Restructuring Arrays for Efficient Parallel Loop Execution

Shun-Tak Leung and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 94-02-01
February 1994

Restructuring Arrays for Efficient Parallel Loop Execution

Shun-Tak Leung and John Zahorjan *
Department of Computer Science & Engineering
University of Washington

February 1994

Abstract

In a sequential program, data are often structured in a way that is optimized for a sequential execution. However, when the program is parallelized, the data access pattern may change drastically. If the structure of the data is not changed accordingly, parallel performance will suffer. In this paper, we consider this problem in the context of runtime loop parallelization [8, 9], which is a general technique to parallelize loops not amenable to compile-time analysis.

In a parallel execution of a loop, iterations may be performed in a very different order than in the sequential execution. This may result in undesirable cache effects on distributed shared-memory multiprocessors, unless the structure of the arrays accessed by these iterations is changed accordingly. We discuss what these problems are and how they arise. We then describe two data restructuring techniques to address them: the restructuring of read-write arrays to reduce inter-processor communication due to false sharing, and the restructuring of read-only arrays to improve spatial locality.

We also report experiments on a KSR1 [3] to evaluate the effectiveness of these techniques and the preprocessing and postprocessing overheads they entail. The results show that the restructuring techniques can substantially improve performance of the parallelized loop. When restructuring overheads are ignored, we see a doubling of parallel speedups. While restructuring overheads can be quite significant, they can often be amortized across multiple loop executions so that they do not outweigh the performance benefits. In our experiments, it takes only two loop executions to achieve this.

Keywords: data restructuring, false sharing, spatial locality, runtime parallelization, distributed shared-memory multiprocessor

1 Introduction

Automatic parallelization of sequential programs is a popular approach of programming parallel machines. It offers the programmer the familiar sequential programming model, while exploiting the computational power of multiple processors to achieve high performance. The compiler assumes the responsibility of transforming the sequential program into efficient parallel code.

*This material is based upon work supported by the National Science Foundation (Grants CCR-9123308 and CCR-9200832), the Washington Technology Center, and Digital Equipment Corportation (the External Research Program and the Systems Research Center). Authors' addresses: Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195; *shuntak@cs.washington.edu, zahorjan@cs.washington.edu.*

In a sequential program, data are often organized in a way reflecting the sequential nature of the program. In other words, the programmer will likely structure the data in a way that is optimized for the access pattern of a sequential execution. If the program is parallelized without changing the organization of the data, parallel performance may suffer severely: the data organization that is optimal for sequential execution may not be well suited for parallel execution since the data access patterns of sequential execution and parallel execution of the same program can be drastically different.

In this paper, we focus on the parallelization of loops on distributed shared-memory multiprocessors, and we are concerned about how arrays are accessed. For concreteness, we will explain the issues using an example, a sparse lower triangular solve. The loop is shown in Figure 1. In the loop, b and x represent the right-hand side of the linear system and the solution, respectively. Arrays $nzColumns$, $firstNz$ and a together represent the sparse coefficient matrix. Array a contains the values of nonzero elements, while $nzColumns$ contains the corresponding column indices. $firstNz[i]$ indicates the positions in a and $nzColumns$ where the data for row i begin, and (implicitly) where those for row $i-1$ end.

```
do i = 1, n
  temp = b[i]
  do nz = firstNz[i], firstNz[i+1] - 1
    temp = temp - a[nz] * x[nzColumns[nz]]
  enddo
  x[i] = temp
enddo
```

Figure 1: Sparse Lower Triangular Solve

In a sequential execution of the loop, the processor reads through all the read-only arrays (a, b, $firstNz$, $nzColumns$) consecutively because the data for row $i+1$ are placed immediately after those for row i. Thus, the processor accesses each cache line, reads all the array elements in it, and does not access the cache line again in this execution of the loop. The elements of x are also written consecutively. Notice that the loop reads the elements of the read-only arrays only once (except those of $firstNz$, which are each read twice by two consecutive iterations) and also writes the elements of x only once. Therefore, regardless of the order of iteration execution, there is little temporal locality. The sequential execution maximizes spatial locality and therefore is best in terms of locality [1].

When we parallelize the loop, problems arise if we do not change the structure of the data accordingly. Since the loop contains loop-carried dependences, we cannot simply let each processor execute a large chunk of consecutive iterations in parallel. Instead, iterations must be executed according to a schedule that respects the dependences and at the same time tries to allocate iterations evenly across processors. These issues require iterations to be executed in an order that bears little resemblance to the iteration index order, which the sequential execution follows. Consecutive iterations may be executed by different processors at different points in time. The situation is depicted in Figure 2, which is taken from a real schedule.

We focus in particular on two problems that result from this: loss of spatial locality and false sharing. To understand the first problem, let us consider how one of the processors accesses data as it executes its share of the iterations. We assume that its cache is not large enough for all the necessary data. For instance, if a processor executes iteration 2 immediately after iteration 1 (as would be the case in a sequential execution), many of the data it needs for iteration 2 may be already in its cache since they are adjacent to those for iteration 1 and thus may share the same cache line [2]. Unfortunately, the parallel schedule requires

[1] Elements of x are not read consecutively and may exhibit poor locality, but this is unavoidable since which elements to read is determined by the sparsity structure of the coefficient matrix, which the program cannot control.

[2] The term "cache line" has two different meanings in this discussion. In the spatial locality problem, a cache line is a unit

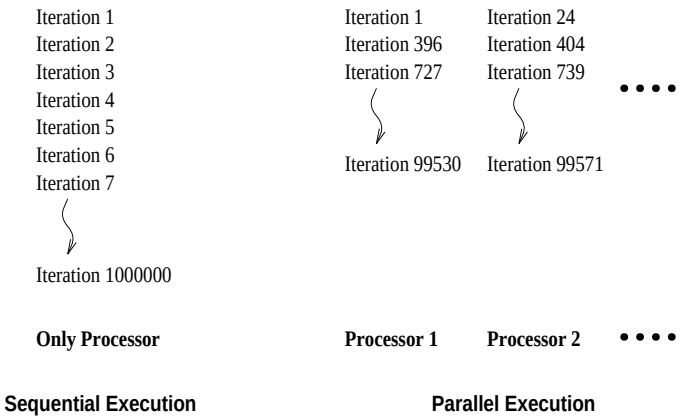


Figure 2: Sequential vs. Parallel Execution

the processor to execute iteration 396 instead (see Figure 2). The necessary data are nowhere near those for iteration 1. Therefore, the processor cannot take advantage of the data brought into its cache by the immediately preceding execution of iteration 1. And even if the processor eventually executes iteration 2, by the time it does so the data may have already been ejected from the local cache to make room for other data accessed in the meantime. Much spatial locality is lost.

To appreciate the performance impact, let us compare the time to execute the example loop on one processor using two different schedules: a "sequential" schedule that causes iterations to be performed in index order, and a parallel schedule computed by runtime parallelization (described in Section 2). In both cases, the processor executes the parallel code, and no synchronization or communication is required since only one processor is involved. They differ only in the iteration execution order. The execution times for four matrices are shown in Figure 3. It is clear that the loss of spatial locality alone can increase execution time substantially.

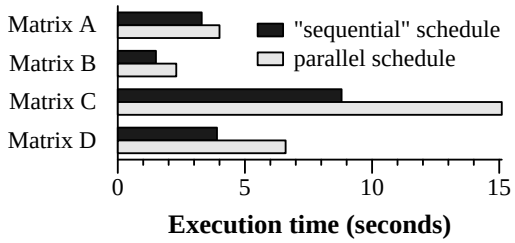


Figure 3: Performance Impact Losing Spatial Locality

The second problem related to data organization is false sharing. In a sequential execution of the loop, adjacent elements of x in the same cache line are written by consecutive iterations [3]. However, the parallel

of allocation, the granularity at which cache space is allocated and old data are ejected from the cache to make room for new data. In the false sharing problem, a cache line is a unit of cache coherence, the granularity at which the memory system moves data between processors to ensure a consistent view of memory. In many architectures, the two are the same, but this is not necessarily the case. In KSR1, for example, the unit of coherence is a *subpage* (128 bytes) while the unit of allocation is a *page* (consisting of 128 subpages) [4].

[3] We assume here that a cache line is larger than an array element. The larger the ratio of their sizes, the more serious is the false sharing problem. Unless an array element is itself a data structure with multiple components (such as a *struct* in C), it is likely to be smaller than a cache line, especially in view of the large cache line size in recent architectures.

schedule scatters these consecutive iterations onto different processors to be executed at different points in time. Whenever an array element is written, all copies of the containing cache line are invalidated. This leads to considerable inter-processor communication. In the worst case, each write leads to one invalidation and multiple unnecessary cache misses. The situation is similar for an update-based cache coherence protocol, and the result is the same. We shall see in Section 4 that false sharing has a very large impact on performance.

Often false sharing can be reduced or even eliminated by placing each logically distinct data structure in a different cache line, but this is not a good solution in our case. For one thing, it requires a huge amount of memory because we are dealing with a large number of small array elements, rather than a few shared data structures. Also, since in our case each element is written only once in the loop, this approach does not reduce the number of invalidations. Every write to the array still causes one invalidation because copies of the cache line which other processors acquired in the previous loop execution have to be invalidated before the write can proceed. We need to reorganize the data in order to effectively tackle this problem.

The problems described above result from parallel execution of a sequential loop without a corresponding change in the way data are structured. In this paper, we are particularly concerned about these problems in the context of runtime parallelization [8, 9, 5], a technique to automatically parallelize loops containing loop-carried dependences that cannot be fully determined by the compiler. Since the parallel schedule, and hence the data access pattern, is not determined until run time, any restructuring of the data can also be done only at run time. In this paper, we explore techniques to do this.

The remainder of this paper is organized as follows. Section 2 briefly discusses runtime parallelization. In Section 3, we describe techniques to restructure read-write and read-only arrays so that the cache penalty can be reduced. In Section 4, we present the results of experiments on a KSR1 to evaluate the effectiveness of these techniques and the overheads involved. Section 5 concludes this paper.

2 Runtime Parallelization

Parallelizing a sequential loop requires an analysis of the loop-carried dependences. In many cases, this can be done at compile time, and so the compiler can compute a valid parallel schedule for the iterations. However, there are also many cases in which the dependences cannot be determined until run time. One possible reason is that they depend on the contents of indirection arrays which are known only at run time. Sparse matrix problems, such as our example in Figure 1, are typical examples. To address this problem, runtime parallelization has been proposed [8, 9]. Instead of producing a parallel schedule directly, the compiler generates two pieces of code: the *inspector* and the *executor*. At run time, the inspector computes a valid schedule, and the executor executes iterations in parallel according to the computed schedule.

Let us call the sequential loop that we want to parallelize the *source loop*. The form of source loops we consider in this paper is shown in Figure 4 (adapted from [8]). Since different iterations may read and write the same element of the array x, the loop may contain loop-carried dependences, thus preventing completely parallel execution. Moreover, if the array indexing expressions $g(i)$, $h(i)$, etc. involve, say, indirection arrays whose contents are known only at run time, the compiler cannot find a correct parallel schedule.

```
do i = 1, n
  x[i] = F(x[g(i)], x[h(i)], ...)
enddo
```

Figure 4: Sequential Source Loop

What the compiler can do, however, is to generate an inspector to compute the schedule at run time, when the necessary information is available. Immediately before the loop is executed, the inspector evaluates the array indexing expressions $g(i)$, $h(i)$, etc., determines the loop-carried dependences, and calculates a valid parallel schedule for the iterations. In the schedule, the set of all iterations is partitioned into a number of non-intersecting subsets, called *wavefronts*. All iterations of a particular wavefront are independent of one another and therefore can be performed in parallel. The wavefronts are processed one by one in a specified order. In effect, the sequential source loop is transformed into a sequence of parallel loops, each containing some of the original loop's iterations. If the parallel schedule thus computed can be reused many times, or if it can be computed in parallel [5], the overhead of running the inspector will be more than compensated for by the savings of parallel loop execution.

Implementation of the executor on a shared-memory multiprocessor can be very straightforward. It has an outer loop that goes through the wavefronts one by one. In one iteration of this outer loop, each processor executes its share of the source loop iterations in the current wavefront and then synchronizes with other processors at a barrier. Once a processor has selected an iteration index, the corresponding iteration can be executed in exactly the same way as it would be by the sequential source loop. The executor for our example is shown in Figure 5(b), with the source loop (originally in Figure 1) repeated in Figure 5(a) for easy comparison. In the rest of this paper, we will call this basic executor implementation the baseline executor.

Although this basic implementation is simple, its performance is disappointing, as we shall see in Section 4. The reason, as we have already discussed, is that parallelization is not accompanied by a suitable reorganization of the data. In the next section, we describe techniques that restructure the arrays so that they are better suited to the access pattern of parallel execution.

```
do i = 1, n
  temp = b[i]
  do nz = firstNz[i], firstNz[i+1] - 1
    temp = temp - a[nz] * x[nzColumns[nz]]
  enddo
  x[i] = temp
enddo
```

(a) Sequential Source Loop

```
do w = 1, wavefronts
  do i = iterations in wavefront w
    temp = b[i]
    do nz = firstNz[i], firstNz[i+1] - 1
      temp = temp - a[nz] * x[nzColumns[nz]]
    enddo
    x[i] = temp
  enddo
  barrier synchronization
enddo
```

(b) Baseline Executor (code for one processor)

Figure 5: Sparse Lower Triangular Solve

3 Data Restructuring

In Section 1, we discussed two problems arising from parallel execution without a corresponding restructuring of data: false sharing of read-write arrays and the loss of spatial locality due to non-consecutive accesses to read-only arrays.

To address these problems, we can try to pay more attention to locality when performing iteration assignment. There is little flexibility in choosing which wavefront each iteration is assigned to. It is largely determined by the loop-carried dependences. There is greater flexibility in the allocation of iterations of a given wavefront to processors. There are in fact sophisticated algorithms that partition array elements

(and thereby iterations of parallel loops operating on these elements) across processors so as to minimize inter-processor communication [10, 1]. Unfortunately, the existence of loop-carried dependences limits, if not precludes, the possibility of adapting them for use in our case. Conceivably, one can still apply some form of such algorithms to iterations within a given wavefront (which are independent), but because these iterations are not contiguous in the iteration space to begin with, the effectiveness of this approach is questionable.

We try to attack the problems from a different direction. Instead of making the parallel schedule suit the organization of data, we change the organization of data to suit the parallel schedule. In this section, we discuss the techniques in some details. We will first describe the techniques dealing with each of the two problems. Then the assumptions behind their applicability will be discussed more fully. As before, for concreteness, we will explain the techniques with reference to the example source loop in Figure 5(a).

3.1 Restructuring Read-Write Arrays

In this section, we discuss techniques for restructuring read-write arrays. We call this *read-write restructuring*. The basic idea is to separate the physical location of an array element from its index. In other words, the index names an array element but does not directly indicate where in memory the element is placed. The mapping from index to location is represented by an indirection table. The executor for the example source loop with read-write restructuring is shown in Figure 6. Note that accesses to x, the only read-write array in this example, go through the indirection table *map*.

```
do w = 1, wavefronts
  do i = iterations in wavefront w
    temp = b[i]
    do nz = firstNz[i], firstNz[i+1] - 1
      temp = temp - a[nz] * x2[map[nzColumns[nz]]]
    enddo
    x2[map[i]] = temp
  enddo
  barrier synchronization
enddo
```

Figure 6: Executor with Read-Write Restructuring (code for one processor)

With the freedom to place an array element independent of its index, we can organize the data in order to avoid false sharing. We start with the parallel schedule produced by the inspector, which specifies which iterations a particular processor should execute in a particular wavefront. The array elements are structured in such a way that all elements in a cache line are written by the same processor in the same wavefront, while trying to pack as many elements as possible into each cache line. We will describe our implementation in more details later in this section, but for the time being it suffices to keep this key goal in mind.

With read-write restructuring, false sharing is no longer a problem. When a cache line is first written in a certain wavefront, invalidation of copies is of course necessary. After this, all other elements in the cache line can be written without further invalidations or misses because no other processors will access them in the current wavefront. First, the organization of data ensures that no other processors *write* these elements. Secondly, we know that because the parallel schedule respects loop-carried dependences, no other processors *read* these elements in the current wavefront. If this were not the case, the iteration reading an element would have a loop-carried dependence with the iteration writing it. A valid schedule cannot have both in the same wavefront.

After the current wavefront has been completed, copies of the cache line gradually migrate to other processors as its elements are read. Since the entire cache line will only be written in the next loop execution, no more coherence misses will occur until then. Thus, one cache line leads to one invalidation and (possibly) some misses in each loop execution. In fact, a latency hiding feature of the KSR1 (on which we have implemented our techniques) further reduces misses [11]: when a copy of a subpage travels through the memory system, processors that have an outdated copy will update their copies even if they have no outstanding requests for the subpage. Thus, the first miss on any processor can bring the subpage not only to that processor but also to other processors, which may then avoid a cache miss when they read the subpage in the future. The assumption that processors have outdated copies of the subpage is clearly invalid in the first execution of the loop but more likely to be true in subsequent executions.

Now, we discuss some specific issues in our implementation. First of all, we describe how the array elements are organized given a parallel schedule. A schedule specifies when (i.e., in which wavefront) and where (i.e., on which processor) each iteration is to be executed. Furthermore, it may specify the order in which iterations assigned to the same wavefront and processor are performed. We arrange the array elements first by wavefront, then by processor, and finally by the order in which they are written, as depicted in Figure 7. Other options are possible. Our choice is mainly for ease of implementation.

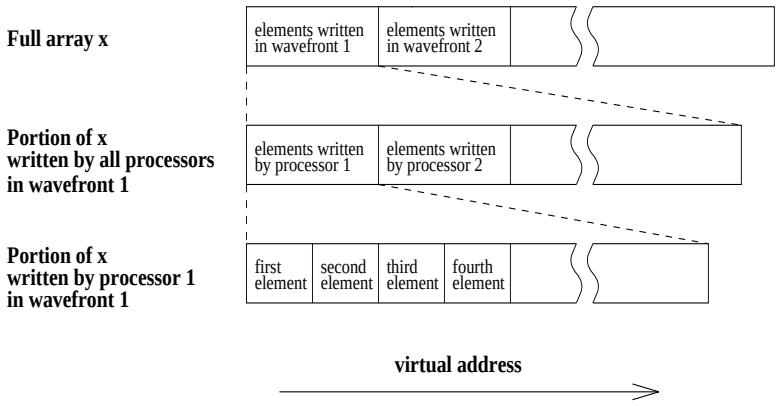


Figure 7: Restructured Read-Write Array x

With this organization, some false sharing may still remain because cache lines at the "boundary" between two processors are written by both. However, the impact is expected to be small in part because the majority of cache lines are written by only one processor. Moreover, some of the elements in such a "boundary" cache line are written at the beginning of a wavefront by one processor while the rest are written near the end by another. Thrashing is unlikely because the cache line is not written simultaneously. If we want to eliminate false sharing completely, we can make each set of array elements written by the same processor in the same wavefront start on a cache line boundary. (Our implementation does not enforce this.)

Finally, we consider how the reorganization of data is actually done in our implementation. The arrays are not reorganized *in situ*. Instead, we adopt a copy-in-copy-out approach. Before the loop is executed, each read-write array is copied to a temporary area, and reorganized on the fly. After the loop has been executed, it is copied back. One reason for this is that the overhead would be prohibitive if all references to the array elements in the entire program have to go through an extra level of indirection translating indices to locations. Moreover, the array may be accessed in different manners by different loops. A data organization suitable for one loop may not be good for another. The copying does represent some overhead, but it can be done completely in parallel and, as we shall see in Section 4, is not a major concern.

3.2 Restructuring Read-Only Arrays

In this section, we discuss how to restructure read-only arrays, that is arrays which are read but not written in the source loop (although possibly written elsewhere in the program). We call this technique *read-only restructuring*, and the combination of read-write and read-only restructuring *complete restructuring*.

The primary purpose of read-only restructuring is to maximize spatial locality. To do this, the read-only array elements to be read by each processor are laid out in memory following the order in which they will be read, as depicted in Figure 8, so that the processor can go through them consecutively. If an array element is read twice, it is duplicated. (For example, notice that $firstNz[13]$ appears twice in Figure 8.) In fact, we go beyond restructuring each array individually; elements of all the restructured arrays are interspersed and arranged according to the order in which they are read. Doing so not only allows processors to keep reading data consecutively for as long as possible, but also simplifies implementation.

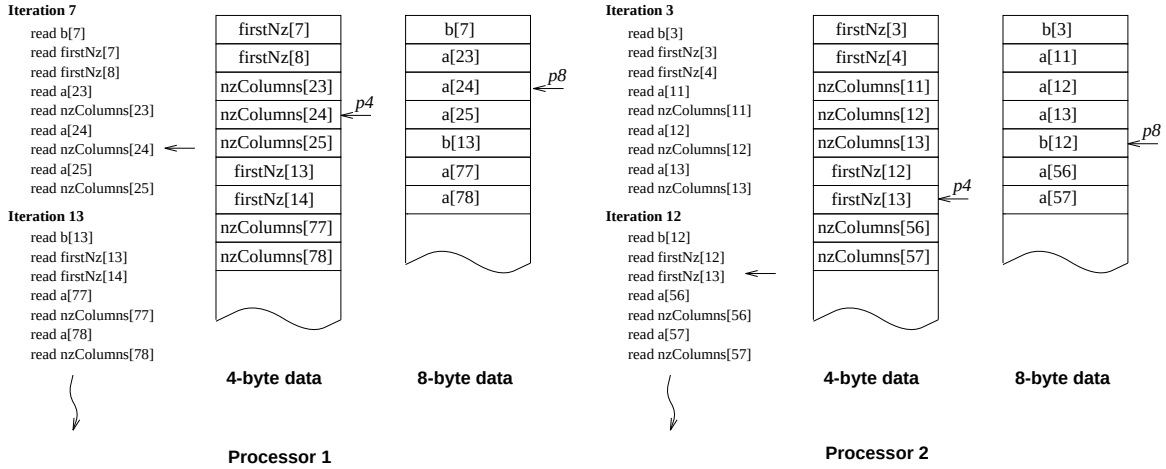


Figure 8: Read-only Restructuring

Data alignment requirements on many architectures add a minor complication. For example, the KSR1 hardware requires eight-byte data to be aligned on eight-byte boundary. Thus, array elements of different sizes cannot be freely interspersed. We either end up with "holes" of unused memory, or have to break up one memory access into multiple accesses of a smaller granularity. Neither is satisfactory. We choose to intersperse array elements that have the same alignment requirement but separate those that do not in different regions of memory. The number of regions needed is small and depends solely on the architecture, not the program.

The example executor implementing *complete* restructuring is shown in Figure 9(a). Each processor maintains a pointer (in fact one for each kind of alignment requirement) to keep track of where to read the next array element and advances the pointer appropriately after each reference. The C-style expressions $*(p4++)$, $*(p8++)$ mean pointer deference followed by increment.

Before we explain the absence of any reference to the indirection array *map* (c.f. Figure 6), let us consider what preprocessing and postprocessing read-only restructuring needs. Like read-write restructuring, read-only restructuring is not done *in situ*. The preprocessing loop (shown in Figure 9(b)) mimics the executor (Figure 9(a)). It reads the read-only array elements as the executor would and copies them to temporary areas. Each processor handles exactly those data that it will need later when it runs the executor. Therefore, preprocessing is done entirely in parallel; no synchronization is needed between wavefronts. As

```
do w = 1, wavefronts
  do i = iterations in wavefront w
    /* p4 points at 4-byte data */
    /* p8 points at 8-byte data */
    initialize pointers p4, p8
    temp = *(p8++)
    firstNz_1 = *(p4++)
    firstNz_2 = *(p4++)
    do nz = firstNz_1, firstNz_2 - 1

      temp = temp - *(p8++) * x2[*(p4++)]
    enddo
    x2[*(p4)++] = temp
  enddo
  barrier synchronization
enddo
```

(a) Executor

```
do w = 1, wavefronts
  do i = iterations in wavefront w
    /* p4 points at 4-byte data */
    /* p8 points at 8-byte data */
    remember pointers p4, p8
    *(p8++) = b[i]
    *(p4++) = firstNz[i]
    *(p4++) = firstNz[i+1]
    do nz = firstNz[i], firstNz[i+1] - 1
      *(p8++) = a[nz]
      *(p4++) = map[nzColumns[nz]]
    enddo
    *(p4++) = map[i]
  enddo
  /* no barrier synchronization */
enddo
```

(b) Preprocessing

Figure 9: Complete Restructuring (code for one processor)

for postprocessing, nothing needs to be done (except perhaps memory deallocation) because the data are not changed by the loop and the original arrays are not disturbed at all.

Now, we can explain why the executor with complete restructuring needs not access the indirection table *map*. This results from an optimization made possible by read-only restructuring. Suppose the source loop accesses one or more read-only indirection arrays for the sole purpose of locating elements in an ultimate target array, which is itself read-only and restructured. The preprocessing loop resolves these levels of indirection to locate the right target array elements and then store them for future use by the executor. Since the source loop (and hence the executor) is really only interested in these target array elements, there is no need to store the indirection array elements themselves, assuming that the corresponding reads in the executor are dropped as well. Effectively, we flatten multiple levels of indirection into no indirection at all. This can significantly reduce the amount of data read.

This optimization can be applied to reducing the indirection overhead of read-write restructuring. Consider the executor shown in Figure 6. In the source loop, the array $nzColumns$ contains indices used solely to index array x. When x is restructured, the reference $x[nzColumns[nz]]$ in the source loop becomes $x2[map[nzColumns[nz]]]$ in the executor. The executor reads $nzColumns[nz]$ for the sole purpose of finding $map[nzColumns[nz]]$. Thus, the preprocessing loop can read $nzColumns[nz]$, find $map[nzColumns[nz]]$, and then store this value for the executor but discard the value of $nzColumns[nz]$. The executor with complete restructuring (Figure 9(a)) therefore needs not read $nzColumns$ or map. Instead, it can directly obtain the value of $map[nzColumns[nz]]$ prepared by the preprocessing loop.

3.3 Discussion

So far, we have informally described the restructuring techniques. We may have alluded to but have not discussed in detail some of the underlying assumptions. In this section, we discuss these assumptions more fully and the extent to which they can be relaxed.

A condition essential for these restructuring techniques is that the data access pattern of the source loop can be determined *a priori*. In other words, the set of array elements accessed by each iteration and the order in which they are accessed do not depend on computation performed earlier in the source loop. For read-write restructuring, naturally this assumption concerns only read-write arrays. The same assumption is necessary for us to be able to determine the loop-carried dependences using an inspector generated from the source loop but without executing the source loop itself. Therefore, it is imperative for the inspector-executor approach of runtime parallelization. Thus, read-write restructuring does not impose an additional constraint on the type of loops that can be handled, although the same cannot be said of read-only restructuring.

Another assumption we have made is that iterations are scheduled statically. The very concept of restructuring is to rearrange data according to a given parallel schedule for the iterations so that processors, when they follow this schedule, can access data less expensively. Clearly, this schedule must be determined before data are restructured and the loop is executed. However, strict adherence to a static schedule may result in serious load imbalance.

Fortunately, it is only necessary to schedule iterations "almost" statically. Our implementation maintains enough information on each iteration so that any iteration can be executed on any processor without producing incorrect results. Of course, the farther we deviate from the static schedule, the less effective restructuring would become. Therefore, the overall strategy is to compute a static schedule and follow it as closely as possible but depart from it when dictated by, say, load imbalance. (One such scheduling policy is affinity scheduling [7].) In this way, restructuring can yield significant benefits in the common case, without precluding more dynamic scheduling policies when necessary.

Also, so far we have assumed that each iteration writes one element of one restructured read-write array. Conversely, each element is written by at most one iteration. Let us consider their implications. If an iteration writes multiple array elements, we simply place these elements together when we restructure the array. Recall that the key goal is that array elements within the same cache line are written by the same processor in the same wavefront. This is as easily satisfied when each iteration writes one array element as when it writes more than one, possibly of different arrays.

The problem is harder if multiple iterations may write the same element. (In fact, parallelization of the source loop is *much* harder because of the more complex dependence pattern [6]. How read-write restructuring can handle this is therefore overshadowed by the more fundamental problem.) Given an assignment of iterations to processors and an array element written by multiple iterations, the question is which iteration the array element should "follow" when we decide where to place it. Any choice is acceptable for correctness, but false sharing unavoidably returns because two or more processors may again write the same cache line. It can perhaps be avoided or reduced by judicious assignment of iterations to processors, but this is beyond the scope of this paper.

As for the read-only arrays, we have been assuming that each element is read only once, or at most a small number of times. If an element is read multiple times, restructuring in effect duplicates it that many times. Thus, the restructured data may require more memory than the original versions, thus causing more cache misses and potentially worse performance if the original data can fit in the cache but the restructured data cannot. Both the benefits of consecutively accessing data and the penalty of accessing more memory must be considered when deciding whether a read-only array should be restructured.

4 Performance Results

A number of experiments were performed to evaluate the effectiveness of the restructuring techniques. We are mainly concerned with two questions. The first is how much performance improvement is achieved by

restructuring (either read-write restructuring alone, or complete restructuring) over the baseline executor. A second equally important question is the size of the preprocessing and postprocessing overheads discussed in Section 3.1 and Section 3.2. If these overheads are too large, they may outweigh the benefits.

The experiments were performed on a Kendall Square Research KSR1 distributed shared-memory multiprocessor [3] running OSF/1. The machine is configured with 64 processors in two 32-processor rings. Communication between any pair of processors within the same ring is equally expensive, while communication between two processors on different rings is significantly more costly than intra-ring communication.

The source loop is the example we have been using (Figure 5(a)). It was manually transformed into the inspector, executor and preprocessing loop, although this process can be automated (at the cost of a non-negligible software development effort). Depending on the characteristics of the matrix represented by $firstNz$, $nzColumns$ and a, the source loop can be considered to implement two different numerical algorithms. If the matrix is lower triangular, the loop corresponds to a lower triangular solve. There are loop-carried flow dependences, but no antidependences or output dependences. If, however, the matrix is not lower triangular, the same loop may be viewed as one iteration of successive overrelaxation (SOR), which is an iterative algorithm for solving linear systems [2]. In this case, both flow dependences and antidependences exist, although output dependences are absent.

Name	Lower Triangular?	Order	# Nonzeros	# Wavefronts
A	no	100000	1145000	20
B	yes	100000	670000	20
C	no	200000	2356000	50
D	yes	200000	1376000	50

Table 1: Sparse Matrix Characteristics

In this paper, we report results of experiments using matrices with randomly generated sparsity structures [4]. Some important characteristics of these matrices are shown in Table 1. The synthetic matrices used in these measurements provide abundant parallelism in the source loop, so that speedups will not be limited by the lack of parallelism. This enables us to focus more on the performance impact of the data access pattern. Naturally, if the matrix in a given problem does not give rise to sufficient parallelism, we cannot expect to see the same speedups as those reported here.

4.1 Executor Performance

In this section, we look at the parallel speedups achieved by the three executor implementations: baseline, read-write restructuring [5] and complete restructuring. The speedups are calculated based on the execution time of the sequential source loop and the execution time of the executor. These timings were measured for the four matrices described earlier and for two different scheduling policies (i.e., the way in which iterations of each wavefront are allocated to processors). Conceptually the iterations of each wavefront are sorted by their indices. Block scheduling gives each processor an equal-sized contiguous block in the list, while wrap scheduling assigns iterations in the list to processors in a round-robin fashion. The former attempts to

[4] The generation algorithm is parameterized by, among other things, the order of the matrix and the number of wavefronts in the resultant parallel schedule. It can produce a sparsity structure that meets these specifications but is otherwise random. Thus, we can control how much parallelism the source loop contains without resorting to unrealistically regular structures.

[5] In these experiments, read-write restructuring refers to "pure" read-write restructuring as described in Section 3.1 plus the read-only restructuring of the indirection table that represents the index-to-location mapping. Earlier experiments showed that performance would be unacceptable without restructuring the indirection table.

preserve locality, while the latter is advantageous when the cost of each iteration varies according to some simple pattern that could cause load imbalance for block scheduling.

The speedups are plotted in Figure 10. Each curve represents a combination of the scheduling policy and the type of executor implementations: baseline, read-write restructuring, and complete restructuring.

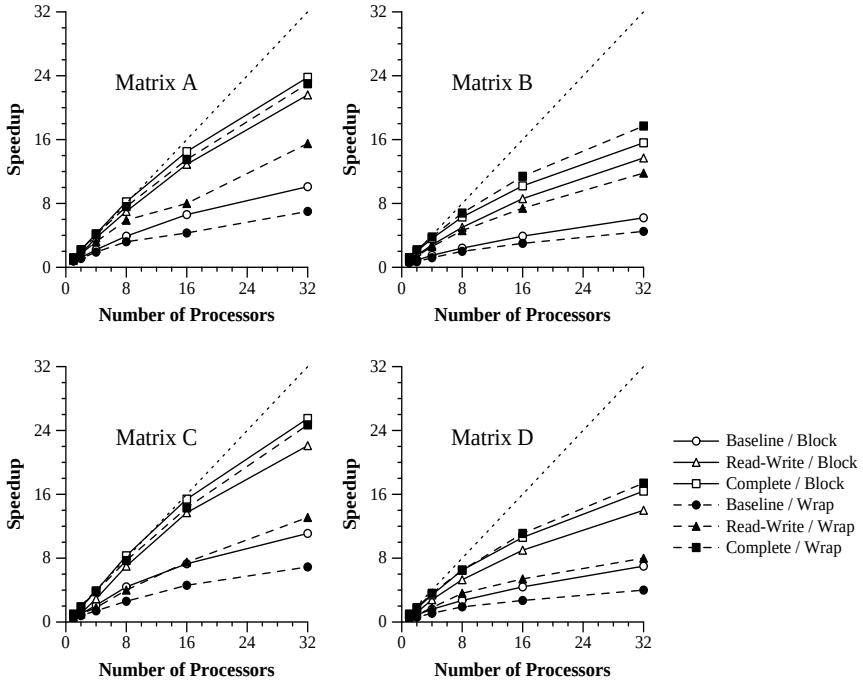


Figure 10: Executor Speedups

We observe that the performance of the baseline executor is far from satisfactory. Its speedup does increase with the number of processors, but only very slowly. Even at 32 processors, the speedup hardly exceeds 10 and is much worse in most cases. This is true for both block scheduling and wrap scheduling, although wrap scheduling is consistently worse.

Comparing the speedups of read-write restructuring to those of the baseline executor, we see a very dramatic improvement in performance. The speedup under both scheduling policies is consistently increased by roughly a factor of two. Since the same relative improvement occurs for block scheduling and wrap scheduling, there remains a wide performance gap between the two.

The improvement is particularly impressive if we recall that read-write restructuring adds a significant amount of work to the executor. Each time the executor has to read an element of the restructured read-write array (i.e., x in the example in Figure 5(a)), it has to perform one more memory read than the baseline executor in order to map the array index to the actual location of the element. Despite this, overall performance increases significantly. This indirectly indicates that the inter-processor communication costs that the baseline executor has to pay for false sharing is indeed very high.

Finally, as we go from read-write restructuring to complete restructuring, performance improves further. In the case of block scheduling, the change is only modest, although clearly evident. For wrap scheduling, it is far more pronounced. The overall result is that, with complete restructuring, wrap scheduling and block scheduling have very similar (though not entirely identical) performance.

The similarity can be explained by the fact that with complete restructuring, the memory access patterns resulting from the two different schedules are almost the same, even though logically the array elements read and written are very different. In both cases, each processor consecutively sweeps through contiguous regions of memory when it reads the read-only arrays and writes the read-write array x. The two data access patterns differ only in how the restructured read-write array is read. Read-write restructuring gives us more control on how its elements are written, but we have much less influence over what the pattern of reads becomes as a result.

The narrowed performance gap between block and wrap scheduling has one important implication. With complete restructuring, poor locality is no longer a necessary consequence of wrap scheduling. Therefore, it can be used much more freely in cases where it is desirable for load balance. More generally, complete restructuring serves to equalize the locality effects of different scheduling policies and thus allows greater flexibility in choosing the right policy for a given situation.

4.2 Preprocessing and Postprocessing Overheads

The improvement in executor performance that we get by means of restructuring techniques must be balanced against the preprocessing and postprocessing costs that come with their use. In this section, we look into this issue.

First, let us consider read-write restructuring. The major tasks that must be done in preprocessing and postprocessing are:

- **indirection table construction.** assigning array elements to appropriate locations according to a given parallel schedule and building an indirection table that represents the mapping,
- **indirection table restructuring.** restructuring the indirection table mentioned above (using read-only restructuring techniques),
- **pre-copying.** copying array elements from their original positions to their temporary locations during loop executions, with restructuring done on the fly, and
- **post-copying.** copying the array elements, which have been updated in the loop, back from their temporary locations to their original positions.

The first three constitute preprocessing, while the last one constitutes postprocessing. Figure 11 shows how much time is spent on each of these tasks, together with the executor times for comparison. The number of processors is 32.

The costs of indirection table construction, pre-copying and post-copying are small relative to the execution time of the executor. The cost of restructuring the indirection table, however, is quite high. It is comparable to the cost of the executor itself, even slightly higher in some cases. These observations should be hardly surprising. The amount of work in building the indirection table and copying elements of the restructured read-write array is roughly proportional to the number of such elements, which in our experiments is the order of the matrix. The amount of work in restructuring the indirection table, however, is proportional to the number of read accesses to the restructured array, which is the number of nonzeros in the matrix. If we add all the costs to the execution time of the executor with read-write restructuring, the total cost is close to, if not greater than, the execution time of the baseline executor. Thus, it would seem that after accounting for all the overheads, read-write restructuring does not really improve performance.

However, we need to look more closely at how often these preprocessing and postprocessing costs are paid. In this respect, we should distinguish between the costs associated with the indirection table and those

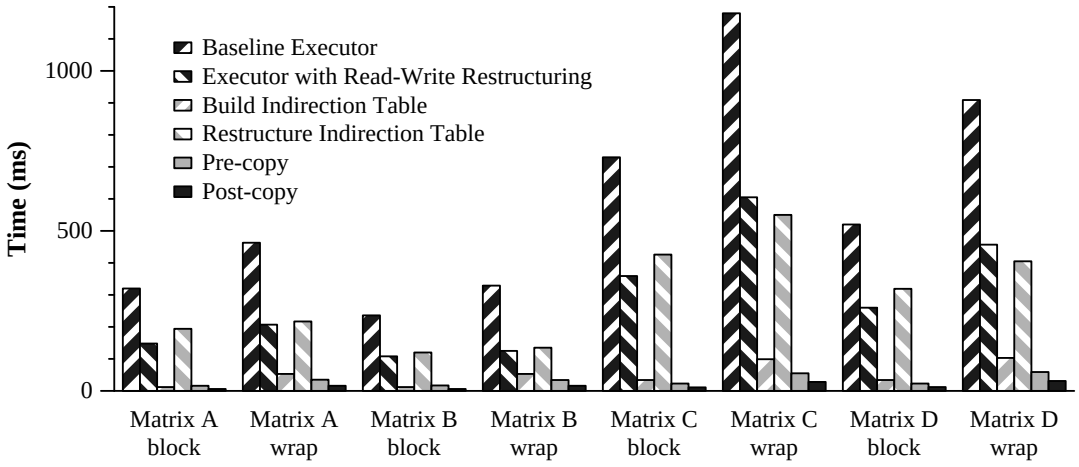


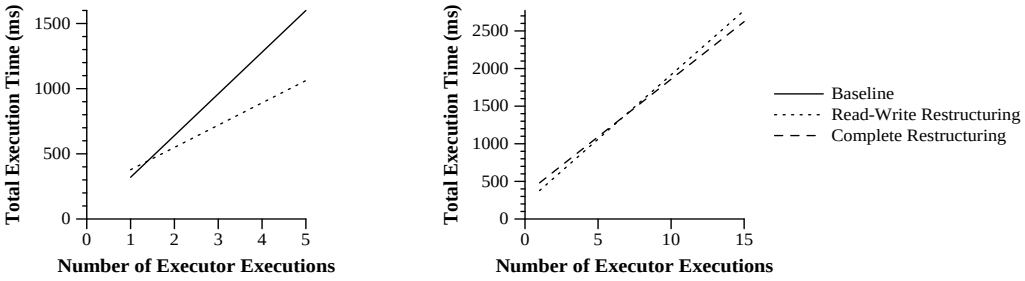
Figure 11: Preprocessing and Postprocessing Costs for Read-Write Restructuring

associated with copying the array. The indirection table is computed from the parallel schedule, which in turn is computed according to loop-carried dependences in the source loop. If the source loop is executed multiple times with the same dependence pattern, the schedule can be reused for each execution and hence the cost of the inspector can be amortized. This occurs, for example, in some iterative solvers for sparse linear systems, where the same triangular system is solved many times with different right-hand sides [8]. Similarly, the indirection table needs not be rebuilt if the schedule has not changed. Thus, the cost of building the indirection table from the schedule and, more importantly, that of restructuring this table can be amortized like the cost of computing the schedule itself.

The cost of copying is potentially paid more often. A straightforward approach would be to pre-copy and post-copy for each executor execution. In fact, this may be acceptable because both costs are small, though certainly not trivial, compared with executor execution time. However, further optimizations are possible. If the source loop is run a number of times without any intervening access to the restructured array, then we need only to pre-copy before the first execution and post-copy after the last. This happens, for example, in the SOR algorithm. SOR is an iterative algorithm in which an "iteration" is one execution of the source loop. Throughout the algorithm, the iterate, which is the array to be restructured, is accessed only by the source loop.

Figure 12(a) illustrates the tradeoff between the baseline executor and the executor with read-write restructuring. The curves are plotted with the execution times for matrix A and block scheduling running on 32 processors. We assume that the total execution time of the baseline executor is simply the time for one execution (measured earlier) multiplied by the number of executions. In the read-write restructuring case, we assume a constant overhead (the time to build and restructure the indirection table) and include the copying costs in the per-execution execution time. From the graph, we see that read-write restructuring does not pay off when the executor is run only once. This is not too surprising considering all the extra work that read-write restructuring has to do. Rather, the surprise lies in the relatively small penalty. In this particular case, if the executor is run two or more times, then read-write restructuring leads to a shorter overall execution time than the baseline approach. In fact, this is true in all the other cases as well.

Now, let us consider the comparison between complete restructuring and read-write restructuring. The tradeoff between the two is similar to that between read-write restructuring and the baseline executor. This is illustrated in Figure 12(b) for matrix A and block scheduling. On one hand, performing read-only restructuring in addition to read-write restructuring increases executor performance. On the other



(a) Baseline Executor vs. Read-Write Restructuring

(b) Read-Write Restructuring vs. Complete Restructuring

Figure 12: Baseline vs. Read-Write Restructuring vs. Complete Restructuring

hand, read-only restructuring requires significant additional preprocessing, although these overheads can be amortized across multiple loop executions as long as the restructured read-only arrays are not written between executions. We need to compare the savings with the extra costs to decide which is the better option in a given situation.

Matrix	Schedule	Read-Write Restructuring		Complete Restructuring		Break-even Point
		Constant (ms)	Per-Execution (ms)	Constant (ms)	Per-Execution (ms)	
A	block	207	171	328	153	7
	wrap	271	259	505	184	4
B	block	134	132	211	113	5
	wrap	189	177	333	129	3
C	block	461	394	836	350	9
	wrap	650	690	1284	412	3
D	block	353	296	531	255	5
	wrap	509	548	989	297	2

Table 2: Read-Write Restructuring vs. Complete Restructuring

Table 2 shows the execution times for read-write restructuring and complete restructuring. In each case, the times are broken down into a constant component and a per-loop-execution component. The break-even point of complete restructuring is the smallest number of loop executions for which the overall execution time for complete restructuring is less than that for read-write restructuring.

We notice that the break-even points are generally smaller for wrap scheduling than for block scheduling. As we saw in the previous section, under wrap scheduling, performing read-only restructuring in addition to read-write restructuring improves executor performance tremendously. Although the preprocessing and postprocessing overheads for wrap scheduling are also significantly higher than those for block scheduling, apparently the overall result is still that it takes relatively few loop executions for complete restructuring to break even. The break-even points for block scheduling are clearly higher but still seem reasonably low in these cases.

Finally, it should be pointed out that all the preprocessing or postprocessing steps mentioned in this section can be completely parallelized with no need for synchronization within each step. In fact, they are, if anything, more parallelizable than the executor itself. Therefore, if the number of processors were increased

beyond the range used in these experiments, the preprocessing and postprocessing costs, both in absolute terms and relative to the time to run the executor, would likely be lower than what we see here.

4.3 Memory Overheads

Restructuring entails significant memory overheads because it is not performed *in situ*. We now look at the amount of memory required. Table 3 shows the memory requirements for matrix A. For each array, we list how much memory is needed by the original array, and by the restructured array when various techniques are applied. Note that actual memory requirements are slightly higher because of other auxiliary data structures, but the tabulated figures serve to illustrate several main issues.

	Memory Requirements (10^3 bytes)			
Array	Baseline	Read-Write Restructuring	Read-Only Restructuring	Complete Restructuring
$firstNz$	400	0	800	800
$nzColumns$	4580	0	4580	0
a	9160	0	9160	9160
b	800	0	800	800
x	800	800	0	800
map	0	4580	0	4580
total	15740	5380	15340	16140

Table 3: Memory Overheads of Restructuring

First, the restructured $firstNz$ array is twice as large as the original since each element is read twice (by two consecutive iterations) and so duplicated. Secondly, the major memory overhead of read-write restructuring is the restructured map array (i.e., the index-to-location indirection table). It is proportional to the number of read accesses to x, rather than just the number of x elements. Although this memory overhead can be vastly reduced if we do not restructure map, earlier experiments showed that the resultant executor performance would be unacceptable due to the irregular access pattern of map. Thirdly, the memory requirement of complete restructuring is less than the sum of those of read-write and read-only restructuring. The reason is that with complete restructuring, the array $nzColumns$ can be ignored as it serves no purpose after map has been restructured. We have already discussed this optimization in Section 3.2.

5 Conclusions

When a sequential loop is parallelized, dependences have to be determined and synchronization inserted to ensure the correctness of a parallel execution of the iterations. However, in order to achieve high performance, we must also pay attention to the organization of the data, especially arrays, accessed by the loop. An organization that is optimal for sequential execution may be poorly suited for parallel execution since the data access patterns of sequential and parallel execution of the same loop can differ significantly. Therefore, parallelizing the loop without changing the way data are structured can severely limit parallel performance.

In this paper, we have looked at this issue in the context of runtime parallelization. We have discussed two problems of parallelizing a sequential loop on a distributed shared-memory multiprocessor without restructuring the data accordingly. Both arise from the fact that in a parallel execution, consecutive iterations are often not executed consecutively as in a sequential execution, but rather by different processors at different

points in time. The first problem is false sharing of read-write arrays. The other is loss of spatial locality and mainly concerns read-only arrays.

We address these problems by restructuring arrays according to the parallel schedule for the iterations. To tackle the false sharing problem, we restructure read-write arrays in such a way that all array elements in a single cache line are guaranteed to be written by the same processor in the same wavefront. This ensures that a cache line is invalidated only once in each loop execution and vastly reduces the number of coherence misses. To deal with the spatial locality problem, we restructure the read-only arrays. In effect, all the read-only array elements that a processor needs are first read and written out sequentially to a temporary area. Later during loop execution, they can be read back sequentially with minimum overhead by the same processor, thus maximizing spatial locality.

We have performed experiments to study the effectiveness of these techniques. The results show that read-write restructuring improves executor performance substantially. It roughly doubles the speedup of the executor in all the cases that we measured. This is achieved despite the extra indirection cost of read-write restructuring. With complete restructuring, executor performance increases further so that it becomes almost independent of how iterations within wavefronts are scheduled. This allows greater flexibility in the choice of scheduling policy for any given situation.

We have also measured the preprocessing and postprocessing overhead that restructuring techniques entail. For both read-write and read-only restructuring, this overhead can be broken down into a constant component and a per-loop-execution component. In our experiments, the per-loop-execution component is relatively small, although the constant component can be quite large. The total overhead cannot be justified by the performance benefits if the executor is run only once. However, if it is run multiple times, the overhead is amortized and restructuring gives rise to a shorter overall execution time. In our case, it takes only two loop executions for read-write restructuring to break even; read-only restructuring takes more but the break-even point is still quite small.

In this paper, we see that parallel execution of a loop without a corresponding change in data organization can lead to adverse cache effects on a distributed shared-memory multiprocessor. We focus in particular on runtime parallelization and find that restructuring arrays according to the computed parallel schedule is a promising technique for alleviating the performance impact of these effects. Not only does it improve the performance of parallel loop execution substantially, but also it does so at the price of an overhead most of which can be amortized across multiple loop executions.

References

[1] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, May 1987.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, 1989.

[3] Henry III Burkhardt, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[4] Kendall Square Research Corporation. *Principles of Operations*, chapter 5. Revision 6.0 edition, October 1992.

[5] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

[6] Shun-Tak Leung and John Zahorjan. Handling general dependences in runtime parallelization. Technical report, Department of Computer Science and Engineering, University of Washington, 1994. In preparation.

[7] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of Supercomputing '92*, pages 104–113, November 1992.

[8] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. In *Proceedings of International Workshop on Compilers for Parallel Computers, Paris*, 1990.

[9] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Runtime parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[10] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, 1991.

[11] Daniel Windheiser, Eric L. Boyd, Eric Hao, Santosh G. Abraham, and Edward S. Davidson. KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *Proceedings of International Parallel Processing Symposium 1993*, April 1993.