during termination.

4.1 Modeling Random Work Stealing

RWS operates as follows. Each of p processors has a local work queue. Idle processors search for work by selecting a peer at random; if any work is available on the peer's queue, the idle processor steals half and starts executing one stolen task, placing the rest on its local work queue. If no work is available on the peer's queue, the idle processor continues the search process, repeatedly choosing peers at random until it finds some work to steal.

In order to formalize the discussion and make an analytic model of RWS tractable, further assume the following:

- Processors are either busy executing useful work (with work available for stealing) or idle (looking for work to steal).
- Processors looking for work to steal do so in a synchronous fashion—in a single *scheduling phase*, each idle processor selects and attempts to steal work from a single peer.
- When a processor steals work, it steals enough to remain busy for the duration of the time period under consideration.
- Only one idle processor is allowed to steal work from any single busy processor during a single scheduling phase. Conflicts between idle processors attempting to steal from the same processor are resolved arbitrarily; one idle processor gets to steal work, all other idle processors continue the search process as if they had found no work during this scheduling phase.

Consider a Markov chain with p states $\{1, 2, ..., p\}$; each state i represents the situation where i processors are busy and $p - i$ processors are idle. Let $P(i, j)$, the transition probability from state i to state j, be the probability that a single scheduling phase that starts with i processors busy and ends with j processors busy.

Because of the assumption that busy processors stay busy for the duration of the time period under consideration, $P(i, j) = 0$ for $i > j$. For $i \leq j$, state transition probabilities can be obtained by breaking the problem into two pieces:

- Let $f(k, l)$ be the probability that exactly k idle processors select a busy peer when there are l busy processors, ignoring momentarily the possibility that some idle processors may conflict.

  Since there are $p - l$ idle processors and each selects a busy peer with probability $l/(p - 1)$ independent of the others, $f(k, l)$ is easily seen to be:

  $$f(k,l) = \binom{p-l}{k} \left(\frac{l}{p-1}\right)^k \left(1 - \frac{l}{p-1}\right)^{p-l-k} \tag{10}$$

- Let $g(r, s, l)$ be the probability that exactly r idle processors successfully steal work given s idle processors select a busy peer and there are l busy processors.

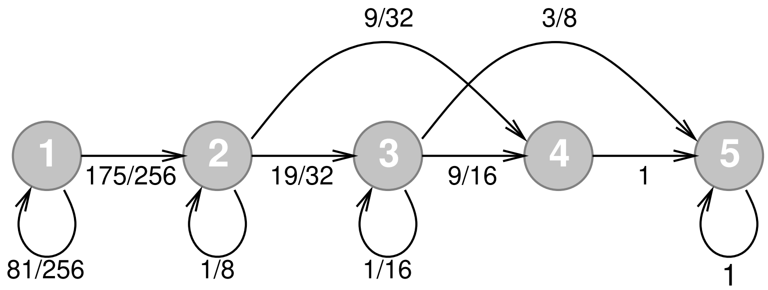

Figure 1: State transition diagram for $p = 5$.

This is equivalent to asking what the probability that exactly r boxes end up non-empty in a problem in which s distinguishable balls are thrown at random into l boxes. It can be shown that this probability is given by:

$$g(r,s,l) = \left(\begin{array}{c} l \\ r \end{array} \right) \sum_{j=0}^{r-1} (-1)^j \left(\begin{array}{c} r \\ j \end{array} \right) (r-j)^s \tag{11}$$

The derivation of this fact is omitted for the sake of brevity.

Individual state transition probabilities can then be obtained according to:

$$P(l, l+q) = \begin{cases} f(0,l) & \text{for } q = 0 \\ \sum_{i \geq q} f(i,l) f(q,i,l) & \text{otherwise} \end{cases} \tag{12}$$

Combining Equations 10, 11, and 12 provides no additional insight, but it does suggest a procedure for computing the exact transition probabilities for specific values of p. Figure 1 shows the state transition diagram and transition probabilities thus obtained for $p = 5$.

Having computed state transition probabilities for a specific value of p, it is straightforward to determine the probability that a particular number of processors are busy after some number of scheduling phases given an initial configuration of busy processors. Given these probabilities, various interesting expectations can be computed directly (*e.g.,* the expected number of busy processors as a function of time or the expected time until some fraction of the processors are busy).

4.2 Analysis

Using this model, we are able to investigate the behavior of RWS in three key regimes: *startup*, where a large pile of work suddenly appears on one processor of an otherwise idle system; *steady state*, where most processors are busy and there is plenty of work for idle processors to steal; and *termination*, where most processors are busy, but only a few of the busy processors have work available for stealing.

