A Scalable Method for Run–Time Loop Parallelization

Lawrence Rauchwerger[†][§]
Center for Supercomputing R&D
University of Illinois
rwerger@csrd.uiuc.edu

Nancy M. Amato[‡]
Department of Computer Science
Texas A&M University
amato@cs.tamu.edu

David A. Padua[§]
Center for Supercomputing R&D
University of Illinois
padua@csrd.uiuc.edu

Corresponding Author: Lawrence Rauchwerger phone: (217) 244-0070 fax: (217) 244-1351
CSRD, Univ. of Illinois, 1308 W. Main St., Urbana IL 61801

Abstract

Current parallelizing compilers do a reasonable job of extracting parallelism from programs with regular, well behaved, statically analyzable access patterns. However, they cannot extract a significant fraction of the available parallelism if the program has a complex and/or statically insufficiently defined access pattern, e.g., simulation programs with irregular domains and/or dynamically changing interactions. Since such programs represent a large fraction of all applications, techniques are needed for extracting their inherent parallelism at run–time. In this paper we give a new run–time technique for finding an optimal parallel execution schedule for a partially parallel loop, i.e., a loop whose parallelization requires synchronization to ensure that the iterations are executed in the correct order. Given the original loop, the compiler generates *inspector* code that performs run–time preprocessing of the loop's access pattern, and *scheduler* code that schedules (and executes) the loop iterations. The inspector is fully parallel, uses no synchronization, and can be applied to any loop (from which an inspector can be extracted). In addition, it can implement at run–time the two most effective transformations for increasing the amount of parallelism in a loop: *array privatization* and *reduction parallelization* (element–wise). The ability to identify privatizable and reduction variables is very powerful since it eliminates the data dependences involving these variables and thereby potentially increases the overall parallelism of the loop. We also describe a new scheme for constructing an optimal parallel execution schedule for the iterations of the loop. The schedule produced is a partition of the set of iterations into subsets called *wavefronts* so that there are no data dependences between iterations in a wavefront. Although the wavefronts themselves are constructed one after another, the computation of each wavefront is fully parallel and requires no synchronization. This new method has advantages over all previous run–time techniques for analyzing and scheduling partially parallel loops since none of them has all of these desirable properties.

[†] Research supported in part by Intel and NASA Graduate Fellowships.

[§] Research supported in part by Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

[‡] Research supported in part by an AT&T Bell Laboratories Graduate Fellowship, and by the International Computer Science Institute, Berkeley, CA.

1 Introduction

To achieve a high level of performance for a particular program on today's supercomputers, software developers are often forced to tediously hand-code optimizations tailored to a specific machine. Such hand-coding is difficult, error-prone, and often not portable to different machines. Restructuring, or parallelizing, compilers address these problems by detecting and exploiting parallelism in sequential programs written in conventional languages. Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades [26, 41], current parallelizing compilers cannot extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. This is an extremely important issue because a large class of complex simulations used in industry today have irregular domains and/or dynamically changing interactions. For example, SPICE for circuit simulation, DYNA–3D and PRONTO–3D for structural mechanics modeling, GAUSSIAN and DMOL for quantum mechanical simulation of molecules, CHARMM and DISCOVER for molecular dynamics simulation of organic systems, and FIDAP for modeling complex fluid flows [8].

Thus, since the available parallelism in theses types of applications cannot be determined statically by present parallelizing compilers [6, 8, 11], compile-time analysis must be complemented by new methods capable of automatically extracting parallelism at *run–time*. The reason that run–time techniques are needed is that the access pattern of some programs cannot be determined statically, either because of limitations of the current analysis algorithms or because the access pattern is a function of the input data. For example, most dependence analysis algorithms can only deal with subscript expressions that are linear in the loop indices. In the presence of non–linear expressions, a dependence is usually assumed. Compilers usually also conservatively assume data dependences in the presence of subscripted subscripts. More powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically-known values. However, nothing can be done at compile-time when the index arrays are a function of the input data [21, 35, 43].

Run–time techniques have been used practically from the beginning of parallel computing. During the 1960s, relatively simple run–time techniques, used to detect parallelism between scalar operations, were implemented in the hardware of the CDC 6600 and the IBM 360/91 [37, 38]. Various synchronization schemes have been proposed to delay execution until certain conditions are satisfied. For example, the HEP multiprocessor [36] has a full/empty bit associated with each memory location and read (write) accesses are delayed until the bit is full (empty). Similar data–level synchronization schemes have also been proposed [13, 27]. Higher–level synchronization primitives such as *lock* or *compare–and–swap* can be used in the same manner [15, 24, 43]. When parallelizing `do` loops, some of today's compilers postpone part of the analysis to run–time by generating two-version loops. These consist of an `if` statement that selects either the original serial loop or its parallel version. The boolean expression in the `if` statement typically tests the value of a scalar variable.

During the last few years, techniques have been developed for the run–time analysis and scheduling of loops [5, 9, 16, 21, 24, 28, 33, 34, 30, 31, 32, 35, 42, 43]. The majority of this work has concentrated on developing run–time methods for constructing execution schedules for partially parallel loops, i.e., loops whose parallelization requires synchronization to ensure that the iterations are executed in the correct order. Given the original, or *source* loop, most of these techniques generate *inspector* code that analyzes, at run–time, the cross-iteration dependences in the loop, and *scheduler/executor* code that schedules and executes the loop iterations using the dependence information extracted by the inspector [35].

1.1 Our Results

In this paper we give a new inspector/scheduler/executor method for finding an optimal parallel execution schedule for a partially parallel loop. Our inspector is fully parallel, uses no synchronization, and can be applied to any loop (from which an inspector can be extracted). In addition, our inspector can implement at run–time the two most effective transformations for increasing the amount of parallelism in a loop: array privatization and reduction parallelization (element–wise). The ability to identify privatizable and reduction variables is very powerful since it eliminates the data dependences involving these variables. Thus, in addition to increasing the available parallelism in the loop these dependence removing transformations also reduce the work required of the scheduler, i.e., it need not consider the affected variables. We describe a scheme for constructing an optimal parallel execution schedule for the iterations of the loop. The schedule produced is a partition of the set of iterations into subsets called *wavefronts*, so that the iterations in each wavefront can be executed in parallel, i.e., there are no data dependences between iterations in a wavefront. Although the wavefronts themselves are constructed one after another, the computation of each wavefront is fully

```
do i = 2, n
  A(K(i)) = A(K(i)) + A(K(i-1))
  if (A(K(i))) then
    ....
  endif
enddo
```

(a)

```
    do i = 1, n/2
S1:   tmp = A(2*i)
      A(2*i) = A(2*i-1)
S2:   A(2*i-1) = tmp
    enddo
```

(b)

```
    do i = 1, n
      do j = 1, m
S1:     A(j) = A(j) + exp()
      enddo
    enddo
```

(c)

Figure 1:

parallel and requires no synchronization. The scheduling can be dynamically overlapped with the parallel execution of the loop iterations in order to utilize the machine more uniformly. Therefore, our new method has advantages over all the previous techniques cited above since none of them has all of these desirable properties (a comparison to previous work is contained in Section 7).

After covering some necessary background information in Section 2, we describe our methods for analyzing and scheduling partially parallel loops in Sections 4 and 5. In Section 6 we discuss some strategies for applying our techniques most effectively, and we compare our new methods to previously proposed run–time methods for parallelizing loops in Section 7. Finally, we present some experimental results in Section 8.

2 Preliminaries

In order to guarantee the semantics of a loop, the parallel execution schedule for its iterations must respect the *data dependence* relations between the statements in the loop body [26, 19, 3, 41, 44]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write), *anti* (write after read), and *output* (write after write). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory-related dependences, are caused by the reuse of memory, e.g., program variables.

If there are flow dependences between accesses in different iterations of a loop, then the semantics of the loop cannot be guaranteed unless those iterations are executed in order of iteration number because values that are computed (produced) in an iteration of the loop are used (consumed) during some later iteration. For example, the iterations of the loop in Fig. 1(a) must be executed in order of iteration number because iteration $i + 1$ needs the value that is produced in iteration i, for $1 \leq i < n$. In principle, if there are no flow dependences between iterations of a loop, then those iterations may be executed in parallel. The simplest situation occurs when there are no anti, output, or flow dependences between iterations in a loop. In this case, these iterations are independent and can be executed in parallel. If there are no flow dependences, but there are anti or output dependences between iterations of a loop, then the loop must be modified to remove all such dependences before these iterations can be executed in parallel. In some cases, even flow dependences can be removed by simple algorithm substitution, e.g., reductions. Unfortunately, not all such situations can be handled efficiently. In order to remove certain types of dependences two important and effective transformations can be applied to the loop: *privatization* and *reduction parallelization*.

