Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors

Evangelos P. Markatos
markatos@cs.rochester.edu

Thomas J. LeBlanc
leblanc@cs.rochester.edu

Abstract

Loops are the single largest source of parallelism in many applications. One way to exploit this parallelism is to execute loop iterations in parallel on different processors. Previous approaches to loop scheduling attempt to achieve the minimum completion time by distributing the workload as evenly as possible, while minimizing the number of synchronization operations required. In this paper we consider a third dimension to the problem of loop scheduling on shared-memory multiprocessors: communication overhead caused by accesses to non-local data. We show that traditional algorithms for loop scheduling, which ignore the location of data when assigning iterations to processors, incur a significant performance penalty on modern shared-memory multiprocessors. We propose a new loop scheduling algorithm that attempts to simultaneously balance the workload, minimize synchronization, and co-locate loop iterations with the necessary data. We compare the performance of this new algorithm to other known algorithms using five representative kernel programs on a Silicon Graphics multiprocessor workstation, a BBN Butterfly, a Sequent Symmetry, and a KSR-1, and show that the new algorithm offers substantial performance improvements, up to a factor of 4 in some cases. We conclude that loop scheduling algorithms for shared-memory multiprocessors cannot afford to ignore the location of data, particularly in light of the increasing disparity between processor and memory speeds.

1 Introduction

Loops are the largest source of parallelism in most applications. Executing the many iterations of a loop on different processors enables applications to take advantage of parallel processors, and thereby reduce their running time. The problem of decomposing a loop into parallel tasks and executing those tasks on a multiprocessor involves finding the appropriate granularity of parallelism, so that the overhead of parallelism is kept small, while the workload is evenly balanced among the available processors.[1]

Both static and dynamic loop scheduling methods have been used to assign the iterations of a loop to processors. Static methods assign iterations to processors statically, minimizing run-time synchronization overhead. Dynamic methods defer the assignment of iterations to processors until run-time, and therefore can achieve better load balancing in the presence of unpredictable transient loads and variable execution times. The major difficulty in dynamic loop scheduling is to keep the run-time synchronization overhead small, while balancing the load.

The simple *static scheduling* algorithm divides the number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically. If all iterations do not take the same amount of time, or if processors begin executing loop iterations at different points in time, then load imbalance may arise, which will cause some processors to be idle while other processors continue to execute loop iterations.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [25, 28]. In this algorithm, each processor repeatedly executes one iteration of the loop until all iterations are executed. The algorithm relies on a central work queue of iterations, where each idle processor gets one iteration, executes it, and repeats

[1] In this paper we consider non-nested completely parallelizable loops only. The problem of transforming nested loops into non-nested loops has been addressed previously [24].

the same cycle until there are no more iterations to execute. Self-scheduling achieves almost perfect load balancing, since all processors finish within one iteration of each other. Unfortunately, this algorithm incurs significant synchronization overhead; each iteration requires atomic access to the central work queue. This synchronization overhead can quickly become a bottleneck in large-scale systems, or even in small-scale systems if the time to execute one iteration is small.

Uniform-sized chunking [16] reduces synchronization overhead by having each processor take K iterations, instead of one. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, resulting in less synchronization overhead. Uniform-sized chunking has a greater potential for imbalance than self-scheduling however, as processors finish within K iterations of each other in the worst case. In addition, choosing an appropriate value for K is a difficult problem, which has been solved for limited cases only.

Guided self-scheduling [24] is a dynamic algorithm that changes the size of chunks at run-time, allocating large chunks of iterations at the beginning of a loop so as to reduce synchronization overhead, while allocating small chunks towards the end of the loop to balance the workload. Under guided self-scheduling each processor is allocated $1/P_{th}$ of the remaining loop iterations, where P is the number of processors. Assuming all loop iterations take the same amount of time to complete, guided self-scheduling ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations.

Since processors take only a small number of iterations from the work queue at the end of each loop, guided self-scheduling can suffer from excessive contention for the work queue. If each iteration takes a short time to complete, then processors spend most of their time competing to take iterations from the work-queue, rather than executing iterations. *Adaptive guided self-scheduling* [11] addresses this problem by using a back-off method to reduce the number of processors competing for iterations during periods of contention. This algorithm also avoids assigning all the time-consuming iterations to one processor by assigning consecutive iterations to different processors, which reduces the risk of load imbalance that arises when the execution times of consecutive iterations vary widely but in a correlated fashion (e.g. if the execution time of iterations decreases linearly). As a result of these modifications, adaptive guided self-scheduling performs better than guided self-scheduling in many cases.

In some cases guided self-scheduling might assign too much work to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming than later iterations. The *factoring* algorithm [15] addresses this problem. Under factoring, allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because factoring allocates a subset of the remaining iterations in each phase, it balances load better than guided self-scheduling when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of factoring is not significantly greater than that of guided self-scheduling.

Like the factoring algorithm, the *tapering* algorithm [19] is designed for loops where the execution time of iterations varies in such a way as to cause load imbalance under guided self-scheduling. Tapering is used for irregular loops, where the execution time of iterations varies widely and unpredictably. The tapering algorithm uses execution profile information to estimate the average iteration time and the variance in iteration times. These estimates are used to select a chunk size that, with high probability, limits the amount of load imbalance that can occur to be within a given bound.

Although guided self-scheduling minimizes the number of synchronization operations needed to achieve perfect load balancing, the overhead of synchronization can become significant in large-scale systems with very expensive synchronization primitives. *Trapezoid self-scheduling* [31] tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling, whereas it is a decreasing function both in guided self-scheduling and in factoring.

All of these loop scheduling algorithms attempt to balance the workload among the processors without incurring substantial synchronization overhead. Each of the algorithms assumes that an individual iteration takes the same amount of time to execute on every processor. This assumption is not valid however on many shared-memory multiprocessors. The existence of memory that is not equidistant from all processors (such as local memory or a processor cache) implies that some processors are closer to the data required by an iteration than others.

Loop iterations frequently have an *affinity* [26] for a particular processor — the one whose local memory or cache contains the required data. By exploiting processor affinity, we can reduce the amount of communication required to execute a parallel loop, and thereby improve performance.

In this paper we describe a new loop scheduling algorithm called *affinity scheduling*. This algorithm attempts to balance the workload, minimize the number of synchronization operations, and exploit processor affinity. Affinity scheduling uses a deterministic assignment policy to assign repeated executions of a loop iteration to the same processor, thereby ensuring most data accesses will be to the local memory or cache. In contrast to most known algorithms, affinity scheduling employs per-processor work queues, which minimize the need for synchronization across processors. As a result of the deterministic assignment policy and per-processor work queues, affinity scheduling introduces synchronization only when load imbalance occurs. If the initial assignment of iterations to processors produces a balanced workload, all processors will finish executing at about the same time without incurring any synchronization overhead. If load imbalance occurs (i.e., a processor is idle while there are iterations to be executed), iterations are reassigned from one processor to another.

The next section provides the rationale for affinity scheduling, and describes the affinity scheduling algorithm. Section 3 presents an analytic evaluation of affinity scheduling and a comparison with other known techniques. Section 4 contains an experimental comparison of the known loop scheduling algorithms, based on five representative kernel programs running on a Silicon Graphics multiprocessor workstation and a BBN Butterfly. Section 5 uses experiments on a Sequent Symmetry and Kendall Square Research KSR-1 multiprocessor to examine how the performance benefits of affinity scheduling scale with an increase in communication costs, processors, or problem size. Section 6 places our results in perspective, by discussing the broader issue of scheduling in shared-memory multiprocessors. Finally section 7 summarizes our results and presents our conclusions.

2 Affinity Scheduling

2.1 Rationale

Our motivation for exploiting processor affinity in loop scheduling derives from the observation that, for many parallel applications, the time spent bringing data into the local memory or cache is a significant source of overhead, ranging between 30-60% of the total execution time [5, 12, 20]. While data movement caused by true sharing is unavoidable, it is possible to minimize data movement caused by a poor assignment of iterations to processors. By scheduling a loop iteration on the processor whose local memory or cache already contains the necessary data, we can significantly reduce the execution time of the iteration.

Affinity scheduling is based on the assumption that, in many cases, loop iterations do in fact have an affinity for a particular processor. In order for this assumption to hold, it must be the case that: (1) the same data is used over and over by an iteration, and (2) the data is not removed from the local memory (or cache) before it can be reused.

Data reuse is common in many applications, particularly those that employ iterative algorithms wherein a parallel loop is nested within a sequential loop. In such cases, each iteration of the parallel loop accesses the same (or nearby) data on successive iterations of the enclosing sequential loop. During the first iteration of the sequential loop, each iteration of the nested parallel loop loads the required data into the local memory or cache, where it may remain during subsequent iterations of the enclosing sequential loop.

Data reuse may also occur in programs produced by a parallelizing compiler. Earlier work has suggested that nested loops be interchanged in such a way as to reduce synchronization and communication overhead [13]. The resulting loop structure nests a parallel loop within a sequential loop, again producing the desired form. If necessary, several parallel loops can be coalesced into one [23].

Whether data resides in local storage long enough to be reused is a more complicated question. Data may be removed from local storage to make room for the data needed by other iterations of the same parallel loop, or another application. If two applications share a single processor, then the data required by one application may be forced out of local storage by the other application. We can minimize this effect under time sharing by increasing the quantum, so that the time required to reload the cache is small relative to the quantum size. Even in this case, affinity scheduling will be of little help if the iterations of a parallel loop cannot be executed repeatedly within a single quantum (a distinct possibility on small-scale multiprocessors with extremely large loops). A better solution is to avoid time-sharing altogether, and employ space sharing instead, wherein each application gets some number of processors for a relatively long period of time. Space sharing not only avoids cache (and memory) interference between applications, it also has other attractive properties that result in improved performance over time sharing [12, 8, 22].

Even if a set of processors are dedicated to a single application, the data needed by one iteration of a loop may be evicted from local storage to make room for the data needed by another iteration of the same loop. [2] Although eviction may have been a serious problem in the past, when local caches (or memory) were quite small, it is less likely to occur in modern multiprocessors. The size of local caches and memory has grown substantially in the last few years in order to bridge the ever-widening gap between processor speeds and communication speed. For example, the BBN Butterfly offered 1 MB of local memory per node in the early 1980's, and 16 MB of memory per node in 1990. With regards to cache-coherent machines, the Sequent Symmetry (introduced around 1987) has 64 KB local caches, the Silicon Graphics 4D/480GTX (introduced around 1990) has 1 MB (second-level) local caches, and the Kendall Square Research multiprocessor (introduced in 1991) has 32 MB of coherent local memory (or cache) per processor. Architecture trends suggest that the density in DRAM memory chips (and the resulting memory size) doubles in size every 1.5 years [14]. Given this trend, the chances are good that the local cache or memory will be large enough to hold the data for many iterations of a loop.

If eviction occurs even with very large local storage, then the program may not be suitable for execution on a multiprocessor. Efficient execution requires that a processor's working set fit in the local cache (or memory). If the working set consists of multiple iterations, and the associated data doesn't fit in local storage, then the program will thrash, spending most of its time loading data from non-local storage. This type of program will not execute efficiently on modern multiprocessors regardless of the loop scheduling algorithm in use.

Finally, there are loops that can execute efficiently on shared-memory multiprocessors, but which do not exhibit affinity. For example, a large parallel loop might force an eviction on every iteration, but if each iteration is time-consuming and makes efficient use of the local cache, then the evictions will not dominate the execution cost. Our work does not address this case; we exploit affinity only where it exists, and thereby significantly improve the performance of a large class of programs.

2.2 Affinity Scheduling Algorithm

We consider the loop scheduling problem to have three dimensions: load imbalance, synchronization overhead, and communication overhead due to non-local memory accesses. Our algorithm for affinity scheduling builds on previous work in loop scheduling, while also attempting to exploit processor affinity. The main ideas underlying our algorithm are:

- As with many known algorithms, we assign large chunks of iterations at the start of loop execution, so as to reduce the need for synchronization, and assign progressively smaller chunks to balance the load.
- We use a deterministic assignment policy to ensure that an iteration is always assigned to the same processor. After the first execution of the iteration, that processor will contain the required data, so subsequent executions of the iteration will not need to load the data into local storage.
- We reassign a chunk to another processor (which also involves moving the required data) only if necessary to balance the load. An idle processor removes chunks from another's queue, and executes them indivisibly, so an iteration is never reassigned more than once.

