Affinity Scheduling of Unbalanced Workloads*

Srikant Subramaniam† and Derek L. Eager
Department of Computational Science
University of Saskatchewan
Saskatoon, SK S7N 0W0
Canada.

Abstract

Scheduling in a shared memory multiprocessor is often complicated by the fact that a unit of work may be processed more efficiently on one processor than on any other, due to factors such as the presence of required data in a local cache. The unit of work is said to have an "affinity" for the given processor, in such a case. The scheduling issue that has to be considered is the tradeoff between the goals of respecting processor affinities (so as to obtain improved efficiencies in execution) and of dynamically assigning each unit of work to whichever processor happens to be, at the time, least loaded (so as to obtain better load balance and decreased processor idle times).

A specific context in which the above scheduling issue arises is that of shared memory multiprocessors with large, per-processor caches or cached main memories. The shared-memory programming paradigm of such machines permits the dynamic scheduling of work. The data required by a unit of work may, however, often reside mostly in the cache of one particular processor, to which that unit of work thus has affinity.

In this paper, two new "affinity scheduling" algorithms are proposed for a context in which the units of work have widely varying execution times. An experimental study of these algorithms finds them to perform well in this context.

1 Introduction

The performance of a parallel program on a shared memory multiprocessor is critically dependent upon the amounts of overhead that arise from the following four sources: *synchronization, process management, communication,* and *load imbalance* [8].

*This research was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

†Currently at Sybase Inc., Emeryville, CA 94608.

Synchronization occurs when a processor must wait for some action by another processor, such as relinquishing a critical region, or completing some unit of computation. Such waiting may increase program execution time, not only owing to the wasted time itself, but also owing to non-productive usage of perhaps scarce, shared resources (such as bus cycles), while waiting (for example, while checking whether or not the required action has yet occurred). While synchronization can be a dominant source of overhead, at least that portion of the overhead owing to non-productive usage of shared resources can often be eliminated to a large extent through the use of appropriate scalable synchronization primitives and algorithms [1] [10].

Process management overhead refers to the time required to create, destroy and schedule multiple units of sequential execution. Although operating system processes are expensive (as operating systems are slow), they may be used sparingly (typically one process per processor), relying instead on light-weight "user level" threads to express concurrency [2].

Communication overhead is introduced by interaction between processes. Communication manifests itself as cache misses in multiprocessors with caches, as non-local memory accesses (and also possibly cache misses) in machines that support a shared memory programming model, yet in which the shared memory is physically distributed, and as the sending or receiving of messages in a distributed-memory machine. Locality management policies attempt to minimize communication by allocating work close to its data.

Load imbalance occurs when some processors are idle and yet there is work ready to be performed that no processor has yet started. There are two distinct sources of load imbalance in a parallel program: (1) an uneven assignment of computation to units of work, and (2) an uneven assignment of units of work to processors. A "fine grain" decomposition (one in which the units of work are very small) minimizes the effects of variance owing to the uneven assignment of computation to units of work. To avoid an uneven assignment of units of work to processors, many shared-memory programming systems use a central work queue from which either idle processors remove light-weight threads (the central queue is a "ready queue" in this case), or idle worker processes remove units of work [14] [15]. A central queue facilitates a dynamic, even distribution of load among processors, and ensures that no processor remains idle while there is work to be done.

Although both load balancing and locality management policies attempt to improve the performance of the system, conflicts can arise between them. Load balancing attempts to distribute the load evenly, thus keeping all the processors busy. Locality management attempts to allocate work close to its data. Once a program begins execution, a load balancing strategy might be eager to shift work from one processor to another so as to alleviate any imbalance as it occurs. A locality management strategy on the other hand might not favor this migration of work, since the work may have an affinity (owing to the data it requires residing in cache or local memory) for the processor on which it has been allocated. As a result, attempts to evenly distribute the workload can adversely affect locality, while attempts to improve locality of reference can create an imbalance in the workload.

This paper concerns the scheduling of units of work within a single application, when these units of work have "affinities" for particular processors (owing to the locations of the data they access). A particular approach to scheduling termed *affinity scheduling* [9], in which both affinity and load balancing considerations play major roles, is considered. Affinity scheduling is studied in the context of a "one process per processor" program structure in which the units of work being scheduled are the iterations of a program loop that has been parallelized, and in which this loop is repetitively executed. This is a common shared-memory program structure in practice. Integrating both affinity and load balancing considerations in scheduling has also been considered elsewhere in the context of object-oriented systems [3], and in the operating system rather than at the user level [17].

A context of "unbalanced" workloads is considered, in which the parallel loop being executed is severely unbalanced with respect to the amounts of computation represented by each iteration, a situation in which simple forms of affinity scheduling may become inappropriate. Two new loop scheduling algorithms, termed here *dynamic* and *wrapped*, are proposed. An experimental study finds that these new algorithms achieve both the objectives of maintaining affinity, and load balance, even for severely unbalanced workloads.

Section 2 overviews the key issues pertaining to loop scheduling in shared memory multiprocessor systems. Section 3 introduces two new affinity scheduling algorithms. Section 4 outlines the experimental study of affinity scheduling algorithms that was performed in this research; while Section 5 presents and analyzes the results obtained. Finally, the major conclusions are summarized in Section 6.

