Reordering Iterations in Runtime Loop Parallelization

Shun-Tak Leung and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 92-12-07
December 1992

Reordering Iterations in Runtime Loop Parallelization

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

December 1992

Abstract

When a loop in a sequential program is parallelized, it is normally guaranteed that all flow dependencies and anti-dependencies are respected so that the result of parallel execution is always the same as sequential execution. In some cases, however, the algorithm implemented by the loop allows the iterations to be executed in a different sequential order than the one specified in the program. This opportunity can be exploited to expose parallelism that exists in the algorithm but is obscured by its sequential program implementation.

In this paper, we show how parallelization of this kind of loop can be integrated into the runtime parallelization scheme of Saltz et al. [17, 18]. Runtime parallelization is a general technique appropriate for loops whose dependency structures cannot be determined at compile time. The compiler generates two pieces of code: the *inspector* examines dependencies at run time and computes a parallel schedule; the *executor* executes iterations in parallel according to the computed schedule.

In our case, the inspector has to solve two problems: choosing an appropriate sequential order for the iterations and computing a parallel schedule. The two problems are treated as a single graph coloring problem, which is solved heuristically. Two methods to do so are described. Furthermore, the basic runtime parallelization scheme for shared-memory multiprocessors pays no attention to locality when scheduling iterations onto processors. One of our methods takes locality consideration into account when making these decisions. The performance implications of reordering are examined experimentally on a KSR1 parallel machine as well as through a simple analytic model of execution time.

1 Introduction

Scientific applications written in sequential programming languages (e.g., HPF or Fortran D [7]) typically have large loops that can be parallelized to exploit the computing power of multiple processors. However, if a loop contains complicated or data dependent array indexing expressions, the inter-iteration dependencies cannot be fully determined at compile time. Unable to parallelize the loop, the compiler must generate code to execute the loop sequentially. To address this problem, Saltz et al. [17, 18] have proposed runtime parallelization as an alternative. We have been looking into various ways to improve the performance of runtime parallelization [12, 13]. Although runtime parallelization can be carried out on various multiprocessor architectures, we concentrate on its use in shared-memory multiprocessors, scalable versions of which have recently been proposed [4, 11, 2]. This paper presents one aspect of our work.

*Support for this work was provided in part by the National Science Foundation (Grants CCR-8619663, CCR-9123308, and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (Systems Research Center and External Research Program). Authors' addresses: Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195; *shuntak@cs.washington.edu, zahorjan@cs.washington.edu.*

```
for i = 1 to n do
  a[i] = F(a[g(i)], a[h(i)], ...);
enddo
```

Figure 1: Sequential Source Loop

In general, when we parallelize a loop, the sequential order of the iterations has to be preserved in the sense that parallel execution of the iterations respects the dependencies in the sequential loop. However, there are many cases in which the algorithm expressed by the loop in fact permits a different sequential order of executing the iterations than the one specified by the sequential program. Under these circumstances, reordering the iterations and then parallelizing the reordered loop may produce more parallelism than simply parallelizing the original loop.

In this section, we first review the basic idea of runtime parallelization proposed by Saltz et al. [17, 18]. We then discuss the notion of reordering iterations to gain parallelism, and some implications of providing programming support for it.

1.1 Runtime Parallelization of Loops

Let us call the loop being parallelized the *source loop*. The form of source loops we focus on is shown in Figure 1, which is adapted from Saltz et al. [17]. The objective is to identify iterations that can be executed concurrently and assign them to different processors in a shared-memory multiprocessor. In the loop, each array element $a[i]$ in order is assigned a value computed from other elements of a, leading to dependencies that must be respected in a parallel execution of the loop. Naturally, the loop body may contain other statements which do not lead to inter-iteration dependencies, or the source loop may be nested inside another loop. For simplicity, these are not shown.

We are interested in those cases where the dependency pattern is sparse. In other words, each iteration depends on only a small fraction of all iterations. In this case, there are many iterations that share no dependencies and thus may be executed in parallel. However, if the array index expressions $g(i)$, $h(i)$, etc. are complicated functions of i or, worse, involve data not available at compile time, the compiler cannot completely determine the dependencies and therefore cannot parallelize the loop. Sparse matrix operations are good examples of this class of computation. In these operations, $g(i)$, $h(i)$ would be indirection arrays containing indices into the array a, which in turn contains the non-zero elements.

The basic idea of runtime parallelization, due to Saltz et al. [17, 18], is for the compiler to generate two code fragments for each source loop: *inspector* and *executor*. At run time, the inspector calculates the array index functions, determines the inter-iteration dependencies, and use these to compute a parallel schedule for the iterations. The executor then executes the iterations in parallel according to the computed schedule.

The parallel schedule computed partitions the set of all source loop iterations into distinct subsets, called *wavefronts*. The executor goes through the wavefronts sequentially in some specified order but, within each wavefront, the iterations are executed in parallel. A schedule is valid if iterations executed in parallel according to this schedule always achieve the same effect as if they were executed sequentially in the source loop.

In this paper, we call the number of wavefronts in a schedule its *depth*. For any given source loop, there are normally many different schedules with different depths. As a heuristic, we wish to find a schedule with minimal depth because a smaller schedule depth allows the efficient use of larger numbers of processors, and for a fixed number of processors usually (though not necessarily) leads to a shorter parallel execution time.