We will assume that the underlying hardware or software implements a coherent memory, so that data is copied into local storage when first accessed. This copy is implemented in hardware on machines with coherent caches, such as the Symmetry and Silicon Graphics machine, and may be implemented in the operating system on machines lacking coherent caches, like the Butterfly [6, 7, 17].

Our affinity scheduling algorithm divides the iterations of a loop into chunks of size $\lceil N/P \rceil$, where N is the number of iterations in the loop, and P is the number of available processors. The i_{th} chunk of iterations is always

[2] We assume that the number of iterations is much larger than the number of available processors, and therefore each processor must execute multiple iterations.

```
loop_initialization(N,P)
// N is the number of loop iterations, P is the number of processors
{
    for(i = 0 ; i < P ; i++) {
        // assign iterations ceil(i*N/P) to min(N,ceil((i+1)*N/P))
        // to processor i
        assign_iterations(i)
    }
}

loop // executed by each processor
    // get 1/k of the local iterations to execute
    range = get_iterations_from_local_queue(1/k) ;
    if (range == empty)
        max_load = find_most_loaded_processor()  ;
        if (max_load == nil) break ;
        // get 1/P of the iterations from the most loaded processor
        range = get_iterations_from_nonlocal_queue(max_load,1/P);
        if (range == nil) break ;
        execute(range) ;
forever
```

Figure 1: Pseudocode for Affinity Scheduling

placed on the local work queue of processor i. When a processor is idle, it removes $1/k$ of the iterations in its local work queue and executes them. [3] If a processor's work queue is empty, it finds the most loaded processor, removes $\lceil 1/P \rceil$ of the iterations in that processor's work queue, and executes them. [4]

Note that we distinguish between assigning a loop iteration to a processor's work queue, and executing the iteration on that processor. Initially, loop iterations are assigned to a processor's work queue in chunks of size $1/P$, so as to balance the load statically. Processors execute $1/k$ of the remaining iterations on their local work queue at a time, which corresponds to at most N/kP iterations. Processors execute $1/P$ of the remaining iterations from a remote work queue, which corresponds to at most N/P^2 iterations.

A pseudocode description for affinity scheduling can be found in Figure 1. Although we implemented this algorithm by hand for our experiments, it could easily be employed by a parallelizing compiler.

In our current implementation an idle processor examines the work queues of all the other processors and removes work from the queue with the most iterations. Our experimental results suggest that this implementation suffices on small-scale machines like the Iris, and on medium-scale configurations, like the 64-processor KSR-1 we used in our experiments. However, this implementation would not be efficient on a large-scale machine, where a scalable or randomized policy would be more appropriate [9].

There are two important differences between affinity scheduling and previous dynamic loop scheduling algorithms. First, the initial assignment of chunks to processors in affinity scheduling is deterministic. That is, processor i is always assigned the i_{th} chunk of iterations to execute. For many programs, this assignment ensures that repeated executions of the loop will access data that is already stored in the local memory or cache. Second, affinity scheduling initially assumes that load imbalance will not occur, and therefore assigns the same number of iterations to each processor's work queue. Each processor gets iterations from its own local work queue; accesses to different work queues can proceed in parallel, and each access is local, and therefore cheap. If load imbalance arises, the algorithm migrates iterations from loaded processors to idle ones. Migrating iterations causes the associated data to move twice in most cases; the data must first move to an idle processor to alleviate load imbalance,

[3] The constant k is a parameter of our algorithm. In most of our experiments we assume k equals P. We describe the effects of changes in k in section 3.

[4] Synchronization is required to remove iterations from a work queue, but not to check the load on a processor.

and then move back to its original location to restore processor affinity. However, under affinity scheduling this overhead is introduced only when load imbalance arises, whereas other algorithms incur this overhead on every iteration.

Despite these differences, we will show that affinity scheduling has all the advantages of the best dynamic loop scheduling algorithms. That is, it balances the load dynamically, minimizes synchronization, and is immune to the arrival and departure of processors in the system.

2.3 Modified Factoring

The affinity scheduling algorithm is intended to address all three dimensions of the loop scheduling problem. An entirely new algorithm is not needed in order to deal with communication overhead however; previous methods can be extended to deal with this new dimension. We will now describe how to reduce the need for communication in the factoring algorithm.

During each phase of the factoring algorithm, iterations are grouped into P equal-sized chunks. Those chunks are placed in the central work queue, and each processor removes the next available chunk. Our modification to this scheme is that during each phase, processor i always removes the i_{th} chunk from the queue, rather than the chunk at the front of the queue. If the i_{th} chunk for this phase is no longer in the queue, an idle processor removes the first chunk in the queue. [5] By selecting the same chunk each time a loop executes, the modified factoring algorithm ensures that an iteration has access to the data it referenced during an earlier execution. However, each access to the central work queue is considerably more expensive than in the case of factoring or guided self-scheduling, and the additional overhead may eliminate the benefits of scheduling iterations close to their data. We will examine this issue in our experiments with the various loop scheduling algorithms.

3 Analytic Evaluation

Under affinity scheduling each iteration is initially assigned to a processor based on affinity considerations, and then reassigned to another processor if necessary to balance the load. Since an iteration is reassigned at most once, the algorithm is stable under load imbalance conditions and avoids processor thrashing [27], where processors spend more time executing migrated work than executing their own assigned work.

The fact that each iteration is reassigned to another processor at most once by affinity scheduling does not imply that the number of synchronization operations associated with reassignment is linear in the number of iterations. Since iterations are assigned (and reassigned) to processors in chunks, synchronization overhead is amortized over the number of iterations in a chunk. Theorem 3.1 places a bound on the synchronization overhead induced by affinity scheduling.

Lemma 3.1 *[24] If each processor takes $1/k_{th}$ of the iterations in a work queue, the worst-case number of accesses is $O(k\log(N/k))$, where N is the initial number of iterations in the work queue.*

Theorem 3.1 *Affinity scheduling will incur at most $O(k\log(\frac{N}{Pk}) + P\log(\frac{N}{P^2}))$ synchronization operations on each work queue.*

Proof: When a processor accesses its local work queue, it removes $1/k_{th}$ of the remaining iterations. Initially, each local queue contains N/P iterations. From Lemma 3.1 it follows that a processor will access its own work queue at most $O(k\log(\frac{N}{Pk}))$ times. When another processor accesses that work queue, it removes $1/P_{th}$ of the remaining iterations. Again using Lemma 3.1 we conclude that no more than $O(P\log(\frac{N}{P^2}))$ accesses by other processors can occur. So the total number of synchronization operations to each work queue is (in the worst case) $O(k\log(\frac{N}{Pk}) + P\log(\frac{N}{P^2}))$. ∎

By way of comparison, guided self-scheduling induces $O(P\log(N/P))$ synchronization operations (worst case) on the central work queue, factoring induces $O(P\log(N))$ operations, and trapezoid self-scheduling induces $4P$ operations.

[5] As with affinity scheduling, load imbalance may cause data to move twice.

One common assumption in loop scheduling is that all processors do not start executing loop iterations at the same time, as there may have been delays due to previous load imbalance or synchronization operations. Theorem 3.2 places a bound on the degree of imbalance that can result from using affinity scheduling under this assumption.

Lemma 3.2 *[24] Assume that all iterations of a loop take the same amount of time to complete. If each processor takes $1/P_{th}$ of the remaining iterations in a work queue, then all processors finish within one iteration of each other.*

Theorem 3.2 *Assume that all iterations of a loop take the same amount of time to complete, and that not all processors start executing loop iterations at the same time. Under affinity scheduling, all processors will finish within $\frac{N(P-k)}{P(P-1)k}+1$ iterations of each other.*

Proof: Under affinity scheduling the worst-case imbalance occurs when all processors except one finish working on their own iterations just as the remaining processor is ready to begin working on its iterations. In this case the late processor will take $\frac{N}{Pk}$ iterations from its own work queue, leaving $\frac{(k-1)N}{Pk}$ iterations to be divided among the other $P-1$ processors. According to Lemma 3.2, if each of the $P-1$ processors removes $1/P_{th}$ of the remaining iterations, they will all finish within one iteration of each other. Under this scenario, one processor will have to execute $\frac{N}{Pk}$ iterations, while each of the other processors has only $\frac{(k-1)N}{P(P-1)k}$ iterations to execute. The resulting imbalance is $\frac{N}{Pk}-\frac{(k-1)N}{P(P-1)k}$, or $\frac{N(P-k)}{P(P-1)k}$. Since processors do not, in general, start executing iterations at exactly the same time, there could be an additional disparity of one iteration. As a result, all processors will finish within $\frac{N}{Pk}-\frac{(k-1)N}{P(P-1)k}+1$, or $\frac{N(P-k)}{P(P-1)k}+1$ iterations from each other. ∎

Under guided self-scheduling and factoring, all processors finish within one iteration of each other. Theorem 3.2 implies that if the constant k is equal to the number of processors P, then all processors will finish within one iteration of each other under affinity scheduling as well.

From these results, we see that k plays an important role in the overhead of affinity scheduling. If k is a small constant, then the number of synchronization operations per local work queue is small (proportional to $\log(\frac{N}{kP})$), while the potential for load imbalance is high (proportional to $\frac{N}{P}$). As k approaches P, affinity scheduling approaches the same worst-case load imbalance as guided self-scheduling and factoring, while simultaneously increasing the number of synchronization operations on the local work queue by a factor of P.

Selecting an optimal value for k is a difficult task, since the best choice depends on a tradeoff between the benefits of load balancing versus the costs of synchronization. This problem is not unique to affinity scheduling however, because most loop scheduling algorithms must make this same tradeoff.

Under affinity scheduling we have separated the synchronization costs associated with access to the local work queue (as represented by k, the fraction of iterations removed from the local work queue) from the synchronization costs associated with access to remote work queues (as represented by P, the fraction of iterations removed from a remote work queue). Since synchronization operations on local work queues are usually inexpensive, we use $k=P$ in our implementation, which results in small initial chunks (N/P^2) and thus good load balancing properties. Smaller values of k could be used to reduce the number of accesses to local queues, while increasing the potential for load imbalance.

We next consider the size of chunks of iterations that should be used with parallel loops wherein the time each iteration takes to execute is a decreasing function of the iteration index. These loops are among the most difficult to schedule because they often result in load imbalance, particularly when the scheduling algorithm assigns large chunks of loop iterations to the first few processors and successively smaller chunks to other processors. Theorem 3.3 indicates how many iterations each chunk should contain so that no more than $1/P_{th}$ of the remaining work is assigned to a processor at one time.

Theorem 3.3 *Assume a parallel loop with N iterations, where the i_{th} iteration takes time proportional to $(N-i)^k$. A chunk of size $\frac{1}{(k+1)P}$ of the iterations corresponds to at most $1/P_{th}$ of the remaining work to be done.*

Proof: Assume that there are R more iterations to be executed. The index of the first iteration is r; the index of the last iteration is $r+R-1$. Assume also that the time iteration x takes to complete is $c\cdot(r+R-x)^k$. This function suggests that the iteration with index r takes time $c\cdot R^k$, while the iteration with index $r+R-1$ takes time c to complete.