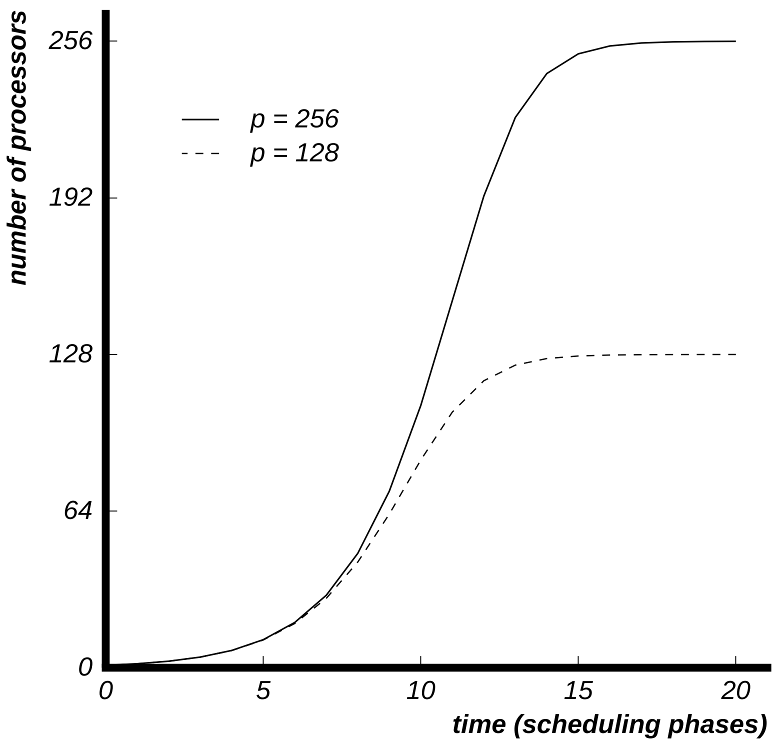

Figure 2: Expected number of busy processors as a function of time, measured in scheduling phases, for $p = 128$ and $p = 256$.

4.2.1 Startup

The startup behavior of RWS can be examined by considering the behavior of the Markov chain given an initial configuration of one busy processor. Assuming this initial configuration, Figure 2 plots the expected number of busy processors as a function of time, measured in scheduling phases, for $p = 128$ and $p = 256$. Note that both curves reach their terminal value (p) after approximately 15 scheduling phases. Curves for other values of p behave similarly.

Figure 3 plots the expected time until all p processors are busy as a function of p. Figure 3 also includes a '99-th pctile' curve indicating the 99-th percentile of the distribution and an 'optimal' curve indicating the best that could possibly be achieved ($\lg p$, given that only one idle processor is allowed to steal work from a busy processor during any scheduling phase). Note that in the data presented here, the expected time until all p processors are busy is never a more than twice than optimal.

Extrapolation of the data presented in Figure 3 leads one to believe that the expected time until all processors are busy will continue to be no more than a factor of two worse than optimal for $p > 256$. This indicates that RWS performs exceptionally well during startup, taking expected time

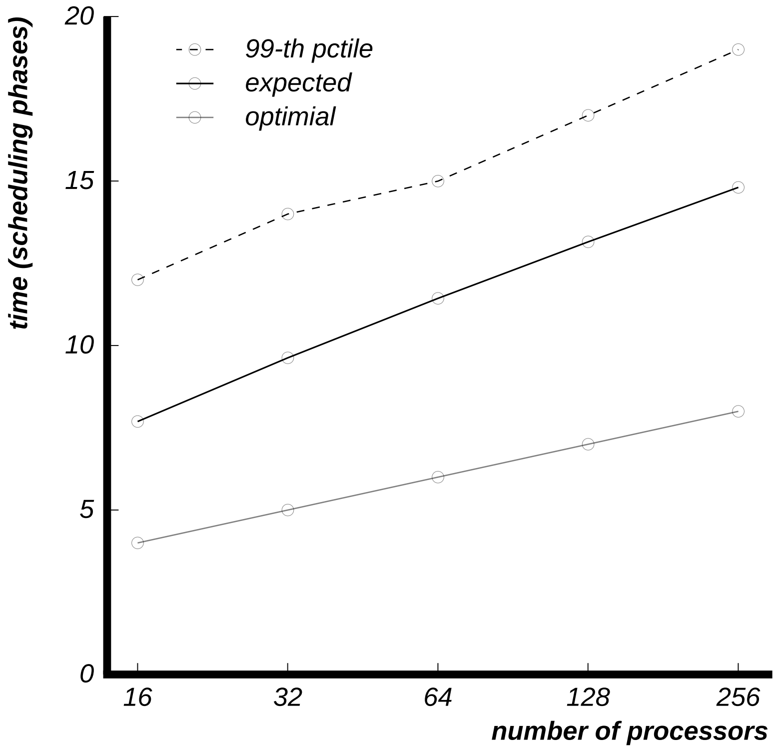

Figure 3: Semi-log plot of expected time until all p processors are busy as a function of p. '99-th pctile' and 'optimal' curves indicate the 99-th percentile of the distribution and optimal values, respectively.

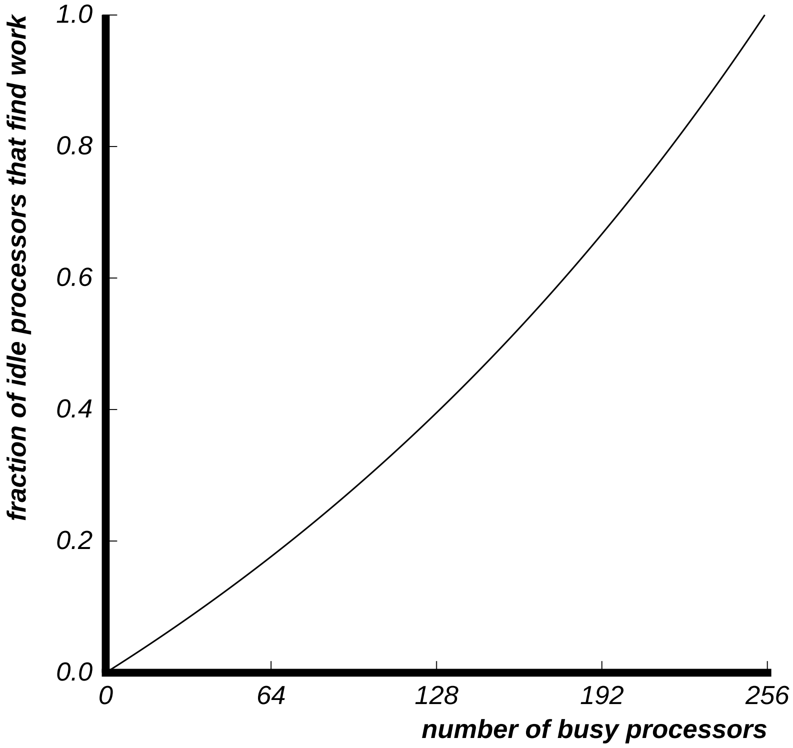

Figure 4: Expected fraction of idle processors that find work in a single scheduling phase as a function of the number of busy processors, $p = 256$.

no worse than twice the optimal value to spread work to all processors.

4.2.2 Steady State

RWS behaves well during steady state for the same reasons it behaves well during the latter stages of startup—when most processors are busy and there is plenty of work to steal, idle processors find work quickly. This can be seen in Figure 4, which plots the expected fraction of idle processors that find work in a single scheduling phase as a function of the number of busy processors for $p = 256$. Note that when most processors are busy, the likelihood that an idle processor finds work during a scheduling phase is high, thus the expected time until an idle processor finds work is low.

4.2.3 Termination

During termination, most processors are busy executing useful work, but few processors have work available for stealing. Modeling the behavior of RWS under these circumstances requires relaxing the assumption that all busy processors have work available for stealing; this has the effect of changing the numerator of the $l/(p-1)$ terms in Equation (10) from the number of busy processors (l) to the number of processors with work available for stealing. Exact state transition probabilities can then be computed according to Equation (12), as before.

Alternately, a simple approximation can be used to gain insight. Assume a system with p processors, n_a of which have work available for stealing and n_i of which are idle, looking for work to steal. When $n_a n_i \ll p$, the probability w that *any* idle processor finds work during a single scheduling phase can be approximated by

$$w \approx \frac{n_a n_i}{p} \tag{13}$$

Thus the expected time T_w until some idle processor finds work is approximately

$$T_w \approx \frac{hp}{n_a n_i} \tag{14}$$

where h is the duration of a single scheduling cycle.

During this time, n_i processors are idle, so $n_i T_w = hp/n_a$ processor-time units are lost to scheduling overhead. Dividing this quantity by p to determine the impact of that overhead on total running time, one finds each such event increases application running time by h/n_a time units, independent of p.