Privatization creates, for each processor cooperating on the execution of the loop, private copies of the program variables that give rise to anti or output dependences (see, e.g., [7, 22, 23, 39, 40]). The loop shown in Fig. 1(b), is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S2` of iteration i and statement `S1` of iteration $i + 1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`. In this paper, the following criterion is used to determine whether a variable may be privatized.

Privatization Criterion: Let A be a shared array (or array section) that is referenced in a loop L. A can be *privatized* if and only if every read access to an element of A is preceded by a write access to that same element of A within the same iteration of L.

In general, dependences that are generated by accesses to variables that are only used as workspace (e.g., temporary variables) *within* an iteration can be eliminated by privatizing the workspace. However, according to the above criterion, if a shared variable is initialized by reading a value that is computed outside the loop, then that variable cannot be privatized. Such variables could be privatized if a *copy–in* mechanism for the external value is provided. The *last value assignment* problem is the conceptual analog of the copy–in problem. If a privatized variable is *live* after the termination of the loop, then the privatization technique must ensure that the correct value is copied out to the original (non privatized) version of that variable. It should be noted that, based on our experience, the need for values to be copied into or out of private variables occurs infrequently in practice.

Reduction parallelization is another important technique for transforming certain types of data dependent loops for concurrent execution.

Definition: A *reduction variable* is a variable whose *value* is used in one associative and commutative operation of the form $x = x \otimes exp$, where $\otimes$ is the associative and commutative operator and x does not occur in exp or anywhere else in the loop.

Reduction variables are therefore accessed in a certain specific pattern (which leads to a characteristic data dependence graph). A simple but typical example of a reduction is statement `S1` in Fig. 1(c). The operator $\otimes$ is exemplified by the $+$ operator, the access pattern of array $A(:)$ is *read, modify, write*, and the function performed by the loop is to add a value computed in each iteration to the value stored in $A(:)$. This type of reduction is sometimes called an *update* and occurs quite frequently in programs. There are two tasks required for reduction parallelization: *recognizing the reduction variable*, and *parallelizing the reduction operation*. (In contrast, privatization needs only to recognize privatizable variables by performing data dependence analysis, i.e., it is contingent only on the access pattern and not on the operations.) Since parallel methods are known for performing reductions operations (see, e.g., [12, 17, 18, 20, 44]), the difficulty encountered by compilers arises from recognizing the reduction statements. So far this problem has been handled at compile–time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement [44].

3 Analyzing Partially Parallel Loops at Run–Time

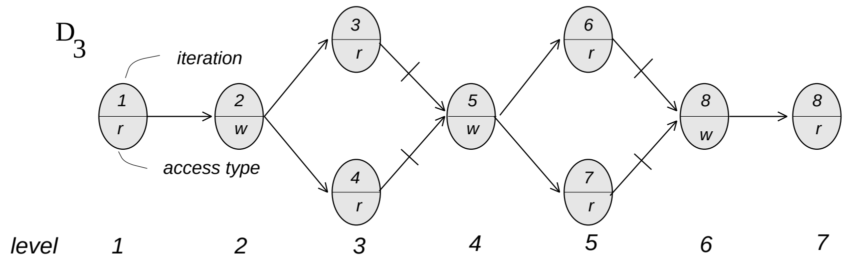
Given a `do` loop whose access pattern cannot be statically analyzed, compilers have traditionally generated sequential code. Although this pessimistic strategy is safe and simple, as mentioned in Section 1, it essentially precludes the automatic parallelization of entire classes of programs, e.g., those with irregular domains and/or dynamically changing interactions. Since compile–time data dependence analysis techniques cannot be used on such programs, methods of performing the analysis at run–time are required. During the past few years, several techniques have been developed for the run–time analysis and scheduling of loops with cross-iteration dependences [5, 9, 16, 21, 24, 28, 33, 34, 35, 42, 43]. However, for various reasons, such techniques have not achieved wide–spread use in current parallelizing compilers.

In the following we describe a new run–time scheme for constructing a parallel execution schedule for the iterations of a loop. The general structure of our method is similar to the above cited run–time techniques: given the original, or *source* loop, the compiler constructs *inspector* code that analyzes, at run–time, the cross-iteration dependences in the loop, *scheduler* code that schedules the loop iterations using the dependence information extracted by the inspector, and *executor* code that executes the loop iterations according to the schedule determined by the scheduler. In the previous techniques, the scheduler and the executor are tightly coupled codes which are collectively referred to as the executor, and the inspector and the scheduler/executor codes are usually decoupled [35]. Although for efficiency purposes our methods can also interleave the scheduler and the executor, we treat them separately since scheduling and execution are distinct tasks that can be performed independently.

First, in Section 4, we describe a new inspector scheme that in many cases should prove superior to previously proposed schemes. Next, in Section 5, we describe a scheduler that can use the dependence information found by the inspector to construct an optimal parallel execution schedule for the loop iterations; in addition, we mention how the scheduler might be interleaved with the executor to more efficiently utilize the machine. After describing the basic components of our methods, in Section 6 we discuss some strategies for applying them most effectively. Finally, we compare our new methods to other run–time parallelization schemes in Section 7.

4 The Inspector

In this section we describe a new inspector scheme that processes the memory references in a loop and constructs a data structure which the executor can use to efficiently assign iterations to wavefronts. In addition, our inspector can implement at run–time two important transformations: (element–wise) array privatization and reduction parallelization (see Section 2). The ability to identify privatizable and reduction variables is very powerful since it eliminates the data



(c)

```
do i = 1,8
  A(W(i)) = ...
    ... = A(R(i))
  work(i)
enddo
W(1:8) = [1 3 2 4 3 5 6 3]
R(1:8) = [3 7 3 3 8 3 3 3]
```

(a)

R_3	1	2	3	4	5	6	7	8	9
iter	1	2	3	4	5	6	7	8	8
type	r	w	r	r	w	r	r	w	r
level	1	2	3	3	4	5	5	6	7

(b)

H_3	1	2	3	4	5	6	7
index in R_3	1	2	3	5	6	8	9

(d)

Figure 2: A (a) source loop, (b) the array R_3 for $A[3]$, (c) its dependence graph D_3, and (d) its hierarchy vector H_3.

dependences involving these variables. In particular, it increases the available parallelism in the loop and also reduces the work required of the scheduler since it need not consider dependences involving such variables when it constructs the parallel execution schedule for the loop iterations.

The basic strategy of our method is for the inspector to preprocess the memory references and determine the data dependences for each *memory location* accessed. Later, the scheduler will use this memory-location dependence information to determine the data dependences between the *iterations*. We describe the method as applied to a shared array A that is accessed through subscript arrays that could not be analyzed at compile-time (see Figure 2(a)). For simplicity, we first consider only the problem of identifying the cross–iteration dependences for each array element (memory location). After describing this inspector, we then discuss how the dependence information it discovers can be used to identify the array elements that are read–only, privatizable, or reduction variables. The inspector has two main tasks.

1. For each array element $A[x]$, the inspector collects all the references to it into an array (or list) R_x and stores them in order of iteration number. For each reference it stores the associated iteration number and access type (i.e., read or write) (see Figure 2(b)).

2. For each array element $A[x]$, the inspector determines the data dependences between all its references and stores them in a data structure H_x for later use by the scheduler.

In Section 4.1, we discuss how the references to each array element can be collected and stored in the array (or list) R_x. Thus, assuming that R_x is available, we now describe how the inspector determines the dependences among the references to $A[x]$ and computes the data structure H_x.

The relations between the references to $A[x]$ can be organized (conceptually) into an array element dependence graph D_x. If adjacent references in R_x have different access types, then a flow or anti dependence exists, and if they are both writes, then an output dependence is signaled. These dependences are reflected by parent-child relationships in D_x. If adjacent references are both reads, then there is no dependence between the elements, but they may have a common parent (child) in D_x: the last write preceding (first write following) them in R_x. For example, the dependence graph D_3 for $A[3]$ is shown in Figure 2(c).