The total work remaining to be done in the loop is:

$$\sum_{x=r}^{r+R-1} c \cdot (r+R-x)^k$$

The time required by the first chunk of size $\frac{R}{(k+1)P}$ iterations is

$$\sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r+R-x)^k$$

In order not to create load imbalance, we want this work to be $1/P_{th}$ of the total work to be done, or

$$\sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r+R-x)^k = \frac{1}{P} \cdot \sum_{x=r}^{r+R-1} c \cdot (r+R-x)^k$$

Using integral approximation, we will prove the theorem by proving:

$$\begin{aligned}
&\sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r+R-x)^k \\
&\leq c \cdot R^k + \int_r^{r+\frac{R}{(k+1)P}-1} c \cdot (r+R-x)^k dx \\
&\leq \frac{1}{P} \int_r^{r+R} c \cdot (r+R-x)^k dx \\
&\leq \frac{1}{P} \sum_{x=r}^{r+R-1} c \cdot (r+R-x)^k
\end{aligned} \tag{1}$$

Because $(r+R-x)^k$ is a decreasing function of x we know that for all $b \geq r+1$:

$$\sum_{x=r+1}^{b} (r+R-x)^k \leq \int_r^b (r+R-x)^k \leq \sum_{x=r}^{b-1} (r+R-x)^k \tag{2}$$

These inequalities represent an upper and lower bound on the numerical value of the integral. Using (2) it is straightforward to prove the first and last inequality of the set of inequalities (1). In order to complete the proof of (1) we need to show that

$$c \cdot R^k + \int_r^{r+\frac{R}{(k+1)P}-1} c \cdot (r+R-x)^k dx \leq \frac{1}{P} \int_r^{r+R} c \cdot (r+R-x)^k dx$$

or

$$(k+1)R^k + (R)^{k+1} - \left(R+1-\frac{R}{P(k+1)}\right)^{k+1} \leq \frac{1}{P} R^{k+1}$$

or

$$\frac{k+1}{R} + 1 - \left(1+\frac{1}{R}-\frac{1}{P(k+1)}\right)^{k+1} \leq \frac{1}{P} \tag{3}$$

Because $(1 \pm 1/x)^k \geq 1 \pm k/x$, we have

$$\frac{k+1}{R} + 1 - (1+\frac{1}{R}-\frac{1}{P(k+1)})^{k+1} \leq \frac{k+1}{R} + 1 - 1 - \frac{k+1}{R} + \frac{1}{P} \leq \frac{1}{P}$$

which proves inequality (3), which in turn proves inequality (1). ∎

Theorem 3.3 suggests that when all iterations take the same amount of time, $1/P_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload. When the iterations have a decreasing triangular form, that is iteration i takes time proportional to $(N - i)$, then the first $1/(2P)_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload. When the iterations have a decreasing parabolic form, that is iteration i takes time proportional to $(N - i)^2$, then the first $1/(3P)_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload.

Loops with decreasing workloads, such as those described above, are among the most difficult loops to schedule. The scheduling algorithm must be careful to avoid assigning so many iterations to one processor that the remaining iterations are insufficient to balance the workload. Our experiments indicate that the factoring and trapezoid algorithms have better load balancing properties than guided self-scheduling for this type of loop. This result can be traced to the fact that both factoring and trapezoid start with a chunk that contains $1/(2P)_{th}$ of the iterations to be scheduled, while guided self-scheduling starts with a chunk that contains $1/P_{th}$ of the iterations. According to theorem **3.3**, the first chunk will be the bottleneck in guided self-scheduling, while it will not be a bottleneck for factoring or trapezoid.

In general, if a loop scheduling algorithm assigns less than $1/P_{th}$ of the remaining workload to each idle processor, then the minimum imbalance will result. Theorem **3.3** states how many iterations correspond to this fraction of the remaining workload for loops wherein successive iterations require a polynomially decreasing amount of work. If the amount of work per iteration increases polynomially, then the loop is easy to schedule: $1/(kP)_{th}$ of the remaining iterations *always* corresponds to less than $1/P_{th}$ of the remaining work.

Summarizing our results, affinity scheduling (with $k = P$) offers worst-case load imbalance guarantees that are the same as (or in some cases better than) those of guided self-scheduling and factoring, but can, in the worst case, introduce about P times more synchronization operations. Fortunately, these synchronization operations are directed to P different work queues, and so the number of *serializable* synchronization operations under affinity scheduling is somewhat smaller than the number of serializable synchronization operations under guided self-scheduling or factoring. Since affinity scheduling can also dramatically reduce communication overhead, affinity scheduling should perform much better than either guided self-scheduling or factoring. We will now examine the relative performance of these loop scheduling algorithms experimentally.

4 Experimental Evaluation

In order to evaluate the performance benefits of affinity scheduling, we implemented many of the known loop scheduling methods by hand on a Silicon Graphics 4D/480GTX Iris workstation, a bus-based, cache-coherent machine with 8 processors, a BBN Butterfly I shared-memory multiprocessor with 60 processors, and a KSR-1 large-scale cache-coherent multiprocessor with 64 processors. We then measured the performance of each of the scheduling algorithms on a suite of applications.

4.1 Scheduling Algorithms

We implemented the following loop scheduling algorithms by hand on the Iris: static scheduling (STATIC), self-scheduling (SS), guided self-scheduling (GSS), factoring (FACTORING), trapezoid self-scheduling (TRAPEZOID), affinity scheduling with $k = P$ (AFS), modified factoring (MOD-FACTORING), and a hand-optimized algorithm (BEST-STATIC). BEST-STATIC represents our attempt at the best static assignment possible, given complete knowledge of the application and its input. We implemented this assignment by hand, after examining the application and the input, so as to maximize locality of reference and minimize load imbalance. While not generally realizable, since it requires programmer intervention and assumes knowledge of the application's input, BEST-STATIC is a useful base-line for evaluating other loop scheduling algorithms.

4.2 Applications

We carefully selected five application programs that present loop scheduling algorithms a range of opportunities for addressing load imbalance, synchronization overhead, and communication overhead. Our application suite contains the following programs:

- Successive Over-Relaxation (SOR):

```
DO SEQUENTIAL 19 I = 1,MAXITERATIONS
    DO PARALLEL 29 J = 1,N
        DO SEQUENTIAL 39 K = 1,N
           A(J,K) = UPDATE(A,J,K)
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

All iterations of the parallel loop take about the same time to execute, so better load balancing algorithms are not likely to produce much better performance. However, the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix, so scheduling algorithms that exploit processor affinity are likely to produce much better performance.

- Gaussian Elimination:

```
DO SEQUENTIAL 19 K = 2,N
    DO PARALLEL 29 I = K,N
        DO SEQUENTIAL 39 J = K-1,N+1
            A[I][J] = A[I][J] - A[K-1][J] * A[i][K-1]/A[K-1][K-1]
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

This application exhibits some load imbalance across iterations, and offers some opportunities for exploiting processor affinity. Although successive executions of an iteration of the parallel loop do not access exactly the same matrix elements each time, there is significant overlap in the elements referenced by successive executions of an iteration. We expect scheduling algorithms that exploit affinity to improve performance, but not as much as in the previous case.

- Transitive Closure:

```
DO SEQUENTIAL 19 K = 1,N
    DO PARALLEL 29 J = 1,N
        IF (A(J,K) .EQ. TRUE)  THEN
            DO SEQUENTIAL 39 I = 1,N
                IF (A(K,I) .EQ. TRUE) A(J,I) = TRUE
            39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

The distinguishing characteristic of this application is that each iteration of the parallel loop may take time $O(1)$ or $O(N)$ (where the input matrix is of size $N \times N$), depending on the input data. Since the input values determine the variation in iteration execution time, this application will serve to evaluate the effectiveness of load balancing for each scheduling algorithm. This application will also benefit from some form of affinity scheduling, since the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix.

- Adjoint Convolution:

```
DO PARALLEL 19 I = 1,N*N
    DO SEQUENTIAL 29 K = I,N*N
        A(I) = A(I) + X*B(K)*C(I-K)
    29  CONTINUE
19  CONTINUE
```

This application exhibits significant load imbalance; the i_{th} iteration of the parallel loop takes time proportional to $O(n^2 - i)$. There is no affinity to exploit however, so this application serves to evaluate the effectiveness of load balancing in the absence of affinity.