2 Loop Scheduling

Loops are the single largest source of parallelism in most programs. Executing the many iterations of a loop in parallel on different processors enables an application to exploit parallelism. The iterations of a loop can be scheduled either statically (at compile time) or dynamically (at run-time) onto the processors of a parallel machine. The major advantage of static scheduling is the reduction in the run-time synchronization and scheduling overhead. Since dynamic scheduling defers the assignment of iterations to processors until run-time, better load balancing in the presence of factors unpredictable at compile time is achieved. The major difficulty in designing dynamic loop scheduling algorithms is in keeping the run-time synchronization and scheduling overhead small, without losing the attractive load balancing properties. In this paper, we restrict our attention to dynamic scheduling methods, specifically *self-scheduling* methods in which each processor is itself responsible for determining what work to do next.

There are two distinct classes of self scheduling algorithms: *fixed* and *variable*. In the fixed method [7], each processor retrieves fixed size chunks of k loop iterations, whenever it becomes idle. In "pure self scheduling", loop iterations are retrieved one at a time ($k = 1$). This is inappropriate for fine grained loops, as it can result in excessive overhead if each iteration represents only a small amount of work. At the other extreme, each processor obtains $\frac{1}{P}$ of the loop iterations in a single chunk, where P denotes the number of processors. This strategy fails to meet the goal of robustness and flexibility as it does not allow for variability in the number of processors available to the application (during loop execution), nor for loops that are very unbalanced with respect to the work present in each region of the loop. In general, choosing an appropriate value of k is quite difficult, and an optimal choice would require detailed information about the loop and the machine environment that may be difficult to obtain in practice.

In variable self scheduling algorithms [4] [6] [11] [16], processors initially obtain large chunks but take increasingly smaller chunks as loop iterations become depleted. By taking many iterations in each initial chunk, these methods offer the possibility of low overhead since relatively few scheduling operations are required. By taking small final chunks, these methods promise good load balance, since these small chunks can be used to even out the load imbalances incurred in the execution of the initial chunks.

An additional important factor that has to be considered in the assignment of iterations to processors is the location of data that is accessed by these iterations. The main motivation for incorporating affinity considerations in loop scheduling is to reduce the time spent in bringing data into the local memory or cache. This can be a significant source of overhead, and it has been shown that it may consume 30-60% of the total execution time in this context [8]. By scheduling loop iterations on processors whose local memories or caches already contain the necessary data, the communication time can be significantly reduced.

One important context in which loop iterations have an affinity for a particular processor is when:

1. the same data has to be repeatedly used by successive executions of an iteration (as in the case of an inner parallel loop within a outer sequential loop), and,

2. the data is not removed from the local memory (or cache) before it can be reused.

An affinity scheduling algorithm of specific interest is the one proposed by Markatos and LeBlanc [9]. This loop scheduling algorithm simultaneously attempts to balance the workload, minimize synchronization, and co-locate loop iterations with the necessary data. The algorithm is applicable for the specific scenario of a parallel loop embedded within an outer sequential loop, the context that is also assumed in this paper. The algorithm of Markatos and LeBlanc divides the iterations of a loop into partitions of size $\frac{N}{P}$ iterations, where N is the number of iterations in the loop, and P is the number of available processors, as shown in the pseudocode of Figure 1 (where the algorithm is referred to as the "static partitioned affinity scheduling " algorithm). The assignment of partitions to processors is done in the *assign_iterations* procedure, and results in the initial values for per-processor data values *lower* and *upper* giving the loop index of the next iteration that should be handed out, and the upper limit of the loop partition, respectively. In the loop execution, each processor removes $\frac{1}{P}$ of the iterations from its local work queue (as implemented by *lower* and *upper*) and executes them. If a processor's work queue is empty, it finds the most loaded processor, removes $\frac{1}{P}$ of the iterations from that processor's work queue, and executes them. As described by Markatos and LeBlanc [9], variants of this approach are also possible, wherein fractions of iterations other than $\frac{1}{P}$ are used.

```
loop_initialization(N,P)
// executed by one processor at the beginning
// of the loop. 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) - 1) to
   // processor i
      assign_iterations(i)
   }
}
loop { // executed by each processor
   // get 1/P of the local iterations to
   // execute
   range = get_iterations(myproc, 1/P);
   if (range == empty) break;
   execute (range);
} forever
loop { // executed by each processor
   // find the most loaded processor
   max_load = find_most_loaded_processor();
   // get 1/P of the iterations from the most
   // loaded processor
   range = get_iterations(max_load, 1/P);
   if (range == empty) break;
   execute (range);
} forever
```

Figure 1: Pseudocode for Static Partitioned Affinity Scheduling

3 Proposed Affinity Scheduling Algorithms

This paper proposes two new affinity scheduling algorithms for loops in which the iterations have widely varying execution times. These iteration times may vary in a correlated way; i.e. the i^{th} iteration may require time proportional to some function of i (e.g., a linear function as in a "triangular" iteration space).

In this case, a uniform initial allocation of iterations to all processors may result in a situation where there are lightly loaded processors that finish well ahead of the other processors. In the algorithm proposed by Markatos and LeBlanc [9], as described in Section 2, idle processors probe the work queues of other processors and remove $\frac{1}{P}$ of the iterations from that of the most loaded processor. This migration of work from the most loaded processors to the idle processors results in a cache-reload overhead that is experienced every time the loop is executed, since there is no mechanism to ensure that the same iterations are migrated to the same processors on subsequent executions. It is quite possible that this overhead might nullify to some extent the gains that result from exploiting affinity.

In this paper we propose two algorithms that are robust to even large, correlated imbalances in loop execution times. (The first of the algorithms assumes, however, that the execution times of any particular iteration do not vary widely from one execution of the loop to the next.) The key concept behind the first of the proposed algorithms is the readjustment of the sizes of the allocated partitions on subsequent executions of a loop. The key concept behind the second algorithm is to assign to each processor iterations that are dispersed throughout the iteration space, rather than consecutive iterations, as was proposed in [4] in the context of Guided Self-Scheduling [11].