Our goal is to encode the predecessor/successor information of the (conceptual) dependence graph D_x in a *hierarchy vector* H_x so that the scheduler can easily look-up the dependence information for the references to $A[x]$. First, we add a *level* field to the records in R_x, and store in it the reference's level in the dependence graph D_x (see Figure 2(b)). Then, for each level, we store in H_x the index (pointer to location) in R_x of the *first* reference at that level. Specifically, H_x is an array and $H_x[i]$ contains the index in R_x of the first reference at level i, i.e., H_x will serve as a look–up table for the first reference in R_x at any level (see Figure 2(d)). Note that this implies that H_x records the position in R_x of every write access and of the first read access in any run of reads.

We now give an example of how the hierarchy vector serves as a look-up table for the predecessors and successors of all the accesses. Consider the read access to $A[3]$ in the sixth iteration, which appears as the sixth entry in R_3. Its

```
do i = 1,8
  A(W(i)) =  ...
     ...  =  A(R(i))
  work(i)
enddo
W(1:8) = [1,3,2,4,3,5,6,3]
R(1:8) = [3,7,3,3,8,3,3,3]
```

(a)

```
doall p = 1,nproc
  private int j
  do j=start(p,niter),end(p,niter)
    markwrite(W(j))
    markread(R(j))
  enddo
enddoall
```

(b)

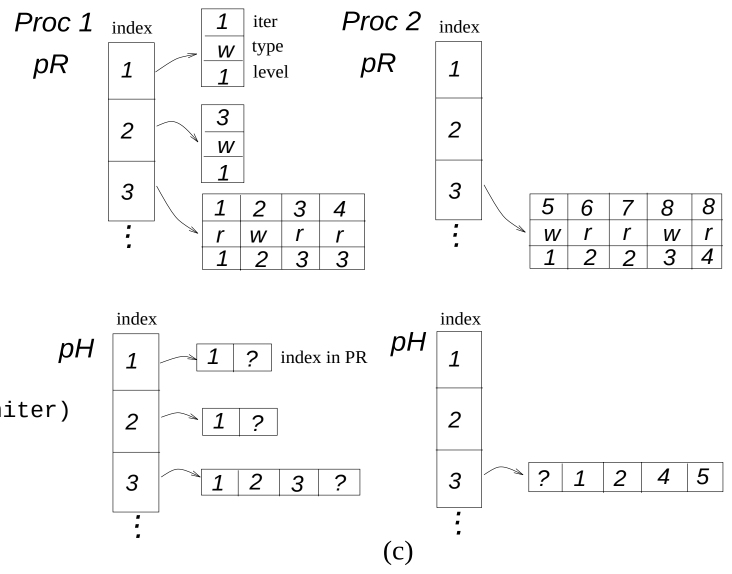


Figure 3: An example of how the private element arrays pR and hierarchy vectors pH (c) when two processors are used in the inspector `doall` loop (b) for the source `do` loop (a).

level is 5, and thus it finds its successor by looking at the $5 + 1 = 6$th element of the hierarchy vector H_3, which contains the value 8 indicating that its successor is the 8th element in R_3. Similarly, its predecessor is found by looking in the $5 - 1 = 4$th element of H_3, which indicates that its predecessor is the 5th element of R_3.

4.1 Implementing the Inspector

We now consider how to collect the accesses to each array element $A[x]$ into the arrays R_x. Regardless of the technique used to construct these arrays, to ensure the scalability of our methods we must process (*mark*) the references to the shared array A in a `doall` (see Figure 3(a) and (b)). The computation performed in the `marking` operations will depend upon the technique used to construct the arrays R_x. In any case, note that since we are interested in cross–iteration data dependences we need only record at most one read and write access in R_x for any particular iteration, i.e., subsequent reads or writes to $A[x]$ in the same iteration can be ignored.

Perhaps the simplest method of constructing the element arrays R_x is to first place a record for each memory reference into an array R_A, and then sort these records lexicographically by array element (first key) and iteration number (second key). After this sort, each array R_x will occupy a contiguous portion (a subarray) in the sorted array R_A. In this case the marking operations will simply record the information about the access into R_A. After the lexicographic sort, the level of each reference in D_x can be computed by a prefix sum computation.

However, since the range of the values to be sorted is known in advance (it is given by the dimension of the shared array A), a linear time *bucket* or *bin* sort can be used in place of the more general $O(n \log n)$ lexicographic sort. Moreover, if the inspector's marking phase is *chunked* (i.e., statically scheduled), then further optimization is possible. In this case, processor i will be assigned iterations $i\lceil n/p \rceil$ through $(i+1)\lceil n/p \rceil - 1$, where p is the total number of processors, n is the number of iterations in the loop, and $0 \leq i < p$. The basic idea is as follows. First, in a *private marking phase*, each processor marks the references in its assigned iterations, and constructs element arrays R_x and hierarchy vectors H_x as described above, *but only for the references in its assigned iterations*. Then, in a *cross–processor analysis phase*, the hierarchy vectors for the whole iteration space of the loop are formed using the processors' hierarchy (sub)vectors.

The private marking phase proceeds as follows. Let $A[1:s]$ be the shared array under scrutiny, and suppose each processor has a *separate* array $pR[1:s, 1:2n/p]$ in which to store the records of the references in its set of iterations. Each record contains the iteration, type of reference, and level as described above. (The second dimension of $1:2n/p$ follows since, as noted above, at most one read and write to any element need to be marked in each iteration, and each processor has n/p iterations.) Assuming a processor marks its iterations in order of increasing iteration number, it can immediately place the records for the references into its array pR in sorted order of iteration number. In addition to the array pR, each processor has a separate array $pH[1:s, 1:2n/p]$ used to store the hierarchy vectors for the references in its assigned set of iterations. Again, assuming that iterations are processed in increasing order of iteration number, the hierarchy vectors can be filled in at the same time that the references are recorded in pR (see Figure 3(c)).

In the analysis phase we need to find for each array element $A[x]$ the predecessor, if any, of the first reference recorded by each processor, i.e., we need to fill in the value in processor i's hierarchy vector for the reference that immediately precedes (in the dependence graph D_x) the first reference to $A[x]$ that was assigned to processor i. Similarly, we must find the immediate successor of the last reference to $A[x]$ that was assigned to processor i. Processor i can find the predecessors (successors) needed for its hierarchy vectors by scanning the arrays of the processors less than (larger than) i. For example, the "?" at the end of $pH[3]$ for processor 1 in Figure 3 would be filled in with a pointer to the first element in the array $pR[3]$ of processor 2. Hence, the initial and final entries in the hierarchy vectors also need to store the processor number that contains the predecessor and successor. These scans can be made more efficient by maintaining some auxiliary information, e.g., for each array element, each processor computes the total number of accesses it recorded, and the indices in pR of the first and last write to that element. In any case, we note that filling in the processors' hierarchy vectors requires a minimal amount of interprocessor communication, i.e., it requires only a "connecting" and not a full "merging" of the different hierarchy vectors.

There are several ways in which the above sketched analysis phase can be optimized. For example, in order to determine which array elements need predecessors and successors (i.e., the elements with non–empty arrays R_x), the processor needs to check each row of its array pR (row i of pR corresponds to the array R_i). This could be a costly operation if the dimension of the original array is large and the processor's assigned iterations have a sparse access pattern. However, the need to check each row in pR can be avoided by maintaining a list of the non–empty rows. This list can be constructed during the marking phase, and then traversed in the analysis phase – thereby avoiding the need to check every row. Another source of inefficiency for machines with many processors is the search for a particular predecessor (or successor) since each processor might need to look for a predecessor in all the preceding (succeeding) processors' iterations. The cost of these searches can be reduced from p to $O(\log p)$ using a standard parallel divide–and–conquer "pair–wise" merging approach [20], where p is the total number of processors.

4.2 Privatization and Reduction Recognition at Run-Time

The basic inspector described above can easily be augmented to find the array elements that are independent (i.e., accessed in only one iteration), read–only, privatizable, or reduction variables. We first consider the problem of identifying independent, read–only, and privatizable array elements. During the marking phase, a processor maintains the status of each element referenced in its assigned iterations *with respect to only these iterations*. In particular, if it finds than an element is written in any of its assigned iterations, then it is not read–only. If an element is accessed in more than one of its assigned iterations, then it is not independent. If an element was read before it was written in any of its assigned iterations, then it is not privatizable. Next, the final status of each element is determined in the cross–processor analysis phase as follows. An element is independent if and only if it was classified as independent by exactly one processor, and was not referenced on any other processor. An element is read–only if and only if it was either determined to be read–only by every processor that referenced it. Similarly, an element is privatizable if and only if it was either privatizable on every processor that accessed it. Thus, the elements can be categorized by a similar process to the one used to find the predecessors and successors when filling in the processors' hierarchy vectors. Finally, if we maintain a linked list of the non–empty rows of pR as mentioned above, then the rows corresponding to elements that were found to be independent, read–only, or privatizable are removed from the list, i.e., accesses to these elements need not be considered when constructing the parallel execution schedule for the loop iterations.