```
DO SEQUENTIAL 1 I1 = 1,50
     DO PARALLEL 2 I2 = 1,10
         DO PARALLEL 3 I3 = 1,10
             DO PARALLEL 4 I4 = 1,10
                 {10}
                 [if C then {50}]
             4 CONTINUE
         3 CONTINUE
     2 CONTINUE
     DO PARALLEL 5 I5 = 1,100
         {50}
         DO PARALLEL 6 I6 = 1,5
             {100}
             [if C then {30}]
         6 CONTINUE
     5 CONTINUE
     DO PARALLEL 7 I7 = 1,20
         DO PARALLEL 8 I8 = 1,4
            {30}
         8 CONTINUE
     7 CONTINUE
 1 CONTINUE
```

Figure 2: Structure of the L4 application

Application	Load imbalance	Affinity
SOR	none	yes
Gauss elimination	little	yes
Transitive closure	input dependent	yes
Adjoint convolution	large	no
L4	little	no

Table 1: Load imbalance and affinity characteristics of the application suite.

- L4: This application was used as a benchmark in [24]; we include it in our study for comparison with previously published results. The structure of L4 can be found in Figure 2. L4 is an example of a hybrid application with non-perfectly nested and multi-way nested parallel loops. In our experiments, all **if** statements are true with probability 0.5. Because this application does not perform any memory accesses, there is no affinity to be exploited.

Table 1 summarizes the properties of our application suite with respect to load imbalance and affinity. If an application exhibits load imbalance, the iterations of the loop may take varying amounts of computation time, so a static scheduling algorithm may not be appropriate. If an application exhibits affinity, we can improve performance by scheduling iterations appropriately.

4.3 Comparison of Loop Scheduling Algorithms

In this section we compare the performance of the various loop scheduling algorithms using the application suite. Due to the large number of scheduling algorithms we consider, we will represent algorithms with comparable performance with a single line in the performance graphs.

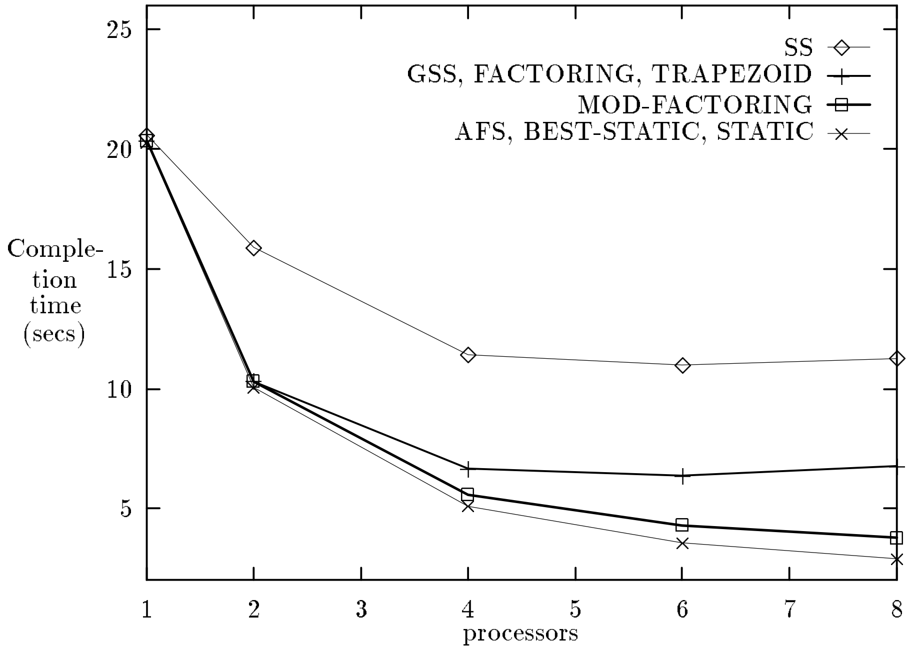

Figure 3: Performance of loop scheduling algorithms for SOR.

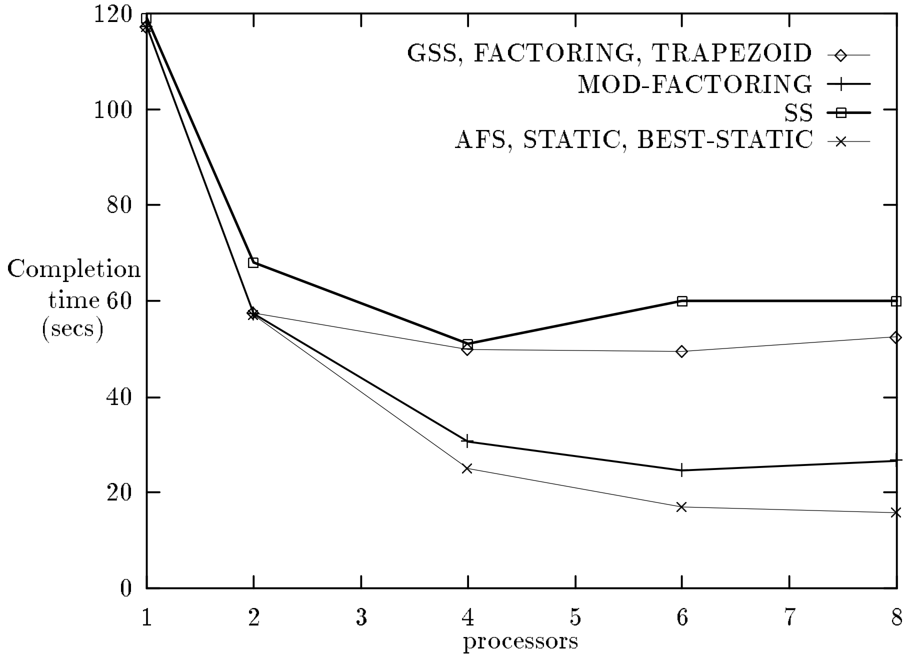

Figure 4: Performance of loop scheduling algorithms for Gaussian elimination.

Figure 3 presents the completion time (in seconds) of the SOR program ($N = 512$) running on 1 to 8 processors. As can be seen in the figure, SS performs the worst of all, due to its high synchronization overhead. Other algorithms with lower synchronization overhead, such as GSS, FACTORING, and TRAPEZOID, perform much better than SS — since there is no significant difference in the execution time of iterations, sophisticated load balancing schemes aren't necessary for this application. All of these algorithms perform worse than the algorithms that exploit affinity. Both STATIC and AFS are comparable to the best possible static algorithm. MOD-FACTORING lies between AFS and FACTORING, since it requires less communication than FACTORING, but requires more expensive access to the work queue than AFS. These results confirm that affinity scheduling can improve the performance of loop scheduling algorithms.

Figure 4 plots the completion time of the Gaussian elimination program ($N = 768$) under the different scheduling algorithms. It is surprising to see that none of the scheduling algorithms that ignore processor affinity can effectively utilize more than two processors. There is simply too much contention for the shared bus under these algorithms, since every iteration must load data into the local cache. SS performs worst of all, because of its high synchronization overhead, but the performance difference narrows quickly as the communication costs of GSS, FACTORING, and TRAPEZOID start to dominate synchronization costs. Once again, AFS and STATIC perform the best; they very close to BEST-STATIC in the worst case, and a factor of 3 better than the traditional dynamic loop scheduling algorithms. MOD-FACTORING is also much better than GSS, FACTORING and TRAPEZOID, but not as good as AFS. Although MOD-FACTORING preserves some affinity, slight load imbalances in the workload may easily result in iteration reassignment, and thus loss of affinity. AFS, STATIC can effectively use all 8 processors, while MOD-FACTORING can effectively use about 6 processors.

This application is a perfect example of the fact that the dominant source of overhead in many applications is communication (caused by cache misses), not synchronization. Loop scheduling algorithms that focus on synchronization overhead alone perform poorly when compared to algorithms that reduce communication overhead by exploiting processor affinity.

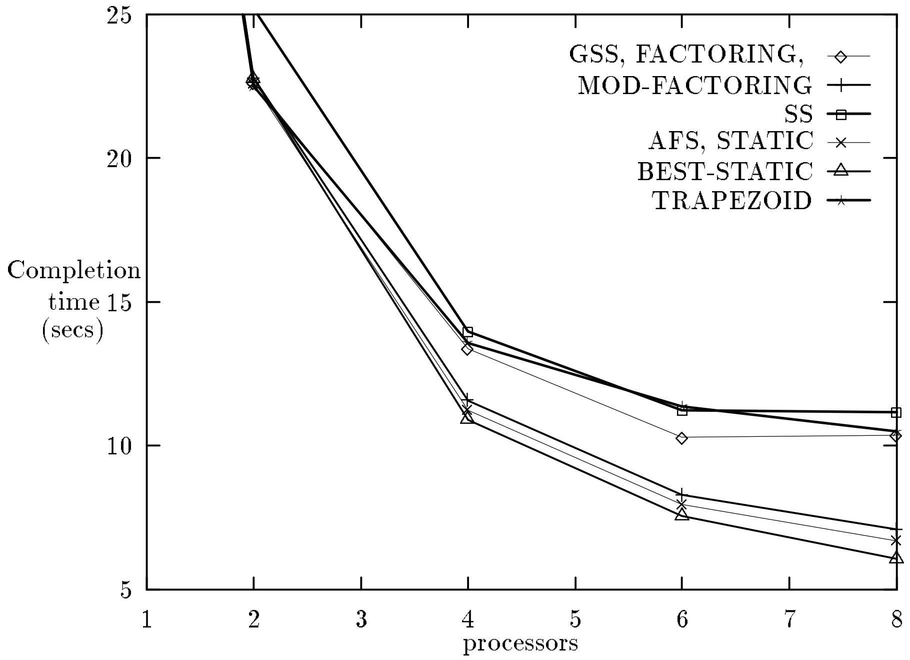

Figure 5: Performance of loop scheduling algorithms for transitive closure (random input).

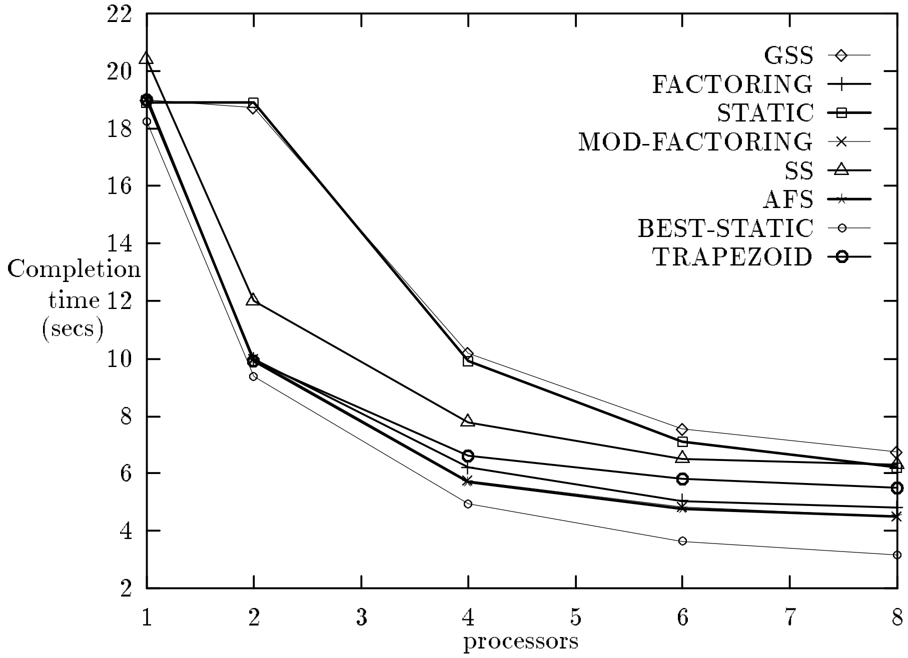

Figure 6: Performance of loop scheduling algorithms for transitive closure (skewed input).

Figure 5 presents the completion time of the transitive closure program when given a random input graph of 512 nodes, with about 8% of the edges present. Because the load is averaged over all iterations, preserving affinity takes precedence over load balancing. As a result, AFS, STATIC, and MOD-FACTORING perform better than GSS, FACTORING, SS, and TRAPEZOID.

Figure 6 presents the completion time of the transitive closure application when given a skewed input graph of 640 nodes containing a clique of 320 nodes, and no other edges. This is the first example where there is significant imbalance in the computation across iterations, which explains why STATIC performs poorly. Although SS manages to balance the load, it still suffers from high synchronization overhead. The surprising result in Figure 6 is that GSS performs worst of all. Although GSS assigns only $1/P$ of the iterations to the first processor, those iterations contain $2/P_{th}$ of the total work; the remaining iterations do not have enough work to balance the load. Both FACTORING and TRAPEZOID start with a smaller initial chunk of iterations, and therefore balance the load better. AFS and MOD-FACTORING have the same load balancing properties as FACTORING and TRAPEZOID, but exploit affinity as well.

Although AFS and MOD-FACTORING perform the best, the improvement over FACTORING and TRAPEZOID is not greater than 15%. The existence of significant load imbalance forces an affinity scheduler to override the initial assignment of iterations to processors and instead execute iterations on any available idle processor. Each time an iteration moves to another processor, the data must be loaded into a different cache. This is also why AFS does not perform as well as BEST-STATIC, which has knowledge of the input, and is therefore able to distribute the clique nodes evenly among the processors, while maintaining processor affinity.

A modification to the AFS algorithm that might reduce the need to reassign iterations for load balancing purposes is to execute an iteration on the processor on which it last executed, rather than assigning an iteration to the same processor every time it executes, and reassigning as necessary to balance the load. This modification would be particularly effective in many simulations of physical systems, where the conditions that produce load imbalance do not vary wildly from one simulation step to the next. If we assign an iteration to the same processor

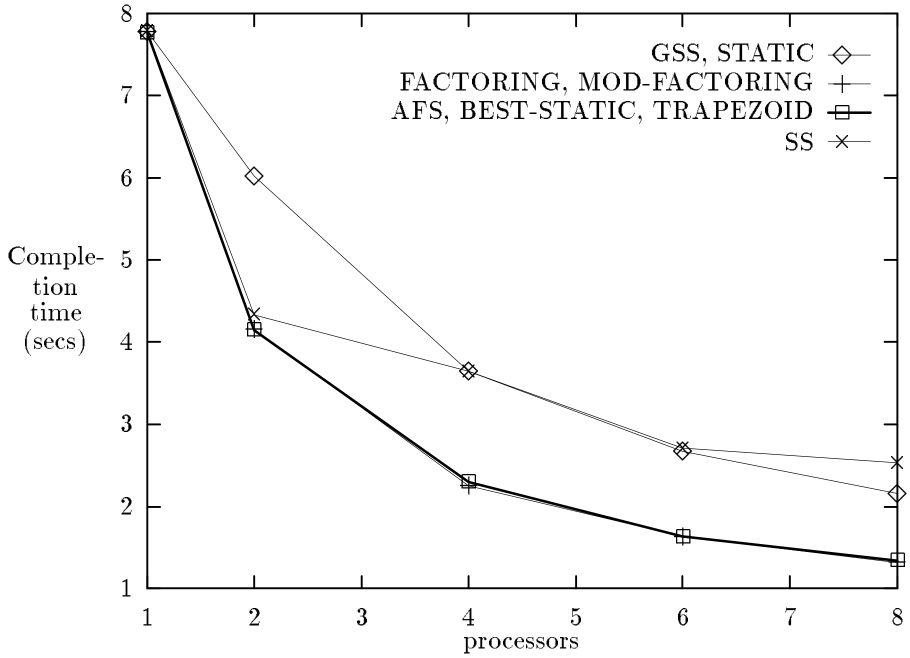

Figure 7: Performance of loop scheduling algorithms for adjoint convolution.

every phase (as in the AFS algorithm), then any load imbalance that exists in one phase is likely to exist in the next, and the iteration will be reassigned each phase. If we assign each iteration to the processor where it last executed, then reassignments performed in one phase to balance the load will likely be unnecessary in the next phase. Assuming that the distribution of work does not change rapidly from phase to phase, this heuristic may result in fewer reassignments of iterations, and therefore less communication. Unfortunately, it may also lead to fragmentation of iterations, where each processor must execute several chunks of a few iterations each, rather than one large chunk iterations. As a result, the costs and benefits of this approach depend on the volatility of load imbalance in the application.

Figure 7 presents the performance of the scheduling algorithms for the adjoint convolution program with $N = 75$. In this application, iterations have no affinity for a particular processor, since the parallel loop is not embedded within a sequential loop. There is significant load imbalance across iterations however, since the first iteration takes time proportional to $O(N^2)$, while the last iteration takes time proportional to $O(1)$. As expected, loop scheduling algorithms that emphasize load balancing, such as FACTORING, MOD-FACTORING, TRAPEZOID and AFS, perform the best. GSS and the static methods assign too much work to the first few processors, and suffer load imbalance as a result. SS again suffers from high synchronization overhead. These results are consistent with those of [15].

We should note that a trivial change to our implementation of GSS would improve its performance to be comparable to FACTORING, although not as good as AFS for these examples. Instead of taking $\lceil N/P \rceil$ iterations, each processor could take $\lceil N/(kP) \rceil$ iterations, where k is an appropriate constant. With this change, GSS could start with smaller chunks, leaving more opportunities to balance the load, without introducing significant synchronization overhead. Eager and Zahorjan [11] argue that decreasing the chunk size is not enough to balance the load if the execution time of iterations decreases at a fast enough rate. Theorem **3.3** quantifies this relationship between the variance in iteration execution times and the resulting load imbalance, and suggests that if the rate of

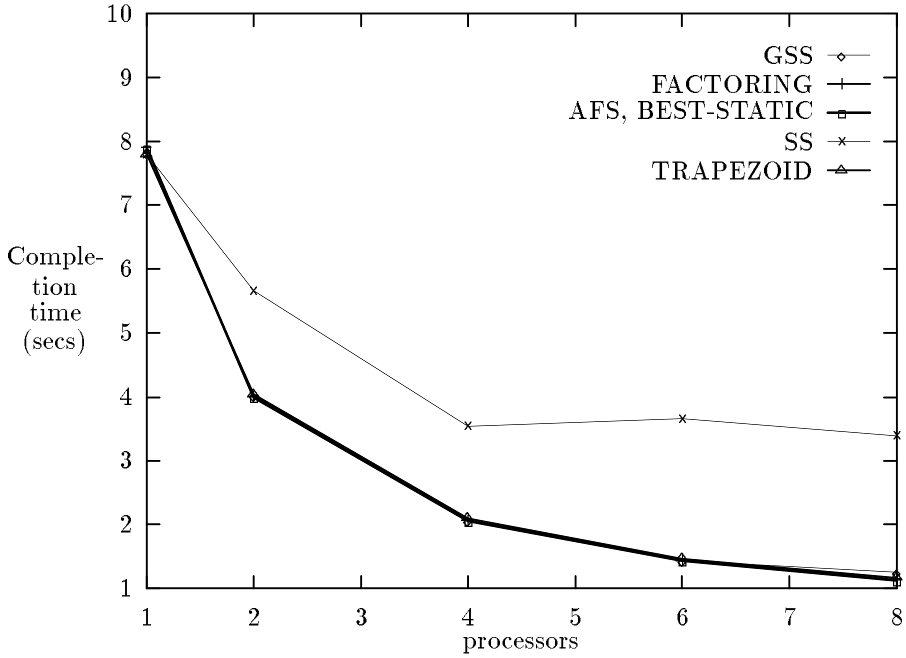

Figure 8: Performance of loop scheduling algorithms for adjoint convolution (reverse index scheduling).

decrease is polynomial with exponent k, and each processor takes no more than $\frac{1}{(k+1)P}$ of the remaining iterations, no imbalance will occur. Thus, a simple decrease in the chunk size is probably enough to balance the load for nearly all programs.

Load imbalance is particularly important in the adjoint convolution problem because the computation times of the iterations decrease linearly; the first few chunks could become a bottleneck. Rather than decrease the chunk size at the beginning of the loop, we could schedule the loop backwards, so that the last iterations execute first. (Reverse execution works in this case because there are no dependencies among the iterations.) Figure 8 presents the performance of several loop schedulers on the adjoint convolution problem, when scheduling the iterations in reverse order. We see that all scheduling algorithms (apart from SS) perform reasonably well, and are comparable in performance to the best scheduling algorithms that execute the loop iterations in index order. Although executing the iterations in reverse order may increase the potential load imbalance (since the last iterations to be executed are the most time-consuming), the potential imbalance is a negligible percentage of the total completion time of the application. If there are N iterations, and the i_{th} iteration takes $O(N-i)$ time to execute, the last iteration to be executed under reverse ordering takes about $O(N)$ time, while the total completion time of the application is about $O(N^2/P)$. Thus, the potential imbalance (time $O(N)$) is asymptotically small when compared to the total completion time ($O(N^2/P)$), unless the number of processors is on the order of the number of loop iterations.

Finally, figure 9 plots the performance of the loop scheduling algorithms for the L4 application. Since there are no memory references in L4, we would not expect an affinity scheduler to perform any better than a scheduler that ignores affinity. In fact, all loop schedulers perform about the same, although the dynamic schedulers perform a bit better than the static scheduler, and self-scheduling is clearly the worst. These results for L4 are consistent with those reported in [24].

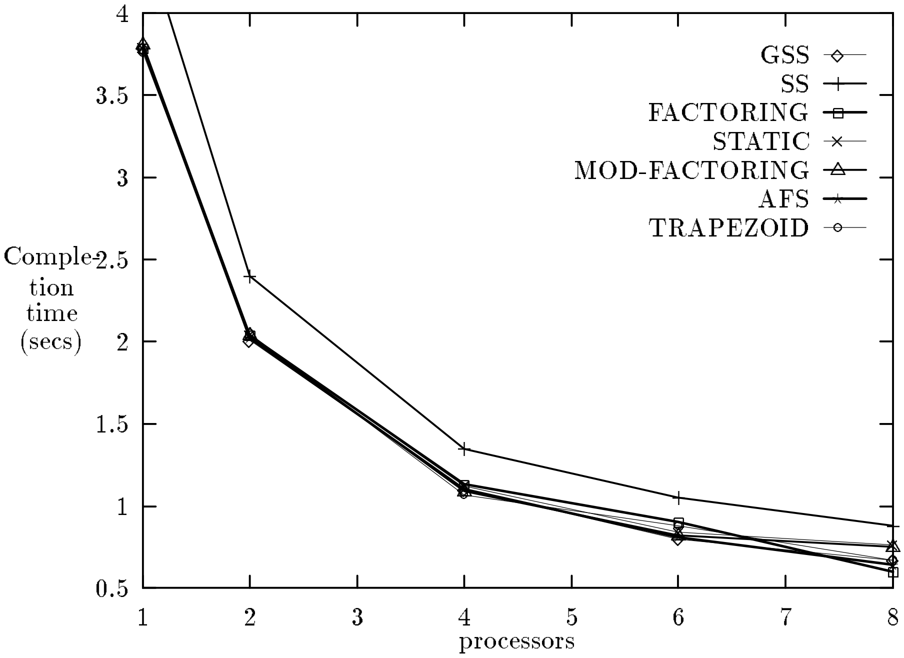

Figure 9: Performance of loop scheduling algorithms for application L4.

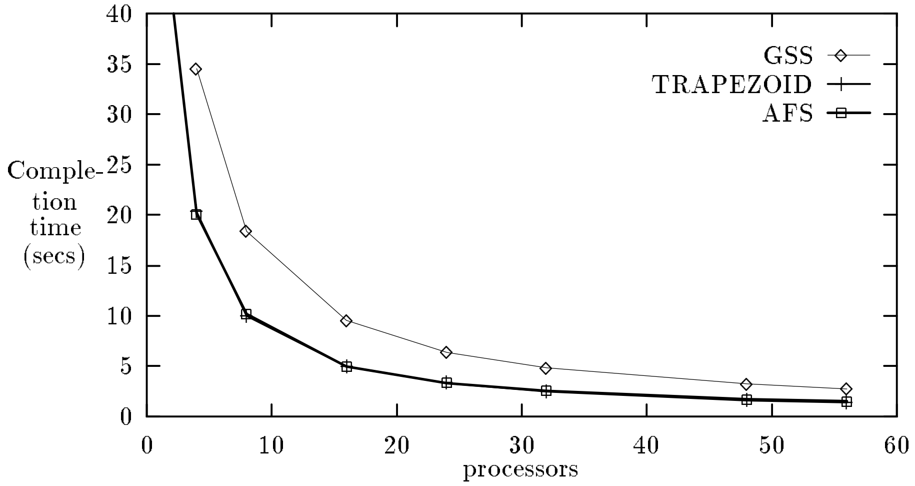

Figure 10: Performance of loop scheduling algorithms on the Butterfly under triangular workload.

4.4 Effects of Load Imbalance

In order to explore the effect of load imbalance in isolation, we implemented three dynamic loop scheduling algorithms (AFS, GSS, and TRAPEZOID) by hand on the BBN Butterfly. The Butterfly is a large-scale NUMA (NonUniform Memory Access) multiprocessor. None of our loop scheduling algorithms on the Butterfly preserve affinity, and even the distributed work queues require non-local access, so any performance differences can be attributed to the load balancing properties of the various algorithms.

We executed three applications on the Butterfly, while progressively introducing more imbalance in the computation. The first application has the following form:

```
DO PARALLEL 19 I = 1, N
   DO SEQUENTIAL 29 J = 1, N-I
      COMPUTE
   29 CONTINUE
19 CONTINUE
```

This application is similar to adjoint convolution, in that the first few iterations of the parallel loop have much more work to do than the last few iterations. Figure 10 plots the performance of the three loop scheduling algorithms for this application, where $N = 5000$. AFS and TRAPEZOID have comparable performance, and both perform better than GSS. The reason for this is given by theorem 3.3, which states that the workload of this application is evenly balanced when processors take $1/(2P)$ of the remaining iterations. TRAPEZOID starts with chunks of exactly that size, while AFS uses smaller chunks, which results in slightly greater synchronization overhead.

Our second application has even greater load imbalance: iteration i takes time proportional to $(N - i)^2$.

```
DO PARALLEL 19 I = 1, N
   DO SEQUENTIAL 29 J = 1, (N-I)**2
      COMPUTE
   29 CONTINUE
19 CONTINUE
```

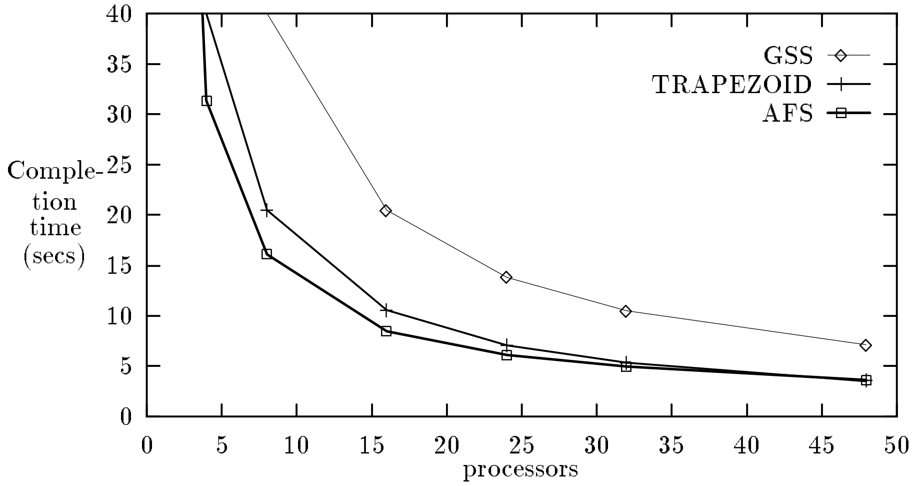

Figure 11: Performance of loop scheduling algorithms on the Butterfly under decreasing parabolic workload.

According to theorem 3.3, each processor should take $1/(3P)$ of the remaining iterations to balance the load evenly. TRAPEZOID allocates chunks larger than that, but smaller than the chunks used by GSS. Therefore, we would expect TRAPEZOID to behave worse than AFS, but better than GSS. Figure 11 plots the results for this program, with $N = 200$. As expected, AFS performs better than TRAPEZOID, which performs better than GSS. Note however that TRAPEZOID is very close to AFS when the number of processors is close to 50 and $N = 200$. Theorem 3.3 explains why: given 50 processors, the first chunk allocated by TRAPEZOID is of size $200/(2 * 50) = 2$ iterations, while the maximum number of iterations that can be allocated without creating imbalance according to theorem 3.3 is $200/(3 * 50) = 1.5$ iterations. Thus, TRAPEZOID is within one iteration of the optimal allocation, which in practice gives performance comparable to AFS.

Our final application has imbalance comparable to that of transitive closure. That is, the first 10% of the iterations take 100 time units to complete, while the remaining 90% of the iterations take one time unit to complete. The code for the application is:

```
DO PARALLEL 19 I = 1,N
    IF (I.LT.(N/10))  THEN
        COMPUTE(100)
    ELSE
        COMPUTE(1)
    ENDIF
19 CONTINUE
```

If the first processor takes more than $1/(10P)$ of the iterations, it will get more than $(1/P)_{th}$ of the work, and will therefore be the last processor to finish. Figure 12 plots the results of executing this program on the Butterfly with $N = 50000$. In this figure, AFS is clearly superior to TRAPEZOID and GSS. Both GSS and TRAPEZOID can be improved, at the expense of synchronization overhead, by starting with smaller chunks of iterations. AFS can afford to start with small chunks of iterations because it uses a distributed work queue, which results in either smaller synchronization overhead for the same load balancing properties, or comparable synchronization overhead for superior load balancing properties.

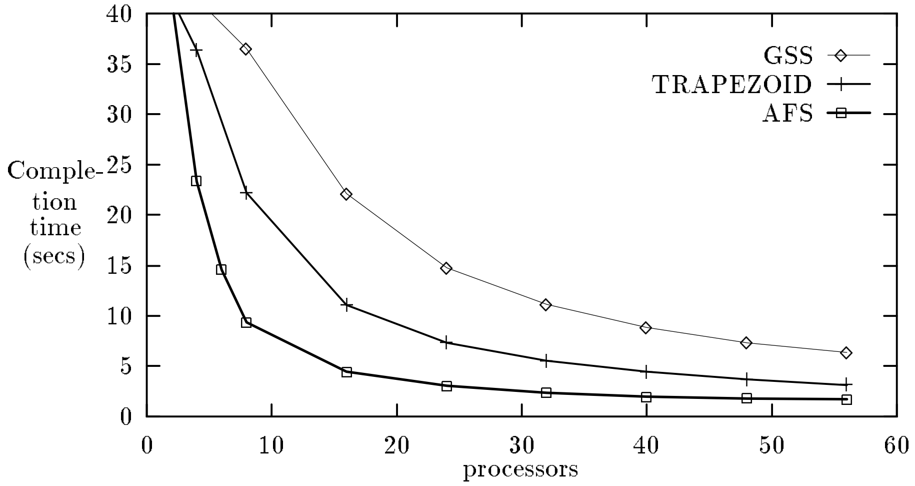

Figure 12: Performance of loop scheduling algorithms on the Butterfly when load is in first 10% of iterations.

4.5 Effect of Processor Arrival Time

In our previous examples, we assumed that all processors begin executing iterations at about the same time. We now focus on the case where not all processors start executing loop iterations at the same time. Static loop scheduling algorithms are clearly inappropriate for this situation; robust dynamic loop scheduling algorithms should be able to distribute the load evenly independent of the starting time of processors.

According to the analysis in section 3, if all iterations take the same time to complete, then under guided self-scheduling, factoring, and affinity scheduling (with $k = P$), all processors finish within one iteration of each other, regardless of the starting time of processors. To confirm this fact experimentally, we implemented a simple, balanced parallel loop with 200 million iterations and no memory accesses on the Iris. Since our loop has no affinity to exploit, the performance differences among the algorithms can be attributed to any load imbalance caused by the non-uniform starting time of processors.

In this set of experiments, all processors start executing loop iterations at the same time, except for one processor, which is delayed for time t_1. We varied the delay and measured the execution time of our simple loop under the different scheduling algorithms. The results appear in table 2. In the table, the delay column represents the number of iterations one processor was delayed. For example, in the first experiment, a processor is delayed for the amount of time it takes one processor to execute one-sixteenth of the iterations. Within this time, the other seven processors can execute $7/16 = 0.43$ of the iterations.

The measured results show that all algorithms perform about the same in the presence of non-uniform starting times for all processors. These results are as expected for GSS, FACTORING and AFS (with $k = P$), because each of these scheduling algorithms guarantees that all processors finish within one iteration of each other. TRAPEZOID also performs close to the best algorithm in each case. Not surprisingly, AFS with $k = 2$ performs worst of all, but even this algorithm is within 10% of the best algorithm.

These experiments suggest that having processors with different arrival times does not affect the performance of good loop scheduling algorithms. The maximum imbalance introduced depends on the size of allocations of iterations relative to the number of remaining iterations. If the remaining iterations are enough to balance the work evenly (as is the case in most loop scheduling algorithms), then different arrival times do not impose any noticeable overhead. The two factors that distinguish the performance of the various loop scheduling algorithms in our experiments are the load imbalance inherent in the computation, and the ability to preserve affinity in the

Delay	GSS	TRAPEZOID	FACTORING	AFS	
				$k=2$	$k=P$
0.0625N	2.31	2.34	2.31	2.42	2.32
0.125N	2.44	2.44	2.45	2.59	2.44
0.1875N	2.53	2.54	2.52	2.82	2.52
0.2031N	2.54	2.57	2.54	2.68	2.54
0.2187N	2.58	2.6	2.58	2.58	2.58
0.25N	2.9	2.9	2.9	2.9	2.9

Table 2: Execution time (in seconds) of simple, balanced loop program with non-uniform start times.

scheduler.

4.6 Synchronization Overhead

In this section we focus on the synchronization overhead imposed by each scheduling algorithm, so as to verify experimentally our analytic results, and to quantify the synchronization overhead incurred by our application suite. Our metric for synchronization overhead is the number of times a processor removes iterations from a work queue. The time required to remove iterations from a work queue might be a more accurate metric, but we are primarily interested in the number of synchronization operations required by each algorithm, and not the implementation details of a particular algorithm on a particular machine.

Every algorithm except affinity scheduling uses a central work queue, wherein each access to the work queue is a global synchronization operation. For affinity scheduling, we identify separately the number of operations performed on local work queues from the operations performed on remote work queues. We note however that on many architectures, operations on remote queues under affinity scheduling would be cheaper than global synchronization operations on a central work queue, since there is less contention for access to each distributed work queue under affinity scheduling.

We should also note that load imbalance does not affect the number of synchronization operations performed by SS, GSS, FACTORING and TRAPEZOID. Load imbalance does affect the number of synchronization operations performed by AFS however, because AFS responds to imbalance dynamically by migrating iterations. Thus, the number of remote synchronization operations performed by AFS will give us insight into the migration overhead incurred by the algorithm.

We will use SOR as an example of a well-balanced application, and adjoint-convolution and transitive closure (with a skewed input) as examples of applications with considerable variance in computation times across iterations. In the transitive closure example, all the work is contained in the first half of the iterations, while in the adjoint convolution example, the computation times of the iterations are linearly decreasing.

Table 3 shows the number of synchronization operations per loop incurred by SOR under the various scheduling algorithms. In our example, there are 512 iterations per loop, so self-scheduling (SS) induces exactly 512 synchronization operations, regardless of the number of processors. TRAPEZOID requires the smallest number of synchronization operations, followed by GSS and FACTORING. As expected, AFS requires a very small number of costly remote synchronization operations, and induces about as many local synchronization operations per queue as TRAPEZOID.

Note that all of the entries in table 3 represent an integer number of synchronization operations, except the entries for affinity scheduling. There are two reasons for this. First, the individual entries in the table for affinity scheduling represent an average across all processors for local and remote synchronization operations. Second, repeated executions of the same parallel loop under affinity scheduling do not always require the same number of local and remote synchronization operations.

Table 4 shows the number of synchronization operations per loop incurred by transitive closure (with a skewed input matrix) under the various scheduling algorithms. Once again, SS induces a large number of synchronization operations independently of the number of processors. TRAPEZOID requires the fewest synchronization

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	512	1	10	3	0	1
2	512	10	18	7	0.5	7.3
4	512	23	32	13	1.0	16.8
6	512	33	50	16	1.1	21.8
8	512	43	56	27	0.4	27

Table 3: Number of synchronization operations for SOR (N= 512).

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	640	1	11	3	0	1
2	640	11	20	7	1.5	8.4
4	640	23	36	13	2.1	16.8
6	640	34	52	18	1.1	23.8
8	640	45	64	22	0.7	28.3

Table 4: Number of synchronization operations for transitive closure on a skewed 640-node graph.

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	5625	1	14	3	0	1
2	5625	14	29	7	4	11
4	5625	31	49	14	6.75	20.2
6	5625	46	69	21	8.3	29.3
8	5625	61	89	28	9.87	35.6

Table 5: Number of synchronization operations for adjoint convolution N = 75.

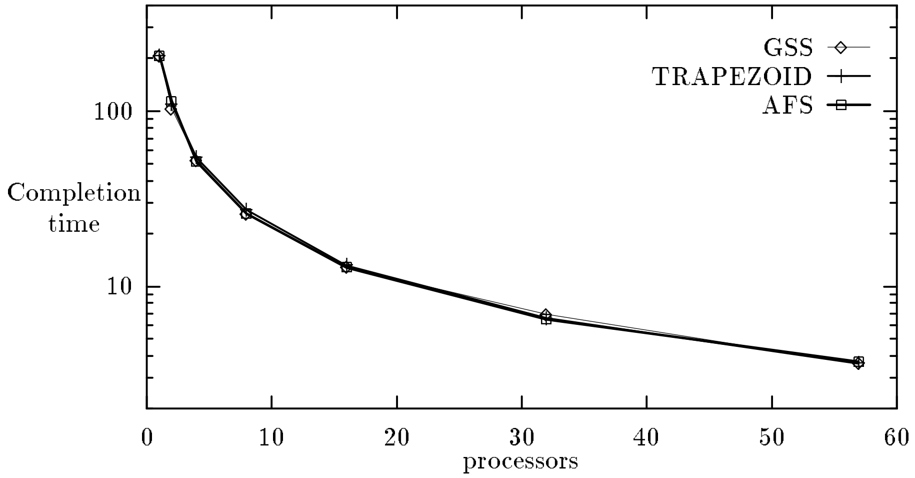

Figure 13: Performance of loop scheduling algorithms on the BBN Butterfly.

operations, but is only slightly better than AFS, which requires very few operations on remote queues.

Even though the input matrix causes a large load imbalance in this application, AFS requires only one or two remote synchronization operations per work queue to balance the load. Each processor accesses a local work queue 20-30 times on average, but rarely accesses a remote work queue. Whereas traditional loop scheduling algorithms always access non-local work queues, AFS accesses a non-local work queue only 5-10% of the time, and yet balances the load just as well. On machines where access to a local work queue is much cheaper than access to a remote work queue (either due to the cost of non-local access or the cost of non-local synchronization primitives), this property of affinity scheduling could have enormous performance advantages.

Table 5 presents the total number of synchronization operations for the adjoint convolution application under the various scheduling algorithms. TRAPEZOID again has the smallest number of synchronization operations. Although AFS does more synchronization operations than TRAPEZOID, the additional overhead is not noticeable, because synchronization is relatively inexpensive on the Iris multiprocessor, and because the number of processors we used in our experiments was rather small. In both cases synchronization was less than 1% of the execution time, so any small savings in the number of synchronization operations would have almost no impact on total execution time.

To confirm that synchronization overhead is not an important factor in the comparative performance of loop scheduling algorithms on shared-memory multiprocessors, we implemented a simple, balanced parallel loop on the Butterfly. In our implementation of affinity scheduling on the Butterfly, all work queues require non-local access. Since our loop has no affinity to exploit, the performance differences among the algorithms can be attributed to synchronization overhead. The results appear in Figure 13. As can be seen from the figure, GSS, TRAPEZOID, and AFS have comparable performance when the effects of affinity scheduling, distributed work queues, and load imbalance are factored out.

4.7 Summary of Results

Our experimental results demonstrate that the affinity scheduling algorithm has load balancing properties comparable to those of the best known loop scheduling algorithms (i.e., guided self-scheduling, trapezoid, factoring), while maintaining processor affinity, and thereby significantly reducing communication overhead. The number of synchronization operations per queue required by affinity scheduling is not much larger than the number of operations required by the trapezoid algorithm, which induces the least amount of synchronization overhead of the dynamic algorithms. Moreover, the number of *serialized* synchronization operations induced by affinity scheduling is always less than the number of *serialized* synchronization operations required by the other dynamic methods. As a result, in most cases affinity scheduling performs better than any other known algorithm on modern shared-memory multiprocessors.

5 Scaling Communication Costs, Processors, and Problem Size

In this section we consider the performance effects of affinity scheduling as we increase the relative cost of communication, the number of processors, or the problem size.

5.1 Increasing the Cost of Communication