The fact that an event that is likely to be more frequent with increasing p can increase running time by an amount independent of p would seem to bode ill for the scalability of RWS. Let $\varphi(p)$ represent the total contribution of scheduling overhead during termination to application running time as a function of the number of the number of processors, p. In order to prevent this overhead from dominating useful work, total problem size must scale as $\Omega(p\,\varphi(p))$. In the unfortunate circumstance that $\varphi(p)$ is itself $\Omega(p)$, something that does not seem entirely out of the question, this implies that RWS requires problem size to be scaled quadratically as p is increased, a most undesirable property.[7]

4.3 Critique

The analysis presented above indicates that RWS should perform well during startup (expected performance no worse than twice optimal) and steady state (idle processors find work quickly), but may face serious scalability problems due to scheduling overhead during termination. In addition, these results suggest that a key measure of a distributed scheduling algorithm is the time idle processors take to find work during termination conditions. In particular, schemes in which T_w has a linear dependence on p (see Equation (14)) are likely to suffer the same scalability problems as RWS. Algorithms in which T_w has a sub-linear dependence on p should perform better during termination, and thus be able to effectively apply more processors to a given problem size.

In [13], Nussbaum compares the performance of XTM with a simple (round-robin) scheduling algorithm similar to RWS. His results confirm the conclusions presented here: the sophisticated

[7] On the other hand, since the duration of the startup transient is $O(\log p)$, problem size must be scaled as $\Omega(p \log p)$ to maintain efficiency, so if $\varphi(p)$ is $O(\log p)$, scheduling overhead during termination would simply lead to less favorable constant factors, not fundamental increases in problem scaling requirements.

algorithms offer little advantage over the round-robin scheme on small systems, but on large systems (*e.g.,* $p > 1000$) the sophisticated algorithms performed significantly better than the round-robin scheme, often yielding running times a factor of two or more smaller than the simple algorithm.[8]

5 Conclusions

This paper surveys the state of the art in multiprocessor scheduling. It provides an overview of centralized and distributed algorithms for scheduling loop-derived parallelism, pointing out the scalability limitations of centralized schemes and general inapplicability of all loop-specific schemes to applications with other than loop-derived parallelism.

After a brief survey of published algorithms for distributed scheduling of general parallelism, the paper develops an analytical model for the performance of a RWS, a particularly simple distributed scheduling algorithm. Using this model, the paper shows that RWS performs well in many situations, but may perform poorly during termination conditions due to idle processors taking $O(p)$ time to find work to steal; this overhead may prove to fundamentally limit the scalability of RWS. This analysis suggests that if a distributed scheduling algorithm based on work stealing is to perform well on large-scale systems, the time taken by idle processors to find work must have a sub-linear (*e.g.,* logarithmic) dependence on p.

It would be interesting to quantify the scheduling overhead of RWS during termination conditions more accurately, either analytically or through simulation. In particular, it would be interesting to know wither or not this overhead fundamentally limits the scalability of RWS beyond the $\Omega(p \log p)$ problem size scaling already required due the duration of the startup transient. Finally, note that distributed scheduling algorithms like RWS could easily be combined with the kinds of communication optimizations employed by AFS and the distributed implementation of TAPER.

References

[1] Derek L. Eager and John Zahorjan. Adaptive Guided Self-Scheduling. Technical Report 92-01-01, University of Washington, Department of Computer Science and Engineering, December 1992.

[2] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic Processor Self-Scheduling for General Parallel Nested Loops. *IEEE Transactions on Computers*, pages 919–929, July 1990.

[3] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–75, April 1989.

[4] Susan L. Graham, Steven Lucco, and Oliver Sharp. Orchestrating Interactions Among Parallel Computations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 100–111, June 1993.

[8] Note that the point at which sophisticated algorithms begin to offer a significant advantage over simple schemes like RWS is strongly dependent on any number of application and architectural parameters (*e.g.,* total problem size, application parallelism, task size, the average cost of stealing work from a remote processor, etc.); actual mileage may vary.

[5] S. F. Hummel and E. Schonberg. Low-Overhead Scheduling of Nested Parallelism. *IBM Journal of Research and Development*, pages 743–765, September/November 1991.

[6] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, pages 90–101, August 1992.

[7] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, pages 1001–1016, October 1985.

[8] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. Submitted for publication, October 1993.

[9] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 200–211, June 1992.

[10] Evangelos P. Markatos and Thomas J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *Proceedings of Supercomputing '92*, pages 104–113, Nov 1992.

[11] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 264–280, July 1991.

[12] David M. Nicol and Joel H. Saltz. An Analysis of Scatter Decomposition. *IEEE Transactions on Computers*, pages 1337–1345, November 1990.

[13] Daniel Nussbaum. *Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, September 1993.

[14] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, pages 1425–1439, December 1987.

[15] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, July 1991.

[16] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, June 1986.

[17] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, pages 87–98, January 1993.

[18] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 979–993, September 1993.