We now consider the problem of verifying that a statement is a reduction using run–time data dependence analysis. Recall, as mentioned in Section 2, that potential reduction statements are generally identified by syntactically matching the statement with the generic reduction template $x = x \otimes exp$, where x is the reduction variable, and $\otimes$ is an associative and commutative operator. The statement is validated as a reduction if it can be shown through dependence analysis that x is not referenced in exp or any where in the loop body outside the reduction statement. Sometimes the necessary dependence analysis cannot be performed at compile–time. This situation could arise if the reduction variable is an array element accessed through subscripted subscripts, and the subscript expressions are not statically analyzable. For example, although statement `S3` in the loop in Fig. 4(a) matches a reduction statement, it is still necessary to prove that the elements of array `A` referenced in `S1` and `S2` do not overlap with those accessed in statement `S3`, i.e., that: $K(i) \neq R(j)$ and $L(i) \neq R(j)$, for all $1 \leq i, j \leq n$. It turns out that this condition can be tested in the same way that read–only and privatizable array elements are identified. In particular, during the marking phase, whenever an element

```
        do i = 1, n
S1:        A(K(i)) = .......
S2:          ........... =  A(L(i))
S3:       A(R(i)) = A(R(i)) + exp()
        enddo
```

(a)

```
doall   i = 1, n
   private   integer   j
   do   j=start(p,niter),end(p,niter)
      markwrite(K(i))
      markredux(K(i))
      markread(L(i))
      markredux(L(i))
      markwrite(R(i))
   enddo
enddoall
```

(b)

Figure 4: The transformation of the `do` loop in (a) is shown in (b). The `markwrite` (`markread`) operation adds a record to the processor's array pR (if its not a duplicate), and updates the hierarchy vector pH appropriately. The `marknoredux` operation invalidates the indicated array element as a reduction variable since it is accessed outside the reduction statement `S3`.

is accessed outside the reduction statement the processor invalidates that element as a reduction variable. Again, the final status of each element is determined in the cross–processor analysis phase, i.e., an element is a reduction variable if and only if it was not invalidated as such by any processor.

This strategy can also be used when the *exp* part of the RHS of the reduction statement contains references to the array A that are different from the pattern matched LHS and cannot be statically analyzed, i.e., the elements referenced in *exp* are invalidated during the marking phase. A more complicated situation is when the loop contains several reduction statements that refer to the same array A. In this case the type of the reduction operation performed on each element must be the same throughout the loop execution, e.g., a variable cannot participate in both a multiplicative and an additive reduction since the resulting operation is not commutative and associative and is therefore not parallelizable. The solution to this problem is to also maintain the reduction type with each potential reduction variable. Whenever a reference in a reduction statement is marked, the current reduction type (e.g., summation, multiplication) is checked with with previous one. If they are not the same, the corresponding element is invalidated as a reduction variable.

4.3 Complexity of the Inspector

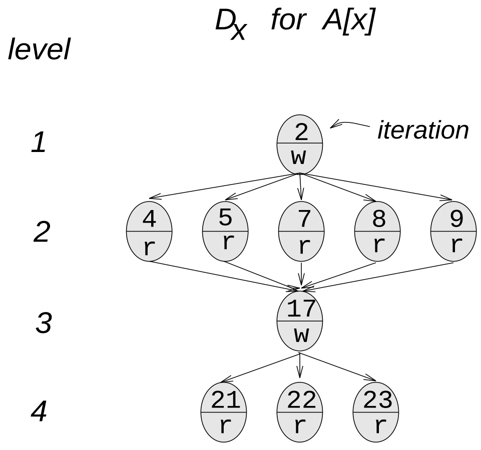
The worst case complexity of the inspector is $O(a \log p)$, where a is the maximum number of references assigned to each processor and p is the total number of processors. In particular, using the bucket sort implementation, each processor spends constant time on each of its $O(a)$ accesses in the marking phase, and the analysis phase takes time $O(a \log p)$ using a parallel divide–and–conquer pair–wise merging strategy [20]. We remark that since the cost of the analysis phase is proportional to the number of distinct elements accessed (i.e., the number of non–empty rows in the pR array) the complexity of this phase could be significantly less than $O(a \log p)$ if there are many repeated references in the loop. Also, if $a \log p > s$, then the merge among the processes can be improved to $O(s + \log p)$ time by chunking the pR arrays.

5 The Scheduler

We now consider the problem of finding an execution schedule for the iterations of the loop. We assume that the inspector described in Section 4 has been used on the loop. The scheduler derives the more restrictive iteration-wise dependence relations from the memory location dependence information found by the inspector. Formalizing this, the memory location dependences define a directed acyclic graph (dag) $D = (V, E)$ describing the cross–iteration dependences in the loop: there is a node $v_i \in V$ for each iteration i in the loop, and there is a directed edge $(v_i, v_j) \in E$ if some memory location has a dependence from iteration i to iteration j. Note that D is implicit in the reference arrays pR and their hierarchy vectors pH. A valid parallel execution schedule for a loop is a partition of the set of iterations

```
        wf(1:numiter) = 0
        done = .false.
        cpl = 1
   4    do while (done.eq..false.)
          rdy(1:numiter) = .false.
          done = .true.
   7      doall  i = 1, numaccess
             a = access(i)
   8         if (wf(a.iter) .eq. 0) then
   9           for each (b in Pred(a))
  10             if (wf(b.iter).eq.0)
                   done,rdy(a.iter) = .false.
               end for
             end if
  14      end doall
  15      doall i = 1,numiter
  16        if (rdy(i).eq..true.)
              wf(i) = cpl
  18      end doall
          cpl = cpl + 1
        end do while
```

(a)



(b)

Figure 5: A simple scheduler. In (a), `wf(i)` stores the wavefront found iteration i, the global variable `done` flags if all iterations have been scheduled, `rdy(i)` signals if iteration i is ready to be executed, lower case letters (`a,b`) are used for references to memory locations, `a.iter` is the iteration which contains reference `a`, and `Pred(a)` is the set of immediate predecessors of `a` in the memory location dependence graphs. The dependence graph for one of the memory locations accessed in the loop is shown in (b).

into ordered subsets called *wavefronts*, so that all dependences go from an iteration in a lower numbered wavefront to an iteration in a higher numbered wavefront. We say that a valid parallel execution schedule is *optimal* if it has a minimum number of wavefronts, i.e., is has as many wavefronts as the longest path (the *critical path*) in the dag.

We remark that the schedulers described below can be used to construct the full iteration schedule in advance (which is how we describe them for simplicity), or alternatively, they can be interleaved with the executor, i.e., the iterations could be executed as they are found to be ready.

5.1 A simple scheduler

A simple scheduler that finds an optimal schedule is sketched in Figure 5(a). In the figure, an array `wf(i)` stores the wavefront found for iteration i, the global variable `done` flags if all iterations have been scheduled, `rdy(i)` signals if iteration i is ready to be executed, lower case letters (`a,b`) are used for references to array elements, `a.iter` is the iteration which contains reference `a`, and `Pred(a)` is the set of immediate predecessors of `a` in the array element dependence graphs. The scheduling is performed in cpl phases (line 4) so that in phase i the iterations belonging to ith wavefront are identified. In each phase, all the references recorded in the pR arrays are processed (lines 7–13), and the predecessors of all references whose iterations have not been scheduled (line 8) are examined. An iteration is found *not ready* if the iterations of any of its reference's predecessors were not assigned to previous wavefronts (line 10). After all the references are processed, all the iterations are examined (lines 14–17) to see which can be added to the current wavefront: an iteration i is ready (line 15) if none of its references set `rdy(i)` to false. Advantages of this scheduler are that it is conceptually very simple and quite easy to implement.

Optimizing the simple scheduler. There are some sources of inefficiency in this scheduler. First, since a write access could potentially have many "parent" read accesses it could prove expensive to require such a write to check all of its "parents" (line 9). Fortunately, this problem is easily circumvented by requiring an unscheduled read access to inform its successor's iteration (the successor, if any, is a write to the same address) that it is *not* ready. Then, a write access only needs to check its predecessor if the (single) predecessor is also a write.