Our experiments on the Iris confirm that communication overhead is a dominant factor in application performance on modern shared-memory multiprocessors. Why then do so many of the known loop scheduling algorithms ignore communication overhead? The answer lies in the changes in hardware that have occurred over the last few years. RISC technology and floating point co-processors have increased the speed of computation dramatically, while memory and interconnection network speeds have improved only modestly. To quantify this trend more lets examine the communication and computation costs in the BBN Butterfly family of large scale multiprocessors: The first member of the family, the BBN Butterfly I was introduced in 1981. BBN I runs at 8 MHz. Its processors communicate via a butterfly switch that delivers up to 4MBytes per second to each processor. The non-local access cost on BBN I is 7 μs. Five years later, the second member of the family was introduced: BBN Butterfly Plus. The Plus runs at 16MHz, and is equipped with a floating-point coprocessor. Our experiments indicate that the BBN Plus is about 8 times faster than the BBN I [21]. The bandwidth and latency of the switch remained the same as the BBN I. In 1989 the latest member of the family was introduced: The BBN TC2000. It is equipped with a RISC (88000) processor that runs at 20MHz. Our experiments indicate that the TC2000 is about 60 times faster than the BBN I. TC-2000 processors are connected via a Butterfly switch that delivers up to 10 Mbytes/sec to each processor. The cost of a non-local memory access ranges from 1.9 μs to 6.17 μs [2]. When we compare the computation and communication improvement from BBN I to BBN TC2000, we see that computation improves at a much faster rate than communication. Computation on the TC2000 has improved by a factor of 60 when compared to BBN I, while non-local memory access latency has improved by at most a factor of $7/1.9 = 3.6$, and switch bandwidth has improved by at most a factor of $10/4 = 2.5$. Similar trends also hold for other multiprocessors [21] as well.

This trend in multiprocessor architecture shifts the emphasis from computation costs to communication costs, since an ever-increasing percentage of an application's execution time is devoted to communication.

In order to demonstrate this trend, we executed our Gaussian elimination program on a Sequent Symmetry S81 multiprocessor, a bus-based, cache-coherent machine that predates the Iris. The processors on the Iris are about 30 times faster than the processors on the Symmetry, but the peak bandwidth of the Symmetry bus is 80 MB/sec, while the peak bandwidth of the Iris bus is only 64 MB/sec. Figure 14 plots the execution time of Gaussian elimination on a 256 by 256 matrix under three dynamic loop scheduling algorithms on the Symmetry. From this figure we can see that AFS and GSS are comparable in performance on the Symmetry, while our earlier results showed that AFS clearly dominates GSS on the Iris. We can conclude that the ability of AFS to exploit processor affinity in Gaussian elimination is of little value on the Symmetry, since communication is cheap relative to computation.

We also see in figure 14 that TRAPEZOID performs 10-15% worse than both AFS and GSS on this application. The cause of this disparity can be traced to the load balancing properties of TRAPEZOID. When all iterations take the same time to execute, processors finish within one iteration of each other under guided self-scheduling [24]. Under TRAPEZOID, processors finish within several iterations of each other [31]. When an iteration takes a long time to complete, the imbalance introduced by the trapezoid algorithm can be noticeable. Although the trapezoid algorithm requires fewer accesses to the work queue, the Sequent does not employ a large number of processors, and therefore the low synchronization overhead of TRAPEZOID does not outweigh the load imbalance it causes.

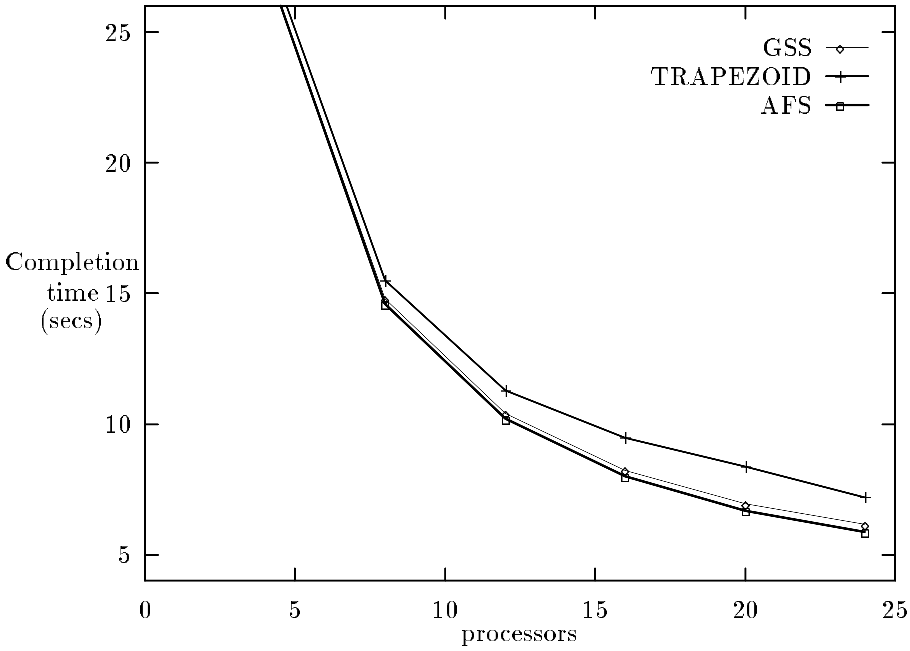

Figure 14: Gaussian elimination on the Sequent Symmetry.

These results suggest that communication was a relatively minor issue on the previous generation of shared-memory multiprocessors, and that both load imbalance and synchronization overhead were dominant. Our results on the Iris argue that the situation has changed dramatically, so much so that communication is now the dominant factor in performance. The following architectural trends suggest that the situation will not change in the near future.

- As processor speeds continue to improve at a higher rate than either memory or interconnection network speeds, the overhead of communication will increase even more.
- The discrepancy in processor and memory speeds is already high enough that a breakthrough in memory and interconnection network technology *without a corresponding breakthrough in processor technology* would be required to reduce the significance of communication in parallel applications.
- The ratio of communication costs to processor speeds are likely to increase given the trend towards scalable multiprocessors consisting of many small-scale, bus-based nodes connected by a scalable interconnection network.

In short, communication is a dominant factor in modern multiprocessors and there is no indication that the situation will change in the foreseeable future. Any scheme designed to reduce communication overhead, such as affinity scheduling, will produce ever-greater returns as long as current trends continue.

5.2 Scaling the Number of Processors

To demonstrate the importance of affinity scheduling on recent large-scale multiprocessors, we performed several experiments on the KSR-1, a large-scale, cache-coherent multiprocessor released in 1992. These experiments used those applications that have locality worth preserving, including Gaussian elimination, SOR, and transitive closure.

Figure 15 presents the completion time of Gaussian elimination (using a 1024 by 1024 matrix) under various loop scheduling algorithms on the KSR-1. In this figure we see that, once again, AFS performs best. It improves the completion time of the application by a factor of 3.7 when compared to FACTORING and GSS, and by a factor of 2.8 compared to TRAPEZOID. The reason that TRAPEZOID performs better than FACTORING and GSS is that TRAPEZOID has the fewest number of synchronization operations, and synchronization is relatively expensive on the KSR.

MOD-FACTORING exhibits interesting behavior in figure 15. It performs reasonably well on a small number of processors, falling somewhere between AFS and TRAPEZOID. As the number of processors grows beyond 12 to 15, the performance of MOD-FACTORING approaches that of FACTORING. The reason that MOD-FACTORING performs so poorly with an increase in processors is that a large number of processors can easily introduce small amounts of load imbalance. These short term fluctuations cause processors to execute the iterations of other processors, destroying almost all affinity.

Figure 16 shows the completion time of transitive closure (1024 node graph, where 40% of the nodes form a clique) under the different loop scheduling algorithms on the KSR-1. From this figure we can easily see the importance of affinity scheduling; the other dynamic scheduling algorithms cannot exploit more than 12 processors. After AFS, the next best algorithm is TRAPEZOID, which has the smallest number of synchronization operations, and therefore manages to degrade more gracefully than the other algorithms. Although AFS performs the best, the improvement over the other algorithms is not as great as it was for Gaussian elimination. There is almost no load imbalance in Gaussian elimination, and hence no need to destroy any affinity, whereas transitive closure does have load imbalance and therefore affinity scheduling must reassign iterations fairly frequently.

Figure 17 presents the completion time of SOR (1024 by 1024 matrix and 128 iterations) on the KSR-1. Although AFS, STATIC, and MOD-FACTORING perform the best in this case, they are not much better than the other algorithms, even though SOR has a lot of affinity to preserve, and there is almost no load imbalance to hinder affinity. So why isn't AFS much better than the other algorithms? The reason for this anomaly is that SOR performs a few floating point additions and one floating point division within the inner loop, and floating point division is implemented in software on the KSR-1. Thus, computation in SOR is expensive on the KSR-1, and the benefits of preserving affinity are not significant in comparison.

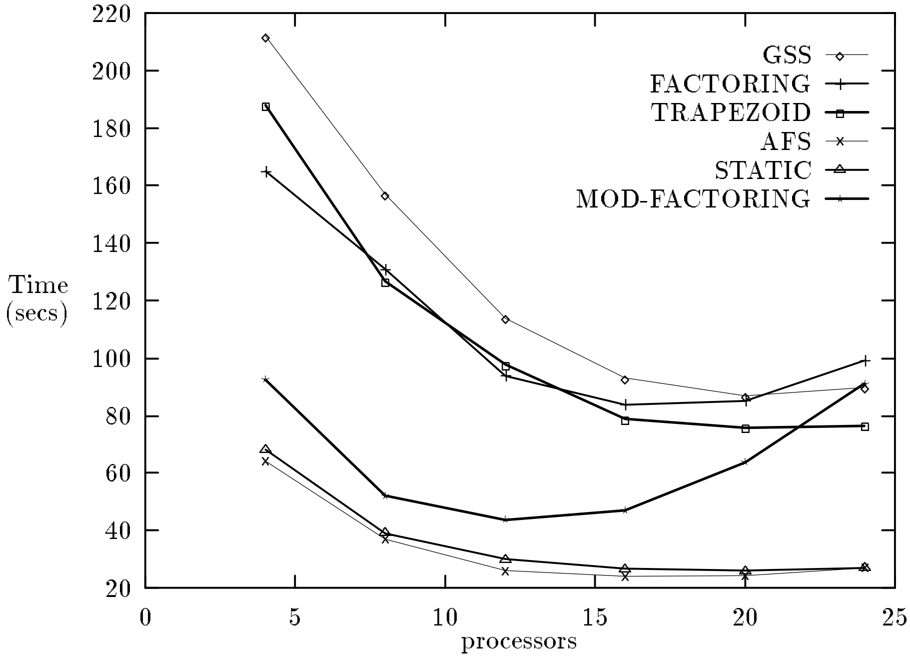

Figure 15: Gaussian elimination on the KSR-1.

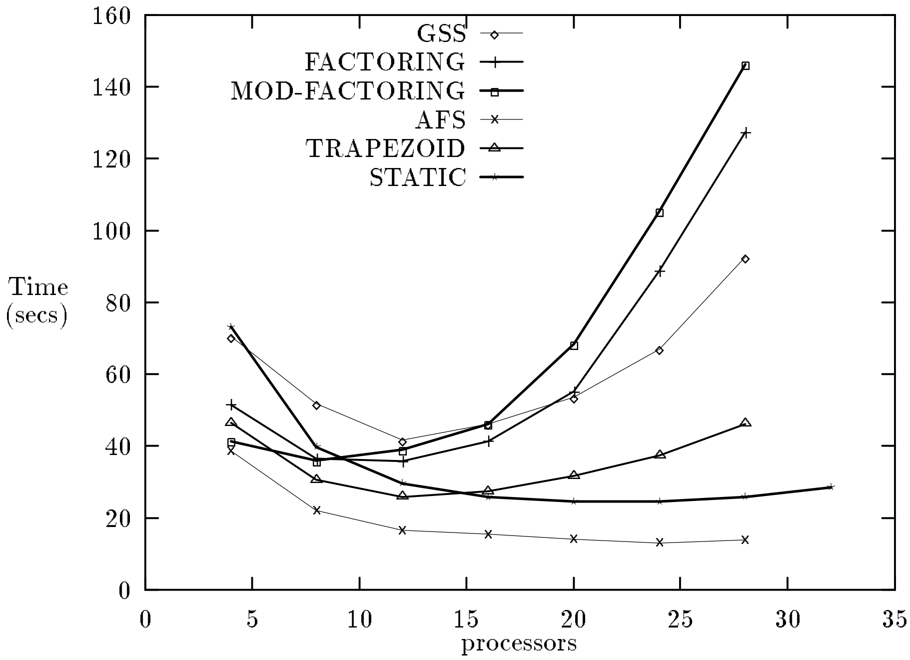

Figure 16: Transitive closure on the KSR-1.

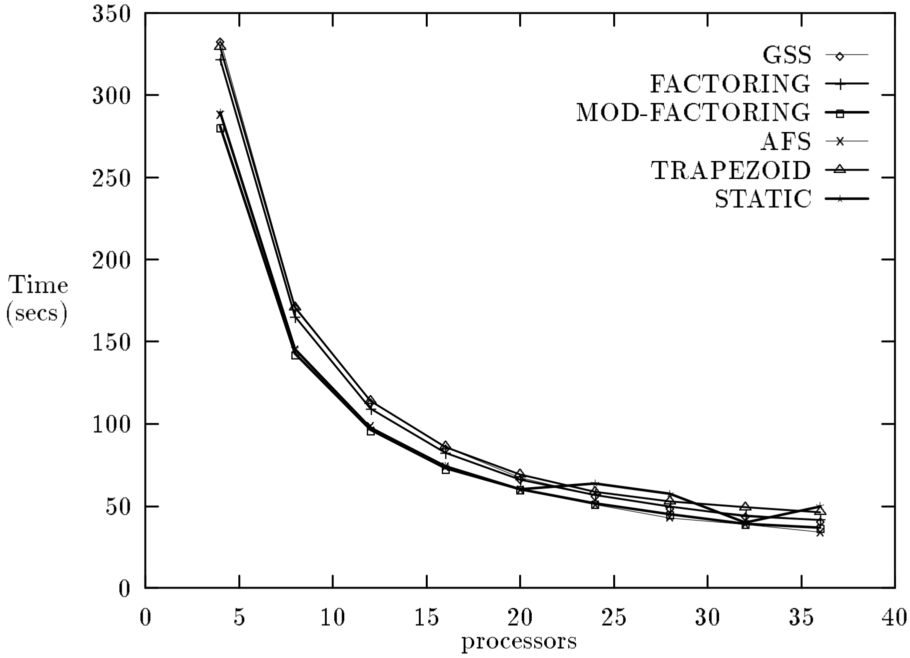

Figure 17: SOR on the KSR-1.

5.3 Scaling the Problem Size

Affinity scheduling performs best when the working set of an application remains in the caches (or local memories) of the multiprocessor. Historically, both caches and local memories were rather small, and could not hold the input for large, scientific problems. The situation has improved dramatically in recent years however, with the size of local memories quadrupling every three years [14]. For example, the SGI Iris has a 1 MB second-level cache, and the KSR-1 has an all-cache memory of 32 MB per processor.

To verify that affinity scheduling results in significant performance improvements even for long-running applications, we ran Gaussian elimination on a 4096×4096 matrix on 16 processors on the KSR-1. This problem needs more than twenty minutes to complete using 16 processors on the KSR-1, about five hours to complete on one KSR-1 processor, and about 25 days to complete on one Butterfly I processor. The completion time of this problem on the KSR-1 under various loop scheduling algorithms is in the following table. We can see that even for a problem of this size, affinity scheduling results in significant performance improvements over FACTORING, TRAPEZOID, and GSS.

scheduling algorithm	completion time (minutes)
AFS	20.6
STATIC	20.9
MODIFIED-FACTORING	22.7
FACTORING	47.3
TRAPEZOID	50.7
GSS	73.7

Of course, not all problems will fit in the caches or memories of the machine, and not all machines have 32 MB of local storage for each processor. However, in those cases where affinity exists, the loop scheduling algorithm should exploit this fact.

6 Related Issues

Loop scheduling can be viewed as part of the general problem of scheduling tasks in multiprocessor systems so as to minimize the completion time of parallel applications. In this context, loop scheduling is analogous to process scheduling, which has also been studied extensively. Process scheduling is concerned with many of the same issues involved in loop scheduling, including concerns about load imbalance, synchronization overhead, and communication overhead.

In many shared-memory multiprocessor systems, a single ready queue is the primary mechanism for process scheduling [30, 29, 10, 3]. The attractive load balancing properties of a central ready queue address an important concern in these systems that a processor not remain idle while there is work to be performed. Recent work [1] has shown that a central ready queue can become a bottleneck in these systems, and that local per-processor ready queues can eliminate contention for the queue, thereby reducing synchronization overhead.

Other recent work in process scheduling has considered the overhead associated with reloading the cache on each context switch when a multiprocessor is multiprogrammed. Squillante and Lazowska [26] showed that a process develops an affinity for a particular processor during execution based on the contents of the local cache. They argued that if a process suspends execution for any reason, it should be resumed on the same processor, avoiding migration whenever possible. They showed that ignoring affinity can result in significant performance degradation.

In subsequent work, both Gupta *et al* [12] and Vaswani and Zahorjan [32] argued that ignoring affinity does not significantly degrade performance. Gupta *et al* showed that cache affinity scheduling produces no more than a 3-4% improvement in application performance when the scheduling quantum is reasonably large and applications perform I/O at normal rates. Vaswani and Zahorjan reached a similar conclusion by showing that the time to reload the cache is small relative to the frequency of context switching. The reason for this apparent contradiction is that Squillante and Lazowska assume a time-sharing kernel policy, where both the time between processor reallocations and the amount of cache corruption between allocations is small, whereas both Gupta *et al* and Vaswani and Zahorjan assume a space-sharing policy with relatively infrequent reallocation of processors.

In an earlier paper [20] we considered the role of cache affinity scheduling for threads *within* an application, rather than in the context of multiprogramming. We showed that using threads to represent fine-grain parallelism can introduce excessive communication overhead, because each thread spends a large percentage of its lifetime bringing the data it needs into the local memory or cache. To reduce the overhead associated with fine-grain threads, we proposed a scheduling policy (*memory-conscious scheduling*) that places threads close to their data. This policy improves performance significantly on the Butterfly, and more dramatically on the SGI Iris, which is consistent with the results presented here.

While some of the results regarding process scheduling apply to loop scheduling, there is an important distinction between the two problem domains: a loop scheduling algorithm must choose an appropriate decomposition (i.e., chunks), while the process scheduling algorithm is given the decomposition selected by the programmer as input. Thus, loop scheduling considers an additional dimension — loop decomposition so as to minimize load imbalance — and therefore must make tradeoffs in three dimensions (synchronization overhead, communication overhead, load imbalance). It is this third dimension that distinguishes work in loop scheduling, and that has dominated loop scheduling research.

When the decomposition is straightforward, the loop scheduling problem reduces to the process scheduling problem. However, communication has been largely ignored as a significant source of overhead in the process scheduling problem. In those cases where the cost of communication has been considered, the scheduler needs detailed information about the application, like the call graph, communication patterns, and exact communication costs [23, 4]. This knowledge is not generally available, as it may depend on unpredictable run-time factors, such as the input values.

Finally, there is an important aspect to loop scheduling we have not considered: scheduling loops that have dependencies within the statements of a single iteration, across iterations, or both [23, 18]. The difficulties in scheduling this type of loop lead to challenging graph-theoretic problems, whose general form is intractable and requires heuristic solutions. Previous approaches to this problem are based on static scheduling, and therefore inherently exploit affinity, unlike dynamic scheduling algorithms.

7 Conclusions

In this paper we argued that the non-uniform access time to data in a shared-memory multiprocessor (due to local caches or memory) introduces a new dimension to the loop scheduling problem: communication overhead. We showed that traditional loop scheduling algorithms, which emphasize load imbalance and synchronization overhead while ignoring communication overhead, impose a significant performance penalty on parallel applications. We described a new loop scheduling algorithm, called affinity scheduling, which reduces communication overhead by exploiting processor affinity. This new algorithm performed better than all other known algorithms in our experiments. The main reasons for this are:

- When a parallel loop is embedded within a sequential loop (a common case), affinity scheduling assigns an iteration of the parallel loop to the same processor each time it is executed. If the iteration accesses the same data each time, then the data will already be in the local memory or cache, reducing communication overhead.

- Affinity scheduling uses per-processor work queues, instead of a central work queue. Accesses to several work queues may proceed in parallel, and most accesses to work queues are local accesses that do not suffer from contention. Synchronization across processor occurs only if load imbalance arises.

Based on our experiments with affinity scheduling and other loop scheduling algorithms on three different multiprocessors, we conclude:

- *Central work queues are an inappropriate scheduling mechanism even for small-scale multiprocessors.* Central work queues (or ready queues) have been criticized for serializing access to work, which can produce a bottleneck in large-scale systems. Efficient synchronization primitives (e.g., fetch and ϕ) and efficient chunking algorithms can help with synchronization overhead, but the problem of communication overhead remains. Central work queues *require the frequent movement of data among processors*, since every process must load the data it needs into the local cache. The resulting communication overhead degrades performance even for a very small number of processors.

- *Loop scheduling algorithms must consider communication as an important source of overhead.* Algorithms that ignore communication incur a significant performance penalty in current multiprocessors. If processor speeds continue to improve more quickly than memory or interconnection speeds, communication will be an increasing percentage of an application's execution time; scheduling methods that reduce both communication and synchronization overhead are going to have an even greater impact in the future.
- *Affinity scheduling simultaneously balances overhead due to synchronization, communication, and load imbalance.* Affinity scheduling has the load balancing properties of the best dynamic scheduling algorithms, reduces synchronization costs by employing per-processor work queues, and exploits processor affinity when it exists.
- *Affinity scheduling is robust.* Our experiments cover a range of applications with widely varying characteristics. For applications that create affinity between iterations and processors, affinity scheduling is by far the best algorithm. For applications with a lot of input-dependent load imbalance (e.g., transitive closure), affinity scheduling was again the best scheduling algorithm. Even for applications that had no affinity to exploit, but exhibited significant potential for load imbalance (e.g., adjoint convolution and L4), affinity scheduling was among the best algorithms.

In summary, our theoretical and experimental evaluation shows that affinity scheduling has the attractive load balancing properties of the best known loop scheduling algorithms, but also reduces communication overhead substantially. This overhead is quite high on current multiprocessors, and is likely to increase in the future. We conclude that loop scheduling techniques, such as affinity scheduling, that minimize communication overhead will be increasingly important in the future.

Acknowledgements

We would like to thank Argonne National Laboratory for allowing us to use their Sequent Symmetry, and Donna Bergmark and the Cornell Theory Center for assistance with and use of their KSR-1. This research was supported by the National Science Foundation under grants CDA-8822724 and CCR-9005633, and the Office of Naval Research Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930).

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessorst. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] BBN Advanced Computers Inc. *Inside the TC2000^{TM} Computer.* Cambridge, Massachusetts, February 1990.

[3] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D.B. Wagner. An Open Environment for Building Parallel Programming Systemst. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[4] S. H. Bokhari. *Assignment problems in parallel and distributed computing.* Kluwer Academic Publishers, Boston, 1987.

[5] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecturet. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[6] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple But Effective Techniques for NUMA Memory Managementt. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 19–31, Dec 1989.

[7] A.L. Cox and R.J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUMt. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 32–44, Dec 1989.

[8] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessorst. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, Dallas, Texas, December 1991.

[9] S. Dandamudi. A Comparison of Task Scheduling Strategies for Multiprocessor Systemst. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 423–426, Dallas, Texas, December 1991.

[10] T. W. Doeppner Jr. Threads: A System for the Support of Concurrent Programmingt. Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[11] D. L. Eager and J. Zahorjan. Adaptive Guided Self-Schedulingt. Technical Report 92-01-01, Department of Computer Science and Engineering, University of Washington, January 1992.

[12] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applicationst. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991.

[13] R. Gupta. Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systemst. In *1989 Interantional Conference on Parallel Processing*, pages II:23–30, 1989.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[15] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loopst. *Communications of the ACM*, 35(8):90–101, August 1992. Previous version appeared in Supecomputing 91, pages 610-619.

[16] C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processorst. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[17] R. P. LaRowe, Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessorst. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[18] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, Englewood Cliffs, NJ, 1992.

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

[20] E. P. Markatos and T. J. LeBlanc. Load Balancing Versus Locality Management in Shared-Memory Multiprocessorst. In *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992.

[21] E.P. Markatos and T.J. LeBlanc. Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performancet. Technical Report 420, University of Rochester, Computer Science Department, March 1992.

[22] C. McCann, R. Vaswani, and J. Zahorjan. May 1993.

[23] C. D. Polychronopolous. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, MA, 1988.

[24] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputerst. *IEEE Transactions on Computers*, C-36(12), December 1987.

[25] B. Smith. Architecture and Applications of the HEP Computer Systemt. In *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981.

[26] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Schedulingt. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[27] M. S. Squillante and R. D. Nelson. Analysis of Task Migration in Shared-Memory Multiprocessor Schedulingt. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 143–155, May 1991.

[28] P. Tang and P.-C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loopst. In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[29] R.H. Thomas and W. Crowther. The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processorst. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[30] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessorst. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[31] T.H. Tzen and L.M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computerst. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[32] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessorst. In *Proceedings of the Thirteenth Sumposium on Operating Systems Principles*, pages 26–40, Pacific Grove, CA, October 1991.