1.2 Reordering Iterations to Expose Parallelism

Generally, when the source loop is parallelized, we must ensure that the inter-iteration dependencies in the sequential source loop are respected in the parallel execution. When implementing an algorithm as the source loop, the programmer assumes sequential execution of iterations in the order that the program specifies. As the parallelizing compiler has no knowledge of the semantics of the application, it must preserve the sequential iteration order to assure that the algorithm implemented is always correctly executed.

However, there are cases in which the algorithm in fact permits a different sequential order of executing the iterations than the one specified by the sequential program. For example, the class of *Gauss-Seidel* iterative numerical algorithms [3] falls into this category.

Like other types of iterative numerical algorithms, Gauss-Seidel type algorithms find a solution vector $\mathbf{x}$ to some problem by repeatedly computing a new estimate $\mathbf{x}^{(t+1)}$ from an old estimate $\mathbf{x}^{(t)}$ until convergence. The distinguishing characteristics of Gauss-Seidel is that when new values are computed, the "most recent" values are used as they become available [1]:

$$x_i^{(t+1)} = F_i(x_1^{(t+1)}, ..., x_{i-1}^{(t+1)}, x_i^{(t)}, ..., x_n^{(t)})$$

where $x_i^{(t)}$ is the value of the i-th component of $\mathbf{x}^{(t)}$ and we assume that the elements of $\mathbf{x}^{(t+1)}$ are calculated in index order. In a sequential programming language, this is normally implemented as a single data array representing both $\mathbf{x}^{(t)}$ and $\mathbf{x}^{(t+1)}$, and a loop that goes through the array elements one by one in index order. In each iteration, a new value is computed from whatever values are then stored in the array; the result is written into the same array and perhaps used to compute other new values later in the loop.

In many situations, the order of going through the array elements is not critical for correctness [3]. Different orders all produce valid answers, within an acceptable level of accuracy. Intuitively, this is because the components are indexed arbitrarily anyway. Processing them in a different order is conceptually equivalent to re-indexing the components and then processing them in the new index order. For a concrete example, suppose we want to solve the Poisson equation on a rectangular grid. The grid points can be indexed in a row-major manner, a column-major manner, or any other manner we may feel like. There is no inherent reason why it must be done one way or another. The choice is usually just a matter of programming convenience.

If any sequential order of performing the source loop iterations is acceptable, we have extra freedom in deciding how to execute the iterations in parallel. Specifically, it allows us to reorder the source loop iterations so as to reduce the depth of the parallel schedule for the reordered loop and thereby expose parallelism that is inherent in the algorithm but is obscured by its sequential program implementation.

In the mathematical notations above, for generality, every vector component is expressed as a function of all other components. However, in many practical applications, a component actually depends only on a much smaller number of other components. One notable example is sparse matrix operations in which $x_j^{(t+1)}$ depends on $x_i^{(t)}$ or $x_i^{(t+1)}$ only if the corresponding element in a certain coefficient matrix is non-zero. In these cases, changing the sequential order of the iterations can change the resultant schedule and reduce its depth dramatically.

[1] In mathematical parlance, Gauss-Seidel refers to a more restricted form of algorithms than what we describe. However, in this paper we will use Gauss-Seidel in the looser sense defined above, for lack of a better term.

1.3 Support for Iteration Reordering

Researchers in numerical algorithms have long recognized that sometimes more parallelism can be identified if operations in a serial algorithm are performed in an appropriate sequential order. For example, numerous parallel algorithms for solving partial differential equations on rectangular grids have been proposed [16]. They generally obtain parallelism by partitioning the set of grid points in certain regular ways specifically chosen for the problem so that points within the same subset are unrelated and thus can be processed in parallel. The subsets themselves are processed in some sequential order. The overall effect is the same as a serial algorithm that goes through the grid points in a corresponding sequential order.

Given suitable compiler and runtime support, programmers need only to implement a serial algorithm as a sequential loop in the most straightforward manner. The compiler and runtime system can then cooperate to choose an iteration order and parallelize the sequential loop automatically. Moreover, the runtime system can also handle loops having irregular dependency structures or whose dependencies are not known until run time.

However, in a sequential programming language the compiler cannot automatically determine whether the ordering of the iterations can be changed. Since the language is sequential, the programmer has to specify an order, whether or not an order is mandated by the algorithm that the loop implements. The compiler cannot tell if the specified order is what the programmer intends or something the programmer is forced into by the language. Therefore, while the compiler and runtime system can provide support for iteration reordering, the language has to be suitably extended so that the programmer can explicitly indicate that reordering is permissible.

Strictly speaking, different sequential orders for the iterations lead to different results and possibly different convergence rates, but these do not necessarily preclude the reordering of iterations to expose more parallelism. Undoubtedly, iteration reordering leads to different intermediate results. However, if the numerical algorithm converges to a unique solution, the final results are the same, within some tolerance. On the other hand, suppose that what the algorithm converges to or even whether it converges at all is dependent on the iteration order. To deal with this, the programmer either carefully chooses an appropriate order or uses an arbitrary order but handles the problem in other ways. In the first case, certainly the chosen order should be preserved and reordering not permitted. In the second case, the original order (which is arbitrary anyway) is not any better than the new order. The case of convergence rate is similar. If it is known in advance that one order converges significantly faster than others, reordering may not be desirable. The decision of whether to allow reordering always rests with the programmer.

The usefulness of reordering iterations depends on the number of processors available and the cost of barrier synchronization. Reordering can dramatically reduce the schedule depth and hence increase the number of iterations that can potentially be executed concurrently. However, these iterations are actually executed in parallel only if there are enough processors. Therefore, if there are a relatively large number of processors, parallel speedup may improve significantly because previously idle processors can now be put to work. If, on the other hand, the number of processors is only modest, the impact on performance would be less significant. However, having fewer wavefronts still improves performance by reducing the total barrier synchronization overhead and also the load imbalance within each wavefront.

Finally, we note that there are explicitly parallel algorithms which do not correspond to the parallelization of serial algorithms [6, 14]. Since our focus is mainly the parallelization of loops specified in sequential programming languages such as Fortran D [7], these algorithms do not fall into the scope of this paper.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 explains the methods we have used to reorder source loop iterations and at the same time compute a parallel schedule for the reordered loop. Two heuristics will be described. One takes locality into account when reordering the iterations while the other does not. Section 3 presents results of measurements on a small-scale shared-memory multiprocessors and studies the performance implications of schedule depth on massively parallel shared-memory machines. Finally, Section 4 concludes this paper.

2 Reordering Iterations and Computing Parallel Schedule

Logically, the inspector consists of two steps. The first is to choose an order for the source loop iterations; the second is to compute a parallel schedule for the reordered loop. In the first step, the new order is to be chosen so that the depth of the schedule computed in the second step is minimized. As we shall see, these two steps can be combined into a single procedure.

In this section, we first formulate the related problems of choosing an iteration order and computing a parallel schedule as a single graph coloring problem. Then two simple heuristics we have used to solve this problem are described. One of them considers locality when scheduling iterations onto processors, while the other ignores locality but attempts to minimize schedule depth.

2.1 Parallelization as Graph Coloring

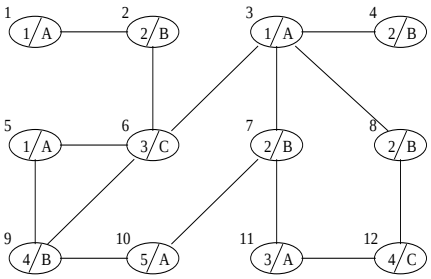
If the sequential order of source loop iterations has to be preserved in parallel execution, the schedule should have this property: if there is a flow dependency or an anti-dependency from iteration k to iteration l, iteration k is in a wavefront *before* that of l [2]. In general, output dependencies impose further restrictions, but the form of source loop we focus on (Figure 1) does not contain output dependencies because each iteration writes a different element of the array. Therefore, output dependencies are not considered in the following discussion.

If we have the freedom to change the source loop's sequential order, it seems possible to simply assign iterations k and l to *different* wavefronts and change the sequential order so that the iteration that is assigned to an earlier wavefront also comes earlier in the new sequential order. In a sense, in order that dependencies are respected in the schedule, the sequential order of the iterations is changed to suit the schedule rather than the other way round. If, say, $k < l$ and iteration k writes an array element read by l, iteration l may or may not be reading the value written by k, depending on the order of the iterations. Given a specific order, if l is before k, then the value which l should read is the one stored in the array element before the current pass, *not* the one k computes in that pass. In this case, placing k in a later wavefront becomes legitimate.

More specifically, the problems of reordering the iterations *and* computing a parallel schedule for the reordered source loop can be formulated as a single graph coloring problem. This was motivated by the problem-specific ordering schemes used in many parallel numerical algorithms (see Section 1.3) and by a general technique that does not handle anti-dependencies [3]. In our graph, nodes represent iterations and edges represent potential dependencies. If iteration k writes to an array element read by iteration l, then nodes k and l are connected by an undirected edge.

[2] Saltz et al. [17, 18] make special provisions in their executor to deal with anti-dependencies so their schedule in fact respects only flow dependencies.

colored graph



original iteration order

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

new iteration order

1, 3, 5, 10, 11	2, 4, 7, 8, 9	6, 12

parallel schedule for original loop

wavefronts:	1	2	3	4	5
iterations:	1	2	6	9	10
	3	4	11	12	
	5	7			
		8			

parallel schedule for reordered loop

colors:	A	B	C
wavefronts:	1	2	3
iterations:	1	2	6
	3	4	12
	5	7	
	10	8	
	11	9	

Figure 2: Computing a Schedule by Graph Coloring

To compute a schedule, the nodes are colored in such a way that adjacent nodes have different colors. All iterations of the same color constitute a wavefront in the parallel schedule. The wavefronts are arranged in an arbitrary order; any ordering is acceptable although each ordering of the wavefronts corresponds to a different reordered source loop. Given a specific ordering of wavefronts, the reordered source loop consists of the iterations (in, say, index order) in the first wavefront, followed by those in the second, etc.

Figure 2 illustrates an example source loop with twelve iterations. In the graph shown, the ovals represent iterations and the edges the dependency relations among them. The numerical labels outside the ovals give the iteration indices. Inside each oval, the left label is that iteration's wavefront in the parallel schedule for the original source loop. The right labels specify a valid coloring (in fact, the one that results from the coloring heuristic we use, described in Section 2.2). In this example, coloring reduces the depth of the schedule from five to three. Without reordering, iteration 9 must be in wavefront 4 because of its dependency on iteration 6. Reordering puts iteration 9 before iteration 6 and thus allows it to be executed in wavefront 2. Other iterations assigned to wavefronts 4 and 5 in the schedule for the original loop are treated similarly.

Note that while we show a new sequential order of the iterations that is consistent with the parallel schedule obtained by coloring, this new order is never explicitly generated by the coloring algorithm. All we need to know is that the parallel execution according to the schedule behaves as if the reordered source loop were executed sequentially.

It is not difficult to see why the parallel schedule is valid for the reordered source loop. If, in the reordered loop, there is a flow dependency or an anti-dependency from iteration k to iteration l (there can be no output dependencies), one of them must write an array element read by the other. Therefore, their corresponding

```
for i = 1 to n do
  C = empty set
  for j such that node j is adjacent to i do
    add wf[j] to C
  enddo
  wf[i] = smallest positive integer not in C
enddo
```

Figure 3: Coloring Heuristic

nodes are connected. The two nodes will have different colors and thus the two iterations are in different wavefronts. In other words, k's wavefront must be either before or after l's. Recall that the order of iterations in the reordered source loop follows the order of wavefronts in the parallel schedule. Thus, if, say, iteration l is after iteration k in the reordered loop, l's wavefront must be after k's wavefront. Flow and anti-dependencies in the reordered loop are therefore respected in the parallel schedule.

2.2 Reordering to Minimize Schedule Depth

Naturally, we wish to minimize the number of wavefronts. Since there is a one-to-one correspondence between colors and wavefronts, minimizing schedule depth is equivalent to using the fewest possible colors. Unfortunately, the problem of finding the minimum number of colors needed is NP-complete [8]. It is doubtful that a reasonably efficient algorithm to do this can be found.

Because of this, we have used a simple coloring heuristic, as shown in Figure 3. Colors are represented by positive integers. Starting from a colorless graph, the heuristic examines the nodes one by one and assigns to each node the "smallest" color (i.e. the smallest positive integer) different from the colors, if any, of all its adjacent nodes. After all the nodes have been colored, no two connected nodes will have the same color because the node colored later in the heuristic must have a color different from that of the node colored earlier.

2.2.1 Implementation

A straightforward implementation of this heuristic involves complex dynamic data structures representing the graph and color sets, as well as time-consuming graph traversal. This would be highly inefficient. To minimize the cost of the inspector, our actual implementation uses arrays and loops; it requires only a single pass over the iterations in some order, although multiple passes may be needed in some very rare cases.

Our implementation is outlined in Figure 4. A compiler can generate it from the source loop automatically. In the following discussion on this implementation, we refer to the order in which iterations are processed *by the inspector* when we say that an iteration is "earlier" or "later" than another. In the figure as well as in our actual code, the iterations are processed in the original index order. This simplifies explanation and coding, but is not strictly necessary; the iterations could have been processed in any order, though there seems no reason to choose any order other than the original one.

Note that node j is adjacent to node i either because iteration i reads $a[j]$ (which iteration j writes) or vice versa. In our implementation, $lr[i]$ is the set of colors denied to iteration i because they have already been used for *earlier* iterations that read $a[i]$. When the color of iteration i is to be selected, $lr[i]$ is consulted to

```
/* lr is initially all empty sets */
for i = 1 to n do
  C = lr[i]
  add wf[g(i)], wf[h(i)], etc.  to C
  wf[i] = smallest positive integer not in C
  add wf[i] to lr[g(i)], lr[h(i)], etc.
enddo
```

Figure 4: Implementation of Coloring Heuristic

find out what these colors are. Furthermore, iteration i itself reads $a[g(i)]$, $a[h(i)]$, etc. If iterations $g(i)$, $h(i)$, etc. have already been processed earlier and assigned colors, iteration i cannot use these colors either. After gathering necessary information, we pick a color for iteration i to satisfy all these constraints. Then, $wf[i]$ and relevant $lr[j]$'s are udpated to supply the necessary information when we handle later iterations.

Representation of color sets has to satisfy two conditions. In the common case, only a handful colors are needed for all iterations. The data structure must facilitate operations like "add a color to set", "find smallest color *not* in set", etc. On the other hand, in the worst case, a different color is needed for each iteration, as when every iteration reads all elements. We do not want to impose an artificial limit on the number of colors that can be handled.

To achieve these ends, we represent a color set as an integer, which is interpreted as a 32-bit bitmap. It is extremely unlikely that more than 31 colors are needed. Nevertheless, if none of the colors 1 through 31 can be chosen for an iteration, it is assigned color 32 and linked to a list of such iterations. After going through all iterations, those of color 32 are processed again. In this second pass, iterations that have not been colored 32 in the first pass are ignored. Again, colors are represented as 32-bit bitmaps except that the least significant bit now means color 32, not color 1. This process can be repeated as many times as necessary until all iterations have been properly colored. In each pass, adding a color to a set is simply an *or* operation while finding the "smallest" color not in a set involves a few arithmetic shifts and bitwise *and*s. All these can be performed very efficiently and no penality is suffered by the common case in order to handle the unlikely cases.

2.2.2 Discussion

Although this heuristic is not guaranteed to use the smallest possible number of colors, it does tend to use very few colors in practice. If each node has degree at most d, this heuristic will use no more than $d+1$ colors. In the worst case, when the heuristic processes a node of degree d, all its d adjacent nodes have been colored with different colors. Even so, there must be a color among 1 to $d+1$ that can be picked for this node without violating the constraint that adjacent nodes must have different colors.

Recall that an edge connects two nodes if one node reads an array element written by the other. Therefore, the degree of node k is at most the number of distinct array elements read by iteration k plus the number of iterations which read $a[k]$. Both are likely to be much smaller than the total number of iterations. So is d and hence $d+1$, the upper bound on the number of wavefronts.

Another interesting observation on this heuristic is that the parallel schedule it computes has the fewest wavefronts *among schedules valid for its corresponding reordered source loop.* A practical implication of this is that there is no point in generating the new ordering of iterations explicitly and finding a parallel schedule for it by applying some inspector algorithm. Even if we do, we cannot further decrease the number

of wavefronts.

To see that the schedule indeed has the fewest possible wavefronts among valid schedules, let us assume that there exists a valid schedule with even fewer wavefronts. Let $w_S(i)$ be the wavefront to which iteration i belongs according to schedule S. Let C be the schedule computed by the coloring heuristic and H be the hypothetical schedule. Since H has fewer wavefronts than C, there must be one or more iterations i such that $w_H(i) < w_C(i)$. Suppose m is the first such iteration. Note that being "first" means appearing first in the reordered source loop, *not* having the smallest iteration index.

Since $w_C(m) > w_H(m) > 0$, we have $w_C(m) > 1$. The coloring heuristic's decision to assign iteration m to wavefront $w_C(m)$ must be that some iterations which have dependencies with m and are examined before m are given colors 1, 2, ..., $w_C(m) - 1$. For any such iteration k, $w_C(k) < w_C(m)$ and hence k is before m in the reordered source loop. Let l be one of these iteration such that $w_C(l) = w_H(m)$. There are two cases to consider:

- $w_H(l) < w_C(l)$. This leads to contradiction because iteration m is supposed to be the first iteration satisfying this condition but l is before m.
- $w_H(l) \geq w_C(l)$. In this case, there is a flow dependency or an anti-dependency between iterations l and m. Moreover, l is before m in the reordered source loop. However, $w_H(l) \geq w_C(l) = w_H(m)$. The schedule H is therefore invalid. If $w_H(l) = w_H(m)$, iterations l and m will be executed in the same wavefront and there is nothing to ensure that they are performed in the right order. If $w_H(l) > w_H(m)$, it is even worse. Iteration l, which appears earlier in the reordered source loop, is executed after iteration m. They are guaranteed to be executed in the wrong order.

Finally, we note that this heuristic automatically produces some common grid point ordering schemes used in the SOR solution of partial differential equations on rectangular grids. For instance, the SOR method with 5-point stencils can be parallelized using the classical red-black ordering [9] while 9-point stencils require a similar ordering with four colors[1]. If the programmer implements the SOR algorithm in these cases as a simple loop going through the grid points row by row, our coloring heuristic will automatically reorder the iterations to achieve the same schedule as those commonly used orderings.

2.3 Reordering for Locality

In the previous section, we presented a graph coloring heuristic whose goal was to minimize schedule depth, with the motivation being in part that this minimizes barrier synchronization overhead. In this section we present an alternative heuristic whose goal is to minimize overhead due to a degradation in locality that often accompanies replacing a sequential execution with a parallel one. This often arises because the data structures are laid out in a way that corresponds to the sequential code. Therefore, it may be beneficial to try preserving the original order while reordering iterations to expose parallelism. What we need is a method to strike a balance between these two seemingly conflicting goals.

The key to scheduling for locality is to notice that when we have to pick a color for iteration k, the only constraint is that the color chosen is different from colors of adjacent nodes. In the previous section, we always choose the "smallest" candidate color in an attempt to minimize the total number of colors used. A different strategy that tends to better preserve the original sequential order is to choose for iteration k the color of iteration $k-1$ if that color is permitted; otherwise, choose the "smallest" permissible color, as before. If, using this scheme, iterations k and $k-1$ have been assigned to the same wavefront, they are further assigned to the same processor so that this processor can execute iterations $k-1$ and k consecutively. This increases the likelihood that consecutive iterations in the sequential source loop are also executed consecutively in the parallel execution.

The implementation is almost identical to that of the previous heuristic. The only change is that, in Figure 4, $wf[i-1]$ is checked to see if it is in C. If not, it is selected as $wf[i]$, the color of iteration i.

This new strategy may lead to more or fewer wavefronts than the earlier strategy of always choosing the "smallest" permissible color, since the latter is only a heuristic. Superficially, it seems that more wavefronts may be needed since this new strategy is less aggressive in using as few colors as possible. However, the number of wavefronts is still bounded by $d+1$, where d is the maximum degree of a node. This is because, as in the earlier strategy, a new color is never used unless there are no other alternatives. Since d is normally very small relative to the number of iterations, the resultant number of wavefronts is not likely to be a major concern.

3 Performance

In this section, we compare the impact on executor performance of three different ways of computing a parallel schedule for the source loop:

- **no reordering.** The original sequential order of the source loop iterations is preserved in the parallel execution. The computed schedule has minimal depth among schedules valid for the original order.
- **locality-oblivious reordering.** The source loop iterations are reordered and the schedule computed using the coloring heuristic described in Section 2.2. The "smallest" permissible color is always chosen.
- **locality-sensitive reordering.** The source loop iterations are reordered and the schedule computed using the coloring heuristic in Section 2.3. If possible, the color of the previous iteration is used.

First, we present some measurement results on a shared-memory multiprocessor with tens of processors. These results serve to illustrate the potential performance advantage of iteration reordering. We then use a simple analytic model of execution time to study the performance implications of iteration reordering on massively parallel machines with up to thousands of processors.

3.1 Measurements on a Small-Scale Multiprocessor

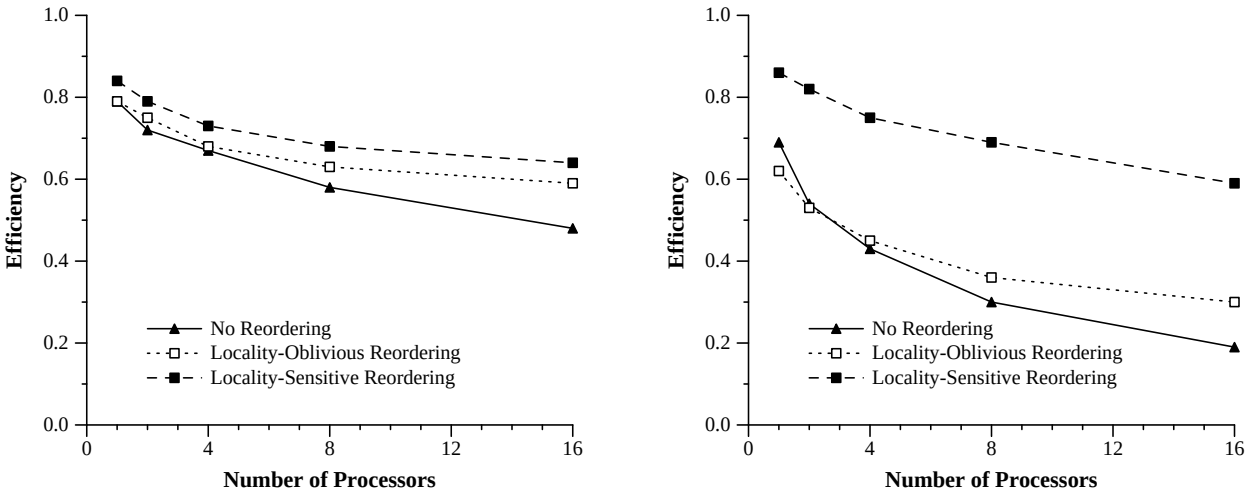
The application that we used is the solution of a sparse linear system by means of Successive Over-Relaxation (SOR). The linear equations are the global balance equations [10] corresponding to a queueing network model with blocking [15]. The model itself consists of a number of service centers in series, each with a finite capacity queue. Such models are commonly used to evaluate the performance of communication networks built from switches with finite buffer space.

The results for two models are presented here. Model A has a 174090×174090 coefficient matrix with about 1146000 non-zero elements. Model B has a 45676×45676 matrix with about 252000 non-zero elements [3]. Measurements were taken on a Kendall Square Research KSR1 shared-memory multiprocessor [4] running OSF/1. All programs were written in C using KSR1's *pthreads*. The translation of the source loop into the inspector and the executor was done manually.

[3] In fact, global balance analysis leads to a coefficient matrix which is sparse except for one dense row whose elements are all equal to one. In a naive implementation of SOR, this row would become a sequential bottleneck. We assume that this row is handled in a separate loop, which in fact is the typical approach even for sequential execution since it avoids the storage of a row of 1's. The source loop we measured is the main loop that deals with the other rows each having only a handful of non-zeros. The number of non-zeros here does *not* include the special row.

Reordering Strategy	Model A		Model B	
	Schedule Depth	Average Iterations per Wavefront	Schedule Depth	Average Iterations per Wavefront
No Reordering	26	6696	67	682
Locality-Oblivious Reordering	6	29015	8	5710
Locality-Sensitive Reordering	4	43522	5	9135

Table 1: Schedules from Different Reordering Strategies



(a) Model A (174090 × 174090)

(b) Model B (45676 × 45676)

Figure 5: Executor Efficiencies for Different Reordering Strategies

Table 1 shows the depths of schedules computed by the three methods and the average numbers of iterations per wavefront. We can see a dramatic decrease in schedule depth brought about by reordering the source loop iterations. There is also a corresponding increase in the number of iterations per wavefront. Notice that locality-sensitive reordering, which seems to be less "aggressive" in using as few wavefronts as possible, may actually lead to fewer wavefronts.

Figure 5 shows the measured executor efficiencies for different reordering strategies. Efficiency is defined as parallel speedup divided by the number of processors. Parallel speedup is the execution time of the best sequential implementation to the time of parallel execution using the executor.

First of all, we note that the efficiencies, in all cases, are far from ideal. We believe that this has to do with the extremely high cost of a local cache miss on the KSR1 (stalling a processor for 150 cycles) and hence the need to preserve locality when iterations are assigned to different processors for execution. We have been exploring various other techniques to improve the performance of parallel loop execution, but these do not fall within the scope of this paper.

Since the machine used for measurements has only a modest number of processors, we do not expect iteration reordering to dramatically improve executor performance by identifying enough parallel work to keep otherwise idle processors busy. This effect can, however, be felt on massively parallel machines having thousands of processors.

Nevertheless, in the measurements presented here, iteration reordering still produces significant performance benefits. Not surprisingly, efficiency declines with the number of processors in all cases. However, the efficiency of no reordering declines faster than that of iteration reordering. At 8 or more processors, no reordering clearly has the lowest efficiency among the three options considered here. Since there are already plenty of iterations per wavefront per processor even without reordering, we believe that the improvement is due to a smaller total cost of barrier synchronization.

Notice that at 1 processor, executions of the same source loop using different schedules can have very different efficiencies, all of which are potentially much lower than 100%. This is partly because in our application, the sparse matrix data structures are laid out in such a way that executing iterations in the order of the original sequential loop best preserves data locality. When iterations are executed in wavefronts, even on only one processor, successive iterations in the source loop may not be executed in sequence. This results in poorer locality and hence degraded performance. Furthermore, different execution orders correspond to different execution times. Locality-sensitive reordering attempts to exploit this by placing successive iterations in the same wavefront as far as possible so that they can later be assigned to the same processor for execution in sequence.

Looking at the performance of locality-sensitive reordering, we observe that it is consistently better than the other two options. For model A, the difference is relatively small (but clearly evident). Conceivably, one may suspect that its superior performance here is only because it happens to yield the schedule with the fewest wavefronts. The same cannot be said of model B. Locality-sensitive reordering performs much better than both no reordering and locality-oblivious reordering. If its superiority over locality-oblivious reordering were due to a difference of 3 wavefronts, then the improvement of locality-oblivous reordering over no reordering should have been much much greater.

In conclusion, we see that in these experiments, reordering does improve performance at larger numbers of processors, even though we do not expect to observe its full potential on a small-scale multiprocessor. Moreoever, changing the order of the iterations can adversely affect locality and thus lead to lower performance, but this may be remedied in part by taking locality into account when we select the new iteration order.

3.2 Performance on Massively Parallel Machines

In this section, we use a simple model of execution time to study the performance implications of iteration reordering on massively parallel shared-memory machines. Examples of such machines are the Kendall Square Research KSR1 [4] and the Tera Computer Corporation multiprocessor design.

We assume that the time it takes to run the executor once consists of two components. The first is the time to execute the iterations themselves; the second is a constant synchronization overhead for each wavefront.

We first derive an expression for parallel execution time, $T(P)$, of I iterations on P processors in W wavefronts. Let t_i and t_b be respectively the time to execute one iteration and the per-wavefront synchronization overhead. We assume that all iterations take the same time to perform. To get the first component of execution time, we assume that the I iterations are roughly equally divided among W wavefronts so that each wavefront has, on the average, $\frac{I}{W}$ iterations and thus takes $t_i\lceil\frac{I}{WP}\rceil$ to complete. The second component is simply t_bW. The parallel execution time can then be written as:

$$T(P) = t_i \left\lceil \frac{I}{WP} \right\rceil W + t_b W$$

The per-wavefront synchronization overhead is due to barrier synchronization between successive wavefronts. First, executing the barrier code takes time. The importance of this would be small, though, if there is direct hardware support for barrier operations, as in new machines like the Thinking Machines CM-5 [5]. Secondly,

since all processors must arrive at the barrier before any one of them can pass through and proceed to the next wavefront, processors arriving earlier have to wait idly for the last arrival. Two processors may arrive at different times because one of them has one more iteration to execute than the other. The first component of the execution time has already accounted for this by means of the ceiling operation. However, even if two processors have exactly the same number of iterations, one may finish later than the other because of delays caused by random events like bus contention. This is attributed to the per-wavefront synchronization overhead.

As a first approximation, we account for these two factors with a constant overhead per wavefront. In fact, it is likely that the overhead due to these factors increases with the number of processors. For example, in our barrier implementation, the time for all processors to go through the barrier code (assuming simultaneous arrivals) is of the order of $\log P$. If the overhead does increase with P, reducing schedule depth when there are many processors would be even more important than this simple model may suggest. Thus, our analysis is conservative with respect to the benefit of reducing schedule depth.

Next we find the sequential execution time of the source loop. Obviously, the implementation of the loop body in the executor would be more complicated than its implementation in the original sequential loop. For simplicity, however, we assume that the two implementations take the same time to execute. If they differ significantly in execution time, the effect is roughly to scale all parallel efficiencies by a constant factor, which does not affect our conclusions on how schedule depth affects parallel efficiency. With this assumption, the sequential execution time is simply:

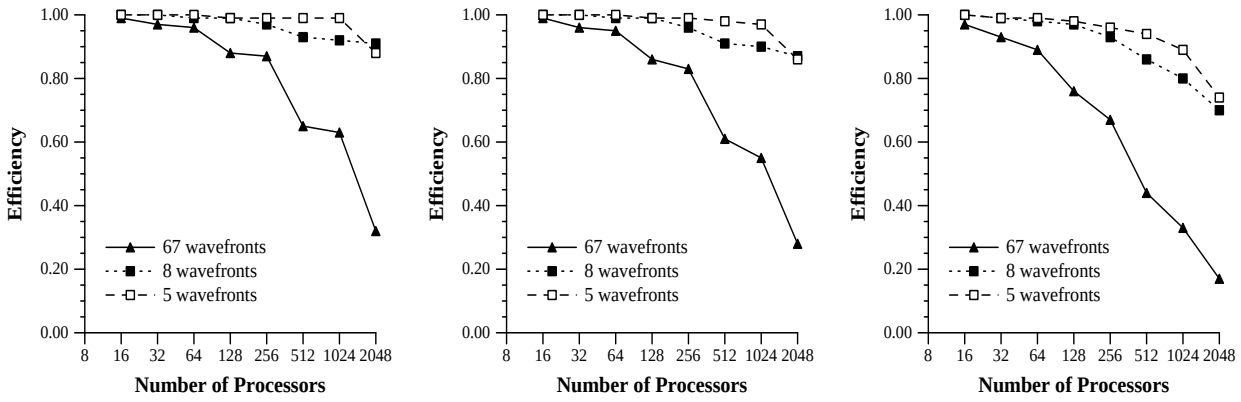
$$T_s = t_i I$$

The efficiency, defined as parallel speedup divided by the number of processors, is thus:

$$E(P) = \frac{T_s/T(P)}{P} = \frac{t_i I}{t_i \left\lceil \frac{I}{WP} \right\rceil + t_b} \frac{1}{WP}$$

In Figure 6, we plot efficiency (E) against the number of processors (P) for different numbers of wavefronts (W) and different values of t_b/t_i. The numbers of iterations and wavefronts are chosen from model B described in the previous section. The value of t_b/t_i is a function of both the source loop body and how barriers are implemented. In our case, the ratio is roughly of the order of 10. Curves for lower values of t_b/t_i are plotted so as to see how efficiency varies with the number of processors on machines with faster barriers and/or for applications having more time-consuming source loop body.

When there are only a dozen or so processors, executor efficiency depends very little on schedule depth. Reordering can significantly reduce the depth but this does not have much effect on performance. At the other extreme, when there are thousands of processors, the number of wavefronts becomes very important. For example, with $t_b/t_i = 1.0$ and $P = 2048$, the efficiency for 8 wavefronts is about 4 times the efficiency for 67 wavefronts. Naturally, for lower values of t_b/t_i (because barriers are faster and/or iterations take longer than our case), the difference is smaller but still extremely significant. When there are so many processors, the main reason of inefficiency is not the synchronization overhead but starvation experienced by some processors. Therefore, if changing the order of source loop iterations helps to reduce the number of wavefronts and hence identify more work that can be done in parallel, it can bring about significant performance improvement on massively parallel machines.



(a) $t_b/t_i = 0.05$ (b) $t_b/t_i = 0.2$ (c) $t_b/t_i = 1.0$

Figure 6: Executor Efficiencies for Different Numbers of Wavefronts

4 Conclusion

In general, when a loop specified in a sequential programming language is parallelized, all its flow dependencies and anti-dependencies must be respected to ensure that the algorithm implemented by the loop is always correctly executed. The effect of parallel execution is as if the original loop were executed sequentially. However, sometimes the sequential order of the iterations can be legitimately changed, as in loops implementing Gauss-Seidel iterative numerical algorithms. This gives us extra freedom in choosing how to perform the iterations in parallel. In particular, the original sequential order can be changed so as to reduce the depth of the parallel schedule for the reordered loop.

The parallelization of these loops can be done within the framework of the runtime parallelization scheme proposed by Saltz et al. The inspector chooses a new order of the iterations and computes a parallel schedule for the reordered loop. These two problems are solved together as a single graph coloring problem. Though finding an efficient algorithm to minimize the number of wavefronts is unlikely, we have used two related heuristics that work reasonably well. They both guarantee that no more than $d + 1$ wavefronts are needed if each iteration is involved in at most d flow dependencies and anti-dependencies. Moreover, one of the heuristics attempts to take locality into consideration when producing the schedule.

Measurements were made on a small-scale shared-memory multiprocessor. Reordering source loop iterations does bring some performance improvement. A side effect, however, is that the reordered loop may have less desirable locality properties. Taking locality into account can help to alleviate this effect. Furthermore, we used a simple execution time model to study the relationship between schedule depth and executor efficiency on machines with many more processors. With thousands of processors, schedule depth becomes extremely important. Reordering the iterations helps to keep schedule depths down and thus could lead to much better executor performance.

References

[1] L. Adams and J. Ortega. A mutli-color SOR method for parallel computation. In *Proceedings of 1982 International Conference on Parallel Processing*, pages 53–56, August 1982.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of International Conference on Supercomputing*, pages 1–6, 1990.

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

[4] 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.

[5] *CM-5 Technical Summary*. Thinking Machines Corporation, Cambridge, MA., 1991.

[6] R. De Leone and O. L. Mangasarian. Asynchronous parallel successive overrelaxation for the symmetric linear complementarity problem. *Mathematical Programming*, 42(2):347–361, November 1988.

[7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran d language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[9] Jules J. Lambiotte, Jr. and Robert G. Voigt. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *ACM Transactions on Mathematical Software*, 1(4):308–329, December 1975.

[10] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice Hall, Englewood Cliffs, 1984.

[11] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[12] Shun-Tak Leung and John Zahorjan. Improving executor performance in runtime loop parallelization. Technical report, Department of Computer Science and Engineering, University of Washington, 1992. In preparation.

[13] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993. To appear.

[14] O. L. Mangasarian and R. De Leone. Parallel successive overrelaxation methods for symmetric linear complementarity problems and linear programs. *Journal of Optimization Theory and Applications*, 54:437–446, 1987.

[15] Raif O. Onvural. Survey of closed queueing networks with blocking. *ACM Computing Surveys*, 22(2):83–121, June 1990.

[16] J. Ortega and R. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(2):149–240, June 1985.

[17] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

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