Another source of inefficiency arises from the fact that each inner `doall` (lines 7–13) requires time $O(n_a/p)$ to identify unscheduled iterations (line 8), where n_a is the total number of accesses to the shared array and p is the number of processors. Thus, the scheduler takes time $O((n_a/p)cpl)$, where cpl is the length of the critical path. Thus, if $p = O(cpl)$, then it cannot be expected to offer any speedup over sequential execution, and even worse, it could yield slowdowns for longer critical paths. However, note that in any single iteration of the scheduler, the only iterations that could potentially be added to the next wavefront must have all their accesses at the lowest unscheduled level in their respective element–wise dependence graphs. For example, consider the dependence graph shown in Figure 5(b). If iteration 2 (level 1) has not been scheduled yet, then none of the iterations with accesses in higher levels could be added to the current wavefront.

Thus, in each of the cpl iterations of the outer `do while` loop, we would like to examine only those references that are in the topmost unscheduled level of their respective dependence graph. First note that we can easily identify the accesses on each level of the array element dependence graphs since references are stored in increasing level order in the pR arrays and the pH arrays contain pointers the first access at each level. However, to process only the accesses on the lowest unscheduled level it is useful to have a count of the total number of (recorded) accesses in each iteration. This information can easily be extracted in the marking phase and stored in an array indexed by iteration number. Then, in the scheduler, a count of the number of ready accesses for each iteration can be computed on a per processor basis in the first `doall` (lines 7–13). In the second `doall` (lines 14–17), the cross-processor sum of the ready access counts for each unscheduled iteration is compared to its total access count, and if they are equal the iteration is added to the current wavefront.

In summary, we would expect this optimized version to outperform the original scheduler if there are multiple levels in them array element dependence graphs, i.e., because it *only examines the accesses at the lowest unscheduled level in any iteration* of the outer `do while`. However, note that if there are not many repeated write accesses (and thus few levels), then it is possible that this version could in fact prove inferior to the original (due to the cross–processors summation of the counts). Therefore, the determination of which version to use should be made using knowledge gained about the access pattern by the inspector. These issues are discussed in more detail in Section 6.

Overlapping scheduling and execution. As mentioned above, the scheduler can construct all the wavefronts in advance or it can be interleaved with the executor so that wavefronts are executed as they are found. A third alternative is to overlap the computation of the wavefronts with the execution of the loop. First, all the processors compute the first wavefront. Then, some processors are assigned to execute the iterations in that wavefront, and the rest of the processors compute the next wavefront. The strategy is carried out repeatedly until all wavefronts are computed. The number of processors assigned to each task would depend upon the amount of work contained in the wavefront. Thus, this approach "fills out" the wavefronts that cannot employ all the the processors, i.e., in effect we dynamically merge the parallelism profiles of the wavefront computation and the loop execution to more fully utilize the machine.

Remark: In this paper we are mainly concerned with constructing a parallel execution schedule for the *iterations* of the loop. However, we would like to note that the array element dependence information extracted by the inspector could also be used for producing schedules that overlap iterations or for creating multiple threads of execution.

6 Strategies for Applying Run–Time Parallelization

In this section we outline the basic strategy of using the methods in a real application environment.

At Compile–Time.

1. *A cost/performance analysis is performed to evaluate whether a speedup can be obtained by these methods (which is not always the case).*

2. *If the compiler decides to perform run–time parallelization, then an inspector for the marking phase is extracted from the source loop and any other code needed for the methods is generated.*

Cost/Performance Analysis. The cost/performance analysis is primarily concerned with evaluating the amount of available parallelism in the loop. Since the data dependence relations between the loop iterations cannot be analyzed statically, an estimate of the available parallelism in the loop can only be made at compile–time using meaningful statistics from previous runs of the program. If the loop is instantiated several times in the same program, then an estimate on the available parallelism in a future instantiation could be made at run–time using statistics from previous invocations of the loop within the same run. For every given (estimated) amount of parallelism, the potential speedup is a function of the ratio between the work of the loop body and the the number of accesses that are shadowed using our methods. The smaller this ratio, the more difficult it will be to obtain a speedup, with the worst case being what we call a "kernel," i.e., a loop that performs only data movement and no computation. Therefore, in order to obtain a speedup, a substantial amount of parallelism, and sufficient processors to exploit it, are needed.

Instrumentation and Code Generation. For the marking phase, the compiler needs to extract a *marking loop*, i.e, a *parallel loop* that traverses the access pattern of the source loop *without side effects* (without modifying the original data). It is imperative that the marking loop be parallel, for otherwise it defeats the purpose of run–time parallelization [21, 33]. (Below, we mention some special circumstances in which speedups might still be obtained using a sequential marking loop.) A parallel marking loop can be obtained if the source loop can be distributed into a loop computing the addresses of the array under test and another loop which uses those addresses (i.e., when the address computation and data computation are not contained in the same strongly connected component of the dependence graph). Unfortunately, in some cases such a marking loop does not exist. In particular, when the data computation in the loop affects future address computations in the loop. After extracting a marking loop, if possible, the compiler augments it with the code for the marking operations, and generates the code for the analysis phase, and for the scheduling and execution of the loop iterations. If a marking loop cannot be extracted, then the compiler must choose between sequential execution and a speculative parallel execution [31].

At Run–Time.

1. *At run–time (and possibly also at compile–time) an evaluation of the storage requirements of the methods is performed.* If these requirements are prohibitive for the full iteration space of the loop, then the marking loop can be strip–mined and the method (i.e, marking, analysis and scheduling) can be applied to each strip. Even in the case of strip–mining, an optimal schedule can be obtained since the scheduling method can easily be modified to assign iterations in each strip to a single wavefront structure.

2. *The marking phase is executed.*

3. *Using information gathered during the marking phase, the compiler decides whether to continue with run–time parallelization.* A lower bound on the length of the critcal path is the maximum level (across processors) assigned to any individual array element. If this lower bound is too high, then parallelization should be abandoned and the source loop should be executed sequentially since speedups are unlikely.

4. *The analysis phase is executed.* Recall that the analysis phase identifies all elements that are independent, read–only, privatizable, or reduction variables, and that accesses to these elements are removed from consideration by the scheduler. If all elements fall into one of these categories, then the loop can be executed as a `doall` and the scheduling step is omitted.

5. *Execute an appropriate scheduler (overlapping it with ready iterations of the source loop).* The optimized simple scheduler should prove superior to the original version unless the element–wise dependence graphs have large average degree (see Section 5). Since the optimal parallel schedule may be imbalanced (the number of iterations in a wavefront can vary significantly between wavefronts), it is desirable to interleave the scheduler and the executor, i.e., overlap the scheduler's wavefront computations with the actual execution of the ready iterations. This can either be achieved with a dynamic partition of the processors among these two tasks (see Section 5) or with a dynamic ready queue [25, 29].

Schedule reuse and decoupling the inspector/scheduler and the executor. Thus far, we have assumed that our methods must be used *each* time a loop is executed in order to determine a parallel execution schedule for the loop. However, if the loop is executed again, with the same data access pattern, the first schedule can be reused amortizing the overhead of the methods over all invocations. This is a simple illustration of the *schedule reuse* technique, in which a correct execution schedule is determined once, and subsequently reused if all of the defining conditions remain invariant (see, e.g., Saltz *et al.* [35]). If it can be determined at compile time that the data access pattern is invariant across different executions of the same loop, then no additional computation is required. Otherwise, some additional computation must be included to check this condition, e.g., for subscripted subscripts the old and the new subscript arrays can be compared. Although a parallel marking loop is always desirable, if schedule reuse can be applied then it may still be possible to obtain speedups with a sequential marking loop since its one sequential execution will be amortized over all loop instantiations.

Method	obtains optimal schedule	contains sequential portions	requires global synchron	restricts type of loop	privatizes or finds reductions
This Paper	Yes	No	No	No	P,R
Zhu/Yew [43]	No^{1}	No	Yes^{2}	No	No
Midkiff/Padua [24]	Yes	No	Yes^{2}	No	No
Krothapalli/Sadayappan [16]	No^{3}	No	Yes^{2}	No	P
Chen/Yew/Torrellas [9]	No^{1,3}	No	Yes	No	No
Saltz/Mirchandaney [33]	No^{3}	No	Yes	Yes^{5}	No
Saltz *et al.* [35]	Yes	Yes^{4}	Yes	Yes^{5}	No
Leung/Zahorjan [21]	Yes	No	Yes	Yes^{5}	No
Polychronopoulous [28]	No	No	No	No	No
Rauchwerger/Padua [30, 31]	No^{6}	No	No	No	P,R

Table 1: A comparison of run–time parallelization techniques for `do` loops. In the table entries, P and R show that the method identities privatizable and reduction variables, respectively. The superscripts have the following meanings: 1, the method serializes all read accesses; 2, the performance of the method can degrade significantly in the presence of hotspots; 3, the scheduler/executor is a `doacross` loop (iterations are started in a wrapped manner) and busy waits are used to enforce certain data dependences; 4, the inspector loop sequentially traverses the access pattern; 5, the method is only applicable to loops without any output dependences (i.e., each memory location is written at most once); 6, the method only identifies fully parallel loops.

Another method to reduce the cost associated with these methods is to hide their overheads by executing them as soon as all the necessary data is available. If this type of decoupling is possible, then the inspector phase could be overlapped with other portions of the program—thereby more fully exploiting the processing power of the machine (of course support for MIMD execution is highly desirable in this case).

7 A Comparison with Previous Methods

In this section we compare the methods described in this paper to several other techniques that have been proposed for the run–time analysis and scheduling of `do` loops. Most of the previous work has concentrated on developing inspectors. Consequently, a wide variety of inspectors have been proposed that differ according to the types of loops on which they can be applied, the techniques they use, and the information they gather. In the following, we briefly describe some of the previous methods, placing particular emphasis on the differences from and similarities to our methods. A high level comparison of the various methods is given in Table 1.

Methods utilizing critical sections. One of the first run–time methods for scheduling partially parallel loops was proposed by Zhu and Yew [43]. It computes the wavefronts one after another using a method similar to the simple scheduler described in Section 5.1. During a phase, an iteration is added to the current wavefront if none of the data accessed in that iteration is accessed by any lower unassigned iteration; the lowest unassigned iteration to access any array element is found using atomic *compare-and-swap* synchronization primitives and a shadow version of the array. Midkiff and Padua [24] extended this method to allow concurrent reads from a memory location in multiple iterations. Due to the compare–and–swap synchronizations, this method runs the risk of a severe degradation in performance for access patterns containing *hot spots* (i.e., many accesses to the same memory location). However, when there are no hot spots and the critical path length is very small, then this method should perform well. An advantage of this method is reduced memory requirements: it uses only a shadow version of the shared array under scrutiny whereas all other methods (except [28, 30, 31]) unroll the loop and store all the accesses to the shared array.

Krothapalli and Sadayappan [16] proposed a run–time scheme for removing anti and output dependences from loops. Their scheme includes a parallel inspector that determines the number of accesses to each memory location using critical sections as in the method of Zhu and Yew (and is thus sensitive to hotspots). Using this information, for each memory location, they place all accesses to it in a dynamically allocated array and then sort them according to iteration number. Next, the inspector builds a dependence graph for each memory location (similar to our arrays R_x), dynamically allocates any additional global storage needed to remove all anti and output dependences (using renaming), and explicitly constructs the mapping between all the memory accesses in the loop and the storage, both old and new, thereby inserting an additional level of indirection into all memory accesses. The loop is executed in parallel using synchronization (full/empty bits) to enforce flow dependences. To our knowledge, this is the only other run–time privatization technique except [30, 31].

Recently, Chen, Yew, and Torrellas [9] proposed an inspector that has a private phase and a merging phase. In the private phase, the loop is chunked and each processor builds a list of all the accesses to each memory location for its assigned iterations. This is similar to the private marking phase of our inspector except that they serialize read accesses (i.e., they have a list instead of the dependence graph). Next, the lists for each memory location are linked across processors using a global Zhu/Yew algorithm [43]. Their scheduler/executor uses `doacross` parallelization [33], i.e., iterations are started in a wrapped manner and processors busy wait until their operands are ready. Although this scheme potentially has less communication overhead than [43], it is still sensitive to hot spots and there are cases (e.g., `doalls`) in which it proves inferior to [43].

Methods for loops without output dependences. The problem of analyzing and scheduling loops at run–time has been studied extensively by Saltz *et al.* [5, 33, 34, 35, 42]. Most of their work assumes that there are no output dependences in the source loop. In `doacross` parallelization [33], an inspector finds the (at most one) iteration in which each variable is written. The scheduler/executor starts iterations in a wrapped manner and processors busy wait until their operands are available. In [35], the inspector constructs wavefronts that respect the flow dependences by performing a *sequential* topological sort of the accesses in the loop, and the scheduler/executor enforces any anti dependences using old and new versions of each variable (possible since each variable in the source loop is written at most once). The topological sort can be parallelized somewhat using `doacross` parallelization. Leung and Zahorjan [21] proposed methods of parallelizing the sequential inspector of [35]. In *sectioning*, the loop is chunked and each each processor computes an optimal parallel for its chunk, and then these schedules are concatenated together, separated by synchronization barriers. In *bootstrapping*, the inspector is parallelized using sectioning. Although bootstrapping might not optimally parallelize the inspector (due to the synchronization barriers introduced for each processor), it will produce the same optimal schedule as the original sequential inspector.

Other methods. In contrast to the above methods which place iterations in the lowest possible wavefront, Polychronopolous [28] gives a method where wavefronts are maximal sets of contiguous iterations with no cross-iteration dependences. Dependences are detected using shadow versions of the variables, either sequentially, or in parallel with the aid of critical sections as in [43].

All of the above mentioned methods attempt to find a valid parallel execution schedule for the source `do` loop. Recently, we considered a related problem [30, 31]: testing at run–time whether the loop is fully parallel, i.e., whether there are any cross-iteration dependences in the loop. Our interest in fully parallel loops is motivated by the observation that they arise frequently in real programs. The test uses shadow versions of the shared variables, is fully parallel, requires no synchronization, and can be applied to any loop. If desired, it can be used speculatively (i.e, without an inspector), and can also identify privatizable and reduction variables.

8 Experimental Results

In this section we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [1]) and 14 processors (Alliant FX/2800 [2]). However, we remark that the results scale with the number of processors and the data size and thus they may be extrapolated for massively parallel processors (MPPs), the actual target of our run–time methods.

To demonstrate that the new methods can achieve speedups, we applied them to three loops contained in the PERFECT Benchmarks [4] that could not be parallelized by any compiler available to us. In addition, in order to analyze the overhead incurred by the methods, we applied them to different access patterns taken from loops in the PERFECT Benchmarks and to synthetic access patterns generated to test their behavior in various situations.

The methods were implemented in Cedar Fortran [14]. The inspector was essentially as described in Section 4. In particular, we implemented the bucket sort version using separate pR and pH data structures for each processor. To avoid checking each row in pR during the analysis phase of the inspector and in the scheduler, each processor constructed a linked list of the non-empty rows in its pR array during the marking phase. Checks for independent, read–only, and privatizable elements were implemented in the inspector (we did not yet incorporate the test for reduction variables). In the analysis phase, these elements are classified at the same time that the predecessors and successors are found for each row. One optimization that we did not yet implement was the "pair–wise" merge across processors when searching for predecessors or successors in the analysis phase (or when classifying elements as independent, read–only, or privatizable). However, this is an important optimization since, as previously noted, without it the analysis phase of the inspector may fail to scale with the number of processors. Since we implemented the optimized version of the simple scheduler described in Section 5, a count of the total number of accesses in each iteration was computed in the marking phase (no inter-processor communication is needed to determine these counts since each iteration is assigned to a single processor). For simplicity, the scheduler and the executor were completely decoupled in the implementation. In general, however, better speedups should be obtainable by interleaving these two tasks (see Section 5).

8.1 Synthetic Access Patterns

Using synthetic loops, we now study the sensitivity of the overhead of the methods to two characteristics of the source `do` loop: its *average parallelism* (the number of iterations divided by the number of wavefronts in an optimal parallel execution schedule) and its *hotspot degree* (the maximum number of repeated accesses to any array element). To simplify the generation of the synthetic workloads, we did not identify independent, read–only, or privatizable elements in the analysis phase. This should not affect our conclusions, however, since these computations can be folded into the searches for predecessors and successors (with little extra work).

Average parallelism. To isolate the affect of the average parallelism in the source loop on the overhead of the methods, we generated access patterns that were as similar as possible in all aspects except for the average parallelism. In particular, there were two accesses in every iteration (a read followed by a write), and every array element was accessed approximately twice (at some boundary conditions some elements are accessed either once or three times).

First, we would not expect the inspector execution time to be dependent on the average parallelism in the loop. In the marking phase each processor marks n_a/p accesses in its private shadow array (to isolate the effects of the average parallelism, we assume that the marking phase is balanced). The overhead of the analysis phase is primarily dependent upon the number of distinct array elements marked in its pR array (since it must find successors and predecessors for each non–empty row). Thus this overhead might vary *inversely* with the hotspot degree, but it is not necessary dependent on the average parallelism because for the same critical path length the hotspot degree can be anywhere between 2 and the number of iterations. In Figures 16 and 17 we display results from a loop with 2048 iterations run on 10 processors. The plot shows the overhead incurred for a loop with a critical path length of “Step” (the average parallelism is the number of iterations divided by the critical path length). As expected, the overhead of the inspector is invariant with the length of the critical path, and that of the scheduler grows linearly with this length.

We now consider how the speedup of the overheads relates to the average parallelism. Since the execution time of the inspector is independent of the average parallelism, its speedup should not depend on it either. Even though the scheduling time does depend on the average parallelism, its speedup is not necessarily similarly correlated. This is because each wavefront is calculated in `doall` loops, i.e., each of iteration of the scheduler can be expected to obtain reasonable speedups, and thus the overhead of the scheduler as a whole can be expected to obtain good speedups as well. In Figures 18 and 19 we show the speedup obtained for the inspector and executor, respectively, on a loop with 2048 iterations and three different values of average parallelism. In both cases the similar speedups are obtained for the sequential loop (average parallelism 1) and the loop that is almost fully parallel (average parallelism 1024). In Figures 20 and 21 we show analogous results on a loop with 1024 iterations. Recall that in our implementation we did not use a “pair–wise” merge among the processors, i.e., in our implementation each processor checks all $p-1$ other processors for predecessors and successors whereas in the pair–wise merge only $O(\log p)$ operations would be needed. This fact is most likely the cause of the slightly diminished slope of the speedup curve after about 10 processors for the overhead of the inspector.

Hotspots. To isolate the effect of the hotspot degree in the source loop on the overhead of the methods, we generated access patterns that were as similar as possible in all aspects except for the hotspot degree. In particular, all loops had 2048 iterations, two accesses in each iteration, and an average parallelism of 51 (a critical path length of 40). Also, a loop with hotspot value h contained h references to each of n/h array elements, where $n = 2048$ is the number of iterations in the loop. In principle, we would not expect our methods to be negatively affected by the hot spot degree. In fact, a larger the hotspot degree implies fewer non-empty rows in the pR array, and thus we might see improved results in the analysis and scheduling phase since few rows would need to be accessed. The results in Figure 15 show that in fact the total overhead (inspector + scheduler) is nearly the same for all hotspot degrees.

8.2 Real Access Patterns

Now we would like to look at access patterns arising in real applications to demonstrate the diversity of partially parallel access patterns and their associated parallelism profiles. By applying the new methods to such access patterns, we can reconfirm the conclusions reached above using synthetic reference patterns. For this purpose we have chosen a loop out of MA28, a blocked sparse UN-symmetric linear solver [10]. Loop MA30cd/DO_120 performs the forward–backward substitution in the final phase of the blocked sparse linear system solver (MA28). We selected this loop because it can generate many diverse access patterns when using the Harwell-Boeing matrices as input. Unfortunately, however, the loop itself is not a good candidate for parallelization since it performs very little work and is highly imbalanced due to the blocked nature of the algorithm employed by MA28.

We will limit our discussion below to two input sets: gemat12, which generates 4929 iterations, and bp_1600, which generates 822 iterations. After extracting and precomputing the linear recurrences from the source loop (based on the methods described in [32]), we generated a fully parallel inspector and applied our methods to compute an optimal parallel execution schedule for the loop.

From the data obtained we constructed the parallelism profiles depicted in Figures 6 and 7. These profiles depict the size of the wavefronts of the optimal parallel execution schedule. As we can see from the figures, the same loop can have vastly different dependence relations between its iterations. These figures clearly point out both the need for run–time analysis techniques and for dynamic and adaptive scheduling schemes capable of overlapping scheduling and execution. Figure 6 shows that most of the iterations of the loop can be executed in the initial wavefronts (the critical path length is 114). This suggests that in this case it would be more beneficial to interleave the parallel wavefront computation with the execution of previous wavefronts than it would be to overlap them, so that parallelization (and its associated overhead) can be abandoned when the sequential tail of the profile is reached. Although in Figure 7 most of the iterations are also executed in the intial wavefronts, in this case it appears that some benefit could be gained by overlapping, i.e., we can take advantage of the "pauses" in parallelism to compute future (hopefully larger) wavefronts. The histograms in Figures 8 and 9 underscore the need for scheduling and execution strategies that can be dynamically adapting depending upon the type of parallelism encounted to more fully utilize the machine.

Despite the differences in the parallelism profiles Figures 10 and 11 show that the overhead of the run–time methods described in this paper achieve similar performance. The reason that larger speedups were not obtained is that the loop is heavily imbalanced due to the blocked nature of the algorithm used in MA28.

8.2.1 Parallelizing Benchmark Loops

We applied the methods to three loops contained in the PERFECT Benchmarks [4] that could not be parallelized by any compiler available to us. In the analysis phase of the inspector it was found that one of the loops was fully parallel, and that the other two could be transformed into `doalls` by privatizing the shared array under test. We show in Figures 12 through 14 the speedup measured for each loop as a function of the number of processors used. As a reference, we give the ideal speedup, which was measured using an optimally parallelized (by hand) version of the loop. These graphs show that the speedup scales with the number of processors and is a significant percentage of the ideal speedup. Below, we discuss each loop in more detail.

We remark here that these loops could also be identified by the LRPD test [30, 31], a run–time test for identifying fully parallel loops, or loops that can be transformed into `doalls` using privatization and reduction parallelization. An advantage of the LRPD test is that it has a smaller overhead than the methods we present here. The disadvantage of the LRPD test is that if the loop cannot be transformed into a `doall`, then the overhead of applying the method is added to cost of the sequential execution, i.e., a slight "slowdown" may be incurred. Ideally, in order to exploit the relative advantages of the two methods, one would like to apply them both simultaneously.

BDNA–ACTFOR–Loop 240. This loop selects certain elements from a large array, and processes the selected elements later in the loop. The shared array is accessed through a subscript array that is computed inside the loop (and thus cannot be analyzed at compile–time). Although there are repeated accesses in this loop, it is determined in the analysis phase of the inspector that the entire shared array is privatizable, i.e., that the loop can be transformed into a `doall` by privatizing the array. As shown in Figure 12, the obtained speedup scales with the number of processors and is a significant percentage of the ideal speedup.

MDG–INTERF–Loop 1000. This loop calculates inter-molecular interaction forces. In the marking loop, to avoid introducing false dependences we computed the branch predicates that guard accesses to the shared array under scrutiny. As with the array in the loop from BDNA, it is found in analysis phase of the inspector that the entire shared array is privatizable. The speedup obtained scales with the number of processors and is a significant fraction of the ideal (see Figure 13).

OCEAN–FTRVMT–Loop 109. This kernel–like loop is utilized in the computation of a 2–dimensional FFT and accesses a vector with run–time determined strides. During the analysis phase of the inspector it is found that all accesses in the loop are unique, i.e., it is a fully parallel loop. Since this loop is invoked 26,000 times, and accounts for 40% of the sequential execution time of the program, it is an excellent candidate for *schedule reuse* (see Section 6). The access pattern for each instantiation of the loop is determined by a set of five scalars. In order to apply schedule reuse, we checked whether the current set of scalars matched a previously analyzed set. If not, then we applied the parallelization techniques, and if they did match then we simply executed the loop as a `doall`. As can be seen in Figure 14, with schedule reuse we obtain scalable speedups that are comparable to the ideal speedup.

9 Conclusion

Parallelizing statically intractable loops at run–time is an important task since automatic, compile–time parallelization had stopped with regular, well–behaved, statically defined programs—which represent only a fraction of all applications. We believe that aggressive, dynamic techniques can break this barrier and extract much of the available parallelism from even the most complex programs. Motivated by these concerns, we proposed new run–time inspector and scheduler methods for parallelizing partially parallel loops. The inspector is fully parallel, uses no synchronization, and can be applied to any loop (from which an inspector can be extracted). In addition, it can implement at run–time the two most effective transformations for increasing the amount of parallelism in a loop: array privatization (element–wise) and reduction parallelization. The scheduler/executer constructs an optimal parallel execution schedule for the iterations of the loop. Although the wavefronts of the schedule are constructed in sequence, the computation of each wavefront is fully parallel and requires no synchronization. These new methods improve on all previously proposed techniques since none of them simultaneously has all these features (Section 7). The experimental results show that the proposed methods are capable of obtaining speedups. In particular, since their overhead scales with the number of processors, given sufficient processors it will become a very small fraction of the sequential execution time of the loop. Therefore, we believe that the significance of these methods will increase with the advent of massively parallel processors (MPPs) in which the penalty of not extracting the available parallelism in a loop could be a massive performance degradation.