The two proposed algorithms are described below:

1. **Dynamic Partitioned Affinity Scheduling:** The basic idea in this approach is to keep track of the number of iterations that were actually executed by each processor in the previous execution of the loop. This serves as an indication of the actual load distribution in the loop and is used in the reallocation of the iterations for subsequent executions.

   There are three distinct phases in the proposed algorithm (the first of which is done only once, the remaining two of which are repeated for each loop execution):

   (a) Loop initialization phase: This phase is identical to the initialization phase in Figure 1 where the iterations of a loop are divided into partitions of size $\frac{N}{P}$ iterations; in this policy, however, this is only done for the first execution of the loop.

   (b) Loop execution phase: A processor removes $\frac{1}{P}$ of the iterations from its local work queue and executes them. If a processor's work queue is empty, it finds the most loaded processor, removes $\frac{1}{P}$ of the iterations from this processor and executes them. Every processor i keeps track of the actual number of iterations that it executed in the i^{th} element of an array *executed* (of size P).

   (c) Re-initialization phase: At the beginning of all but the first execution of the loop, a dynamic readjustment of the initial partition size is achieved by calculating the new partition for processor i (assuming processors are indexed 0 through P - 1, and iterations are indexed from 0) as :

   <u>when i = 0</u>
   partition_start[i] = 0;
   partition_end[i] = executed[i] - 1;
   <u>when i > 0</u>
   partition_start[i] = partition_end[i-1] + 1;
   partition_end[i] = partition_start[i] + executed[i] - 1;

   By dynamically changing the sizes of the initial processor partitions, the proposed algorithm is more capable of handling workloads that are unbalanced with respect to the amounts of computation represented by each iteration. This claim assumes that the algorithm "converges" (this must be verified experimentally) in the sense that eventually the initial partitions stabilize to roughly constant ranges, thus yielding good affinity behavior.

2. **Wrapped Partitioned Affinity Scheduling:** Eager and Zahorjan [4] propose the wrapped assignment of iterations to rectify a shortcoming of the Guided Self-Scheduling [11] algorithm; specifically, the load imbalance that arises when consecutive iteration execution times vary widely, but in a correlated manner. The underlying idea behind this wrapped allocation of iterations is to allocate to each processor iterations that are essentially at a distance P (the number of processors in the system) from each other, as opposed to consecutive iterations that are at a unit distance as in blocked allocation. In our application of this idea, partitioning as in the algorithm of Figure 1 is employed, but within a wrapped iteration space.

   Most of the implementation details of this algorithm are identical to those of the static partitioned affinity scheduling approach shown in

Figure 1. The key difference, however, is in the "execute" code. The wrapped assignment is implemented by mapping the consecutive iteration numbers provided to each processor by "get_iterations" into the iteration numbers that correspond to the desired wrapped allocation. For example, for a loop with 12 iterations indexed from 0 to 11, and for 4 processors, the iterations indexed as 0,1,2,3 in the index space used to allocate iterations would have indices 0,4,8,1 in the actual index space. Figure 2 shows sample initial partitions (showing the actual index values) for a loop with 18 iterations indexed from 0 to 17 and for 4 processors, under a wrapped partitioned assignment.

```
processor 1: 2   6   10   14   3
processor 2: 7   11  15   0
processor 3: 4   8   12   16   1
processor 4: 5   9   13   17
```

Figure 2: Sample initial partitions with wrapped partitioned assignment (4 processors, 18 loop iterations)

4 Experimental Study

The two proposed affinity scheduling algorithms (*dynamic* and *wrapped*), together with the static partitioned affinity scheduling algorithm (*static*) [9] and the Guided Self-Scheduling (*GSS*) [11] algorithms were implemented on a Silicon Graphics eight-way multiprocessor and their performance studied. The most important metric used to compare the relative performance is the *application execution time* using the proposed algorithms. The second metric concerns the *scalability* (application run time for bigger problems on bigger systems) of the proposed algorithms. This metric is particularly relevant, given the rising popularity of large scale shared memory machines like the DDM [5] and KSR1 [12]. The scalability is heavily dependent upon the frequency of remote operations. Scalability is thus measured by counting the *number of iterations migrated* as a result of the proposed algorithms. Recall that one of the major aims of the proposed algorithms is to reduce the number of migrations and thereby reduce the overhead of the resulting cache-reload transient that accompanies every such migration.

The *GSS* algorithm was included in the experimental study so as to provide an additional reference point by which to interpret the results for the other algorithms. *GSS* is a dynamic algorithm in which iterations are allocated from a single global queue. Large chunks of iterations are allocated to processors at the beginning of a loop so as to reduce synchronization overhead, while small chunks are allocated towards the end of the loop to balance the workload. Under *GSS*, when each processor retrieves work from the queue it takes $\lceil \frac{n}{G} \rceil$ of the n remaining iterations, where G is typically chosen to be equal to P, the number of processors. Since the first chunk in *GSS* contains $\lceil \frac{N}{G} \rceil$ of the iterations, where N is the number of iterations in the loop, the remaining iterations may not have enough work to balance the load, if the loop is unbalanced and $G = P$. In the particular implementation of the *GSS* algorithm used in this research, each processor takes $\lceil \frac{n}{2P} \rceil$ of the remaining iterations so as to address this problem in part. With this change, *GSS* now starts with smaller chunks, increasing its ability to appropriately schedule unbalanced loops, without introducing significant additional synchronization overhead. As the *GSS* algorithm does not exploit any affinity, its performance relative to the other affinity scheduling algorithms provides a yardstick by which the performance of the affinity scheduling algorithms can be compared.

In addition to the general goal of assessing the relative quality of the two proposed algorithms, other goals of the study were to observe:

1. The effect of the structure and extent of load imbalance in the parallel loop being scheduled, on the application execution time and the scalability of the proposed algorithms.

2. The "convergence" rate of the *dynamic* algorithm; i.e., the number of parallel loop executions (iterations of the outer sequential loop) required before the initial partitions of the *dynamic* algorithm become roughly constant.

All four scheduling algorithms (*static, dynamic, wrapped* and *GSS*) were implemented on a SGI 4D/380S running IRIX version 4.0.5. The IRIX operating system is based on AT&T's System V.3 UNIX with enhancements to support multiprocessing. The 4D/380S is a bus-based, cache-coherent, eight processor machine. Each processor in the system consists of a 33 MHz MIPS R3000 CPU and a R3010 floating point co-processor with private instruction (64 KB) and data (first level: 64 KB, second level: 256 KB) caches.

4.1 Workload

The proposed algorithms were evaluated using both a synthetic application where the parameters may be varied to allow exploration of algorithm performance under a wide variety of conditions, and also a sample real numerical application to which they are applicable. Each of these is described in turn:

1. The synthetic application considered has the structure shown in Figure 3.

```
// outer sequential loop on k
for (k = 0; k < K; k++) {
    // parallel loop on i
    for (i = 0; i < N; i++) {
        // f is a function
        seq_lim = f(i);
        // inner seq. loop on j
        for (j = 0; j < seq_lim; j++) {
           // perform a write operation
           A[i][j] = 1;
        }
    }
}
```

Figure 3: Outline of the synthetic application

The upper bound K of the outer sequential loop on k controls the number of executions of the parallel loop. Note that for small values of K, the initial partition sizes used by the *dynamic* algorithm may not have yet converged to reasonable values. In such a situation, it is possible that the readjustment of partition sizes may not bring about the desired benefits. The working set size and the degree of parallelism can be varied by changing the upper bound N of the parallel loop on i. The upper bound of the inner sequential loop on j is computed using a function f that takes as input the current value of the parallel loop index i. Forms of f are chosen so as to result in a roughly constant total working set size and computational requirement, across all runs with a given value of N. The form of f chosen (and thus the specific upper bounds of the inner sequential loop on j) controls the structure and extent of load imbalance.

The function f is varied in the experiments (as are the other simple parameters K and N). These parameters are used to generate different types of workloads so as to evaluate the performance of the proposed algorithms under a wide variety of conditions.

2. The real application employed is the Jacobi iterative algorithm: Given a non-singular matrix A and an n-dimensional vector b, we want to solve the system of linear equations $Ax = b$ for the unknown n-dimensional vector x. The Jacobi algorithm starts with an initial approximation of the solution vector, $x^{(0)}$, and repeatedly computes a new estimate $x^{(t)}$ from the previous estimate $x^{(t-1)}$ using the formula:

$$x_i^{(t)} = \frac{\sum_{(j=1,j\neq i)}^{n}(-a_{ij}x_j^{(t-1)}) + b_i}{a_{ii}}$$

The algorithm terminates when the estimates converge or after some pre-determined number of iterations have been performed.

Note that since the i^{th} iteration of the parallel loop always accesses the i^{th} row of the matrix, there is a significant amount of data locality that can be exploited. As we are concerned with applications that also have significant amounts of load imbalance, we consider the use of the Jacobi algorithm on problems in which most of the a_{ij} are zero; i.e., the matrix A is sparse. In this case, a good implementation will avoid storing the zero elements explicitly, and also avoid multiplication with them. The parallel loop part of an example implementation is shown in Figure 4. Only the non-zero coefficients are stored in the array *nzCoeffs*, while the array *nzColumns* records which columns they belong to. Element *firstNz[i]* indicates where the information for row i starts in *nzCoeffs* and *nzColumns*. The array *XO* is used to store the previously computed estimate ($x^{(t-1)}$) of the unknown n-dimensional vector x; while the array *x_new* holds the most recently computed estimates ($x^{(t)}$) of the unknown vector x.

The parallel loop on i is enclosed within an outer sequential loop (not shown in the figure) that iterates until convergence is reached or until some pre-determined number of iterations have been executed.

5 Results

Experimental results for the synthetic application include cases of varying numbers of iterations and it-

```
for (i = 0; i < N; i++) { // parallel loop
    sum[i] = 0;
    for (nz = firstNz[i]; nz < firstNz[i+1];
                          nz++) {
        if (i != nzColumns[nz])
           sum[i]+= nzCoeffs[nz]
                        * X0[nzColumns[nz]];
    }
    x_new[i] = (constant[i] - sum[i])/a[i][i];
}
```

Figure 4: Jacobi algorithm for sparse matrices

eration sizes. With respect to iteration size variability, the following three cases were considered:

1. Loops in which the iteration size decreases linearly (henceforth referred to as the "triangular workload"). The upper bound of the inner sequential loop on j in the i^{th} iteration is set to (N - i + 1), where N is the size of the parallel loop (in Figure 3).

2. Loops in which a fraction (k) of the iterations are of a constant large size, while the other remaining fraction has constant smaller size (henceforth referred to as the "rectangular workload"). The generic code for the parallel loop body is given in Figure 5. Workloads that vary significantly in the amount of load imbalance are generated by varying the value of k. In the experiments conducted, values of k = 2, 4 and 8 were used.

```
if (i < N/k)
   // i is the parallel loop index
   seq_lim = N * (k/2);
else
   seq_lim = 1;

for (j = 0; j < seq_lim; j++)
    A[i][j] = 1;
```

Figure 5: Generic code (for rectangular and balanced workloads)

3. Loops that have a constant iteration size. This workload is achieved by setting the value of k = 1 in the generic code shown in Figure 5.

Full sets of experiments were performed for outer sequential loop bounds of 100, 500, 1000 and 5000, although only the results obtained using an outer sequential loop bound of 500 are presented here, as the results for the other cases are very similar in nature. Results for various smaller outer sequential loop bounds are also presented, so as to assess the convergence rate of the *dynamic* algorithm. The parallel loop bound sizes considered are 128 and 1024.

Experiments were also performed using the Jacobi workload, for varying input matrix sizes. In this particular algorithm, since successive executions of an iteration of the parallel loop access exactly the same matrix elements, there is a significant amount of data locality that can be exploited. Since the implementation of the Jacobi algorithm used is essentially for sparse matrices, the application may also exhibit significant amounts of load imbalance. Experiments were performed using two different matrix sizes (128 $\times$ 128 and 1024 $\times$ 1024), each with approximately 15% sparsity. The elements of the matrix were generated using a function that maintained a correlated variation in the number of non-zero elements between any two successive rows of the matrix. Figure 6 shows the outline of this function. All elements not assigned to by this procedure remain with a value of zero, except the diagonal elements which are set to non-zero values.

```
// #non-zero elements in a row
number = N/4;
// 'i' loop over all the rows of the matrix
for (i = 0; i < N; i+=8) {
    for (k = i; k < i+8; k++) {
        for (j = 0; j < number; j++) {
            // assign random values to the
            // coefficients
            a[k][j] = assign_rand();
        }
    }
    number-= 2;
}
```

Figure 6: Function to generate correlated number of non-zero elements

Selected results are reported in this paper only for the rectangular workload using values of k = 2 and 8, for the balanced workload (k = 1), and for the Jacobi algorithm. A detailed discussion of the results appears in [13].

5.1 Principal Performance Comparison

With the rectangular workload skewed loops can be studied. Different degrees of imbalance are achieved by assigning $k = 2$ and 8 in Figure 5. Figures 7–10 present completion time results using the various scheduling algorithms for this workload. Figures 11–12 show results for the Jacobi algorithm for input matrix sizes of 128 and 1024. As the figures indicate, *GSS* performs the worst among all the four scheduling algorithms. For the Jacobi algorithm workload and for the rectangular workload with $k = 2$, this is simply because the *GSS* algorithm uses a single work queue and does not exploit any affinity information. As the results indicate, the two proposed affinity scheduling algorithms yield improved performance in comparison to *static*. In the case of iterations that decrease linearly in size, a uniform initial allocation of iterations to all processors (as is done in the *static* approach) results in a situation where the lightly loaded processors finish ahead of the other processors. These idle processors now probe the work queues of other processors and remove $\frac{1}{P}$ of the iterations from that of the most loaded processor. The cache-reload transient that arises as a result of the migration of work from the most loaded processors to the idle processors partially nullifies the resulting gains from exploiting affinity. The *dynamic* and *wrapped* algorithms result in a much improved initial load balance, minimize migrations and hence reduce the cache-reload transients. This hypothesis is confirmed by the results in Figures 13 and 14 which present the number of iterations that were migrated, for the rectangular workload with $k = 2$. (Results for the other workloads were similar in nature.) Note that the number of iterations migrated in these figures is the sum over all 500 iterations of the outer sequential loop.

In Figure 8, there is hardly any difference in the execution times under the three affinity scheduling algorithms when there are just two processors and the number of parallel loop iterations is 1024. (A similar effect occurs in Figure 12.) This is because the loop working set for large values of the parallel loop size does not fit in two caches. Thus, there are significantly fewer cache evictions on three or more processors (when the working set does fit); once the caches can contain the entire working set, affinity scheduling reduces the need for any main memory accesses.

Note that increasing the number of processors beyond three has little effect on the relative performance of the two proposed algorithms for large parallel loop sizes. For smaller parallel loop sizes, the relative performance of the *wrapped* algorithm decreases as the number of processors is increased as shown in Figure 7. This is due to the per-chunk overhead in the *wrapped* algorithm. For larger loop sizes (and hence larger chunks of iterations), this overhead has less of an effect as is evident in Figure 8.

Figures 9 and 10 show that there is a significant degradation in the performance of both the *GSS* and the *static* algorithms for larger values of k. Consider the generic code shown in Figure 5. When $k = 8$, the first $\frac{1}{8}^{th}$ of the iterations contain nearly all the work. Although (in the implementation used here) *GSS* assigns only $\frac{1}{2P}$ of the iterations in the first chunk of iterations that it dispenses, for smaller numbers of processors those iterations contain nearly all of the work. Although the *static* algorithm uses per-processor task queues, for smaller numbers of processors the first chunk taken by the first processor still contains nearly all the work, and thus this algorithm yields similar performance to *GSS* in this case. With the *dynamic* algorithm, this condition is experienced only on the first few iterations of the outer sequential loop. The dynamic readjustment of partition sizes ensures a better load balance in the system on successive iterations of the outer sequential loop. As the *wrapped* algorithm allocates iterations that are at a distance P (the number of processors in the system) apart, this algorithm avoids assigning all the time-consuming iterations to a single processor or chunk, and thus minimizes the chances of load imbalance.

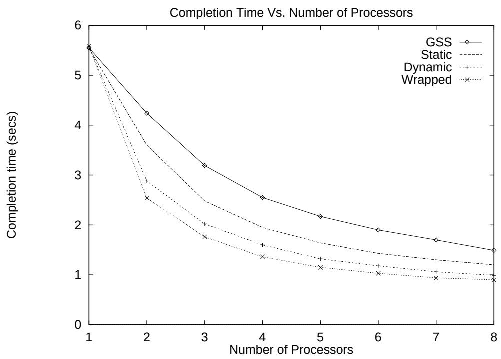

Figure 7: Rectangular Workload ($k = 2$, $N = 128$)

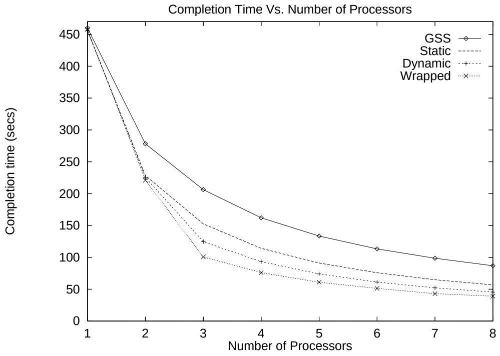

Figure 8: Rectangular Workload ($k = 2$, $N = 1024$)

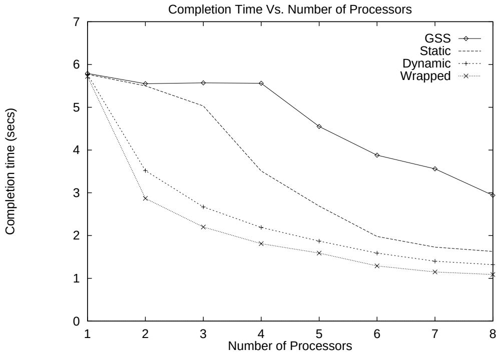

Figure 9: Rectangular Workload ($k = 8$, $N = 128$)

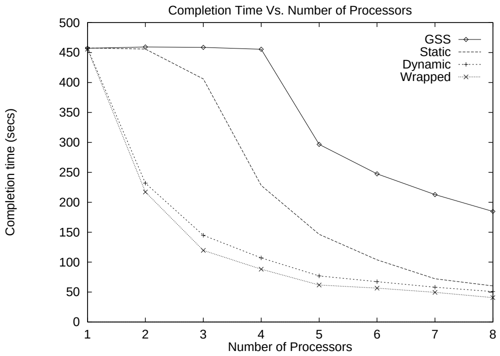

Figure 10: Rectangular Workload ($k = 8$, $N = 1024$)

5.2 Performance Under A Balanced Workload

In order to estimate the amount of overhead introduced by the *wrapped* and the *dynamic* algorithms,

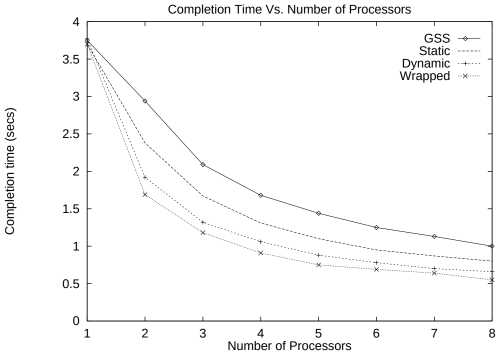

Figure 11: Jacobi Algorithm (matrix size = 128×128)

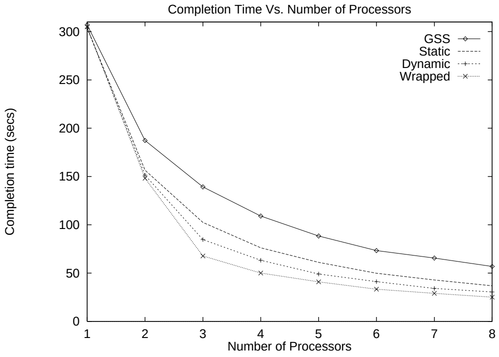

Figure 12: Jacobi Algorithm (matrix size = 1024×1024)

experiments were conducted using perfectly balanced loops (loops with a constant iteration size; achieved by setting $k = 1$ in Figure 5). Figures 15 and 16 present the completion times under each of the scheduling algorithms. There is basically no difference among the execution times under the three affinity scheduling algorithms. This indicates that the additional overhead due to the *wrapped* and *dynamic* algorithms is negligible. Figures 17 and 18 present the number of iterations migrated. Recall that the number of iterations migrated in these figures is a sum over all 500 iterations of the outer sequential loop. So as can be seen, the number of migrations is very small. The *wrapped* algorithm has the fewest migrations, perhaps because there is a little computational overhead added to the loop (due to the wrapped allocation of iterations) that may tend to decrease the effect of variability owing to cache and/or bus queueing delays. On the other hand, the *dynamic* algorithm has a comparatively larger number of task migrations when compared to the other two algorithms. This is because the *dynamic* algorithm follows a rather aggressive policy

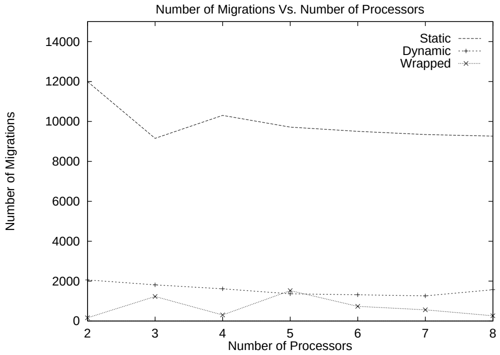

Figure 13: Rectangular Workload ($k = 2$, $N = 128$)

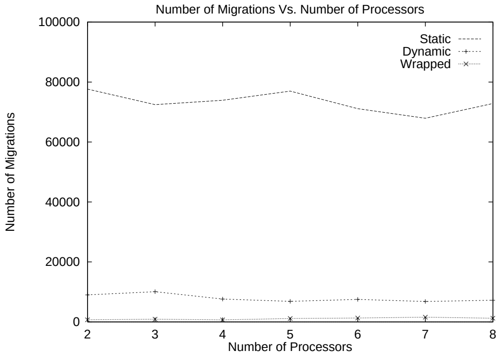

Figure 14: Rectangular Workload ($k = 2$, $N = 1024$)

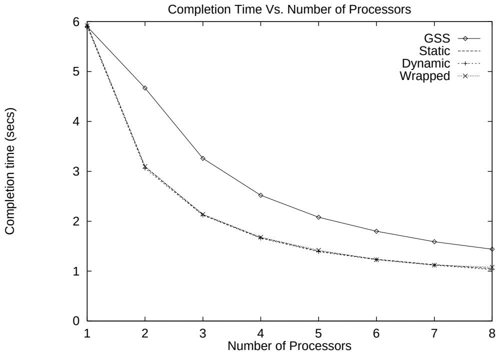

Figure 15: Balanced Workload ($N = 128$)

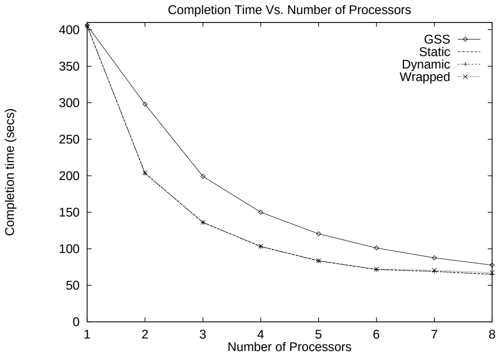

Figure 16: Balanced Workload ($N = 1024$)

in varying the initial partition sizes resulting in a situation wherein the initial partition sizes may often be slightly unequal, unjustifiably so. However, this does not appear to have any significant detrimental effects as there is no appreciable difference in the execution times under the three affinity scheduling algorithms under comparison.

5.3 Convergence Rate of the Dynamic Algorithm

Experiments were also performed for different numbers of outer sequential loop iterations to determine the "convergence" rate of the *dynamic* algorithm (i.e., the minimum value of the outer sequential loop bound for which the initial partitions stabilize to yield roughly constant ranges). Figures 19–22 present the results obtained for the rectangular ($k = 8$) workload. (Results for the other workloads considered are quite similar in nature.) Note that the results are presented only in terms of the number of migrations as the execution times for small sequential loop iteration counts are heavily influenced by the startup overhead. As is evident from these figures, with few outer sequential loop iterations (< 4), there is no perceptible difference between the *dynamic* and *static* algorithms. This is because the initial partition sizes used by *dynamic* have still not converged to reasonable values, and thus the readjustment of partition sizes have not yet brought substantive benefits. Larger outer sequential loop iteration counts afford significantly greater potential for these initial partition sizes to converge. Ten outer sequential loop iterations (Figure 22) appear to be sufficient to achieve convergence in this particular case.

6 Conclusions

Two new affinity scheduling algorithms (*dynamic* and *wrapped*) for parallel loops have been proposed in this paper. An experimental study was performed comparing these algorithms to Guided Self-Scheduling (*GSS*) [11] and the static partitioned affinity scheduling algorithm (*static*) of Markatos and LeBlanc [9].

The major conclusions from this research are:

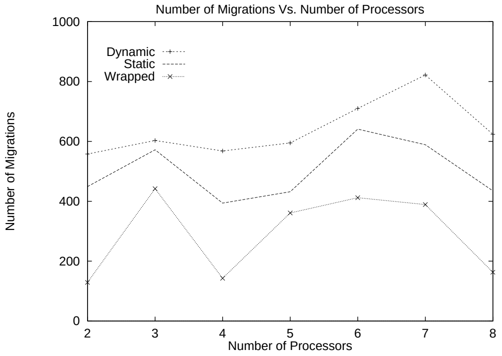

Figure 17: Balanced Workload ($N = 128$)

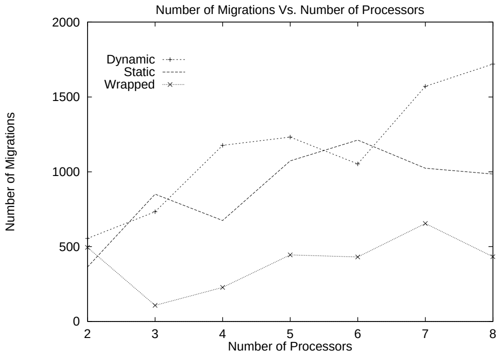

Figure 18: Balanced Workload ($N = 1024$)

- The affinity scheduling algorithms perform much better than *GSS* (which uses a central work queue). This is because a single work queue requires the frequent movement of data among processors, since every processor must first load the data it needs into its local cache. The resulting communication overhead degrades performance.

- By reducing the number of iterations that need be migrated, and thus better respecting processor affinity, the two proposed algorithms (*wrapped* and *dynamic*) perform well for severely unbalanced workloads. Further, the results indicate that there is no substantial additional overhead in either the *dynamic* or the *wrapped* algorithms.

- Both *GSS* and the *static* algorithms may show a marked degradation in performance for severely unbalanced workloads. In such severely unbalanced workloads, the first chunk of iterations may contain most of the work, leaving the remaining chunks very little work to balance the load. For such workloads, the dynamic readjustment of partition sizes and also the wrapped assignment of iterations provides for superior load balancing

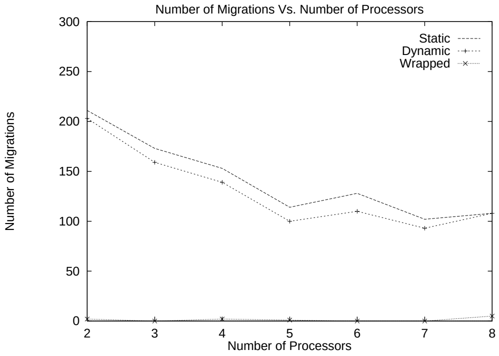

Figure 19: Rectangular Workload ($k = 8$, $N = 1024$, $K = 1$)

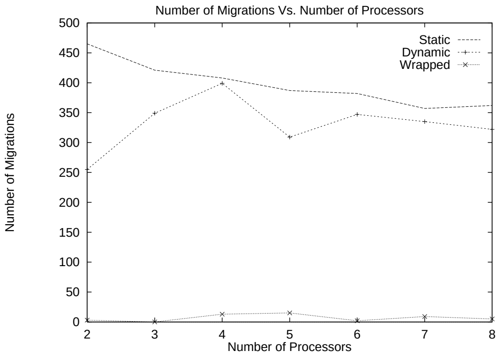

Figure 20: Rectangular Workload ($k = 8$, $N = 1024$, $K = 2$)

capability. The *dynamic* and *wrapped* algorithms thus seem very robust with respect to the type of workload.

- The *dynamic* algorithm requires only a small number of outer sequential loop iterations to "converge" (i.e., for the initial partitions to stabilize). With few outer sequential loop iterations (< 4), convergence cannot be achieved, however, and there is no perceptible difference between the *dynamic* and *static* algorithms.

It is important to note that the good performance of the *wrapped* algorithm does not imply that the *dynamic* algorithm need never be used. The wrapped assignment of iterations results in assigning consecutive iterations to distinct processors – thus violating spatial locality. When consecutive iterations access data that is in consecutive memory locations, a single cache miss may load data useful in multiple consecutive iterations. Due to the wrapped allocation of iterations, processors may not be able to take advantage of the data that is already present in the cache (as a result

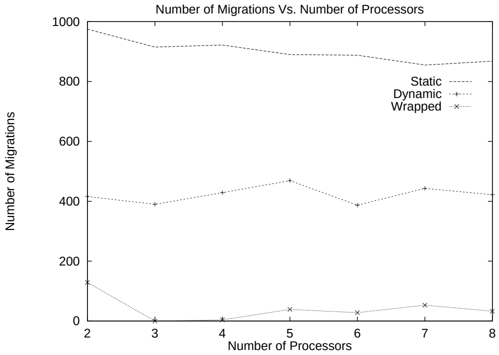

Figure 21: Rectangular Workload ($k = 8$, $N = 1024$, $K = 4$)

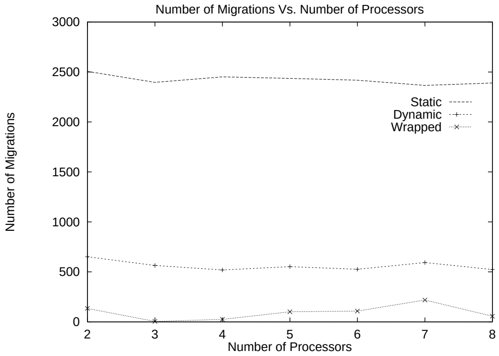

Figure 22: Rectangular Workload ($k = 8$, $N = 1024$, $K = 10$)

of a previous cache miss). Wrapping small intervals of consecutive iterations is a possible solution [4], but this will have to be traded off against the disadvantage of a more complicated mapping function.

References

[1] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1 (January 1990), pp. 6-16.

[2] T. E. Anderson, E. D. Lazowska, H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Computers*, Vol. 38, No. 12 (December 1989), pp. 1631-1644.

[3] R. Chandra, A. Gupta, J. L. Hennessy, "Data Locality and Load Balancing in COOL", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 249-259.

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

[5] E. Hagersten, A. Landin, S. Haridi, "DDM - A Cache Only Memory Architecture", *IEEE Computer*, Vol. 25, No. 9 (September 1992), pp. 44-54.

[6] S. F. Hummel, E. Schonberg, L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops", *Communications of the ACM*, Vol. 35, No. 8 (August 1992), pp. 90-101.

[7] C. P. Kruskal, A. Weiss, "Allocating Independent Subtasks on Parallel Processors", *IEEE Transactions on Software Engineering*, Vol. 11, No. 10 (October 1985), pp. 1001-1016.

[8] E. P. Markatos, T. J. LeBlanc, "Load Balancing vs. Locality Management in Shared-memory Multiprocessors", *Technical Report 399*, Computer Science Department, University of Rochester, October 1991.

[9] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 4 (April 1994), pp. 379-400.

[10] J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1 (February 1991), pp. 21-65.

[11] C. D. Polychronopolous, D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Transactions on Computers*, Vol. 36, No. 12 (December 1987), pp. 1425-1439.

[12] J. Rothnie, "Overview of the KSR1 Computer System", *Kendall Square Research Report TR 9202001*, March 1992.

[13] S. Subramaniam, "Affinity Scheduling of Unbalanced Workloads", *M.Sc. Thesis*, Department of Computational Science, University of Saskatchewan, August 1993.

[14] R. H. Thomas, W. Crowthier, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors", *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 245-254.

[15] A. Tucker, A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 159-166.

[16] T. H. Tzen, L. M. Ni, "Dynamic Loop Scheduling for Shared Memory Multiprocessors", *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991, pp. 247-250.

[17] R. Vaswani, J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors", *Proceedings of the 13th ACM Symposium on Operating System Principles*, October 1991, pp. 26-40.