Although these new methods illustrate the potential benefits of run–time parallelization, there is still much work left to be done. For example, there are many potential scheduling strategies that need to be studied such as decoupling the inspector/scheduler and the executor in order to hide the overheads, dynamically overlapping scheduling and execution, or, constructing parallel threads of execution (as opposed to wavefronts). In any case, further investigation is needed to determine the relative performance of the various strategies in different circumstances. Another important task is to devise effective, automatable strategies for determining when and how to use run–time parallelization. Since speedups obtainable from run–time parallelization are upper bounded by the inherent parallelism of the loop, the compiler needs to estimate obtainable parallelism. Such estimates can be produced only through collection and interpretation of valid statistics from programs in different application domains. The new methods provide a useful tool for such studies since they determine the dependence graph and parallelism profile of the loop. It should be noted that run–time overhead could be significantly reduced through architectural support.

We view the methods described in this paper as a building block in an evolving framework of run–time parallelization as a complement to the existing techniques [30, 31, 32].

Acknowledgment

We would like to thank Paul Petersen for his useful advice, and William Blume and Brett Marsolf for identifying and clarifying applications for our experiments. We are also very grateful to Richard Cole for his suggestions regarding sorting algorithms.

References

[1] Alliant Computer Systems Corporation. *FX/Series Architecture Manual*, 1986.

[2] Alliant Computers Systems Corporation. *Alliant FX/2800 Series System Description*, 1991.

[3] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA., 1988.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[5] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. Interim Report 90-13, ICASE, 1990.

[6] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks^{TM} Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[7] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[8] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Comm. ACM*, 37(4):31–41, April 1994.

[9] D. K. Chen, P. C. Yew, and J. Torrellas. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of Supercomputing 1994*, pages 518–527, Nov. 1994.

[10] I. S. Duff. Ma28– a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE R8730, HMSO, London, 1977.

[11] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 17–25, August 1991.

[12] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[13] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh. CEDAR – A Large Scale Multiprocessor. *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524–529, Aug., 1983.

[14] M. Guzzi, D. Padua, J. Hoeflinger, and D. Lawrie. Cedar fortran and other vector and parallel fortran dialects. *J. Supercomput.*, 4(1):37–62, March 1990.

[15] A.K. Jones and P. Schwartz. Using multiprocessor systems - a status report. In *ACM Computing Surveys 12(2)*, pages 121–166, 1980.

[16] V. Krothapalli and P. Sadayappan. An approach to synchronization of parallel computing. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 573–581, June 1988.

[17] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[18] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 869–876, August 1986.

[19] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[20] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[21] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pages 83–91, May 1993.

[22] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 313–322, 1992.

[23] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proceedings 5th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[24] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. Comput.*, C-36(12):1485–1495, 1987.

[25] Jose E. Moreira and Constantine D. Polychronopoulos. Autoscheduling in a Distributed Shared-Memory Environment . Technical Report 1373, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, June 1994.

[26] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, December 1986.

[27] J.-K. Peir and D. D. Gajski. Data flow execution of fortran loops. In *Proc. First Int'l. Conf. on Supercomputing Systems (SCS 85)*, pages 129–139, Dec. 16-20, 1985.

[28] C. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and Their Imp act on Architecture Design. *IEEE Trans. Comput.*, C-37(8):991–1004, August 1988.

[29] Constantine Polychronopoulos, Nawaf Bitar, and Steve Kleiman. nanoThreads: A User-Level Threads Architecture. *Proc. of the 1993 Int'l. Conf. on Parallel Computing Technologies, Moscow, Russia*, September 1993.

[30] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 33–43, July 1994.

[31] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. Technical Report 1390, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., November 1994.

[32] Lawrence Rauchwerger and David A. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. Technical Report 1349, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res.and Dev., May 1994.

[33] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 International Conference on Parallel Processing*, pages 174–178. CRC Press, Inc., 1991. Vol. II - Software.

[34] J. Saltz, R. Mirchandaney, and K. Crowley. The doconsider loop. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 29–40, June 1989.

[35] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.

[36] B. J. Smith. A pipelined, shared resource mimd computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, 1987.

[37] J. E. Thornton. *Design of a Computer:The Control Data 6600*. Scott, Foresman, Glenview, Illinois, 1971.

[38] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[39] P. Tu and D. Padua. Array privatization for shared and distributed memory machines. In *Proceedings 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, September 1992.

[40] P. Tu and D. Padua. Automatic array privatization. In *Proceedings 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[41] M. Wolfe. *Optimizing Compilers for Supercomputers.* The MIT Press, Boston, MA, 1989.

[42] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 International Conference on Parallel Processing*, pages 26–30. CRC Press, Inc., 1991. Vol. II - Software.

[43] C. Zhu and P. C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13(6):726–739, June 1987.

[44] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

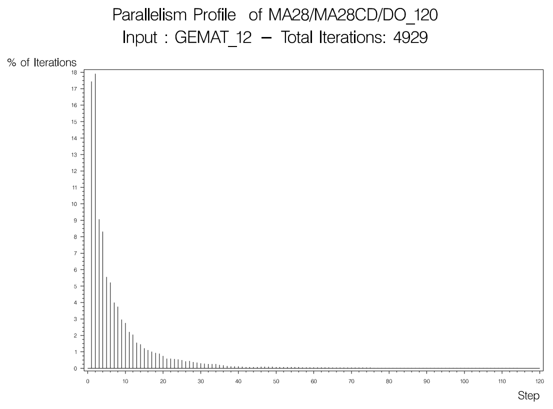


Figure 6:

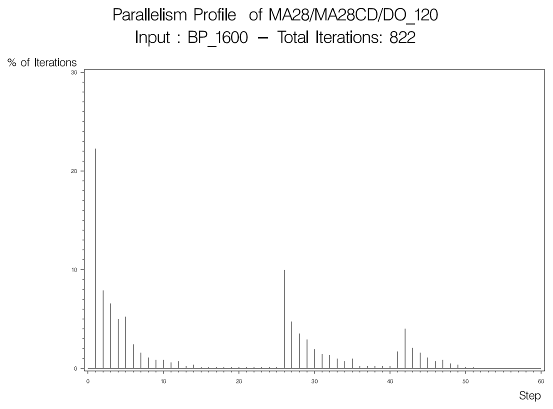


Figure 7:

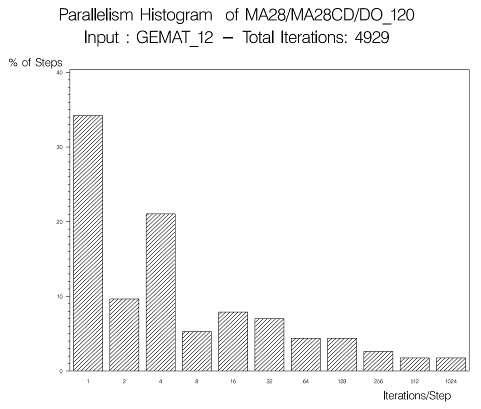


Figure 8:

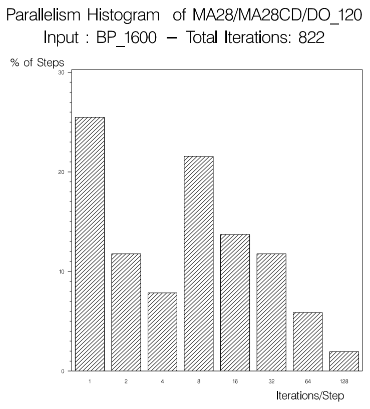


Figure 9:

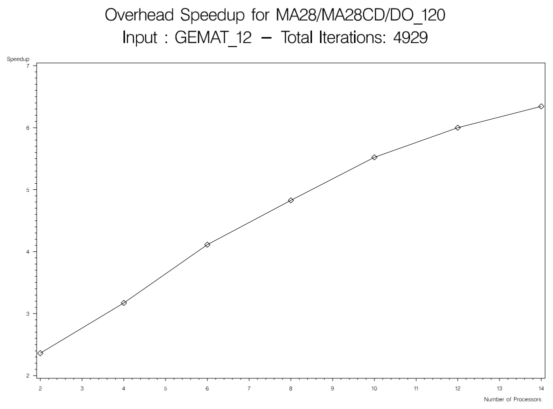


Figure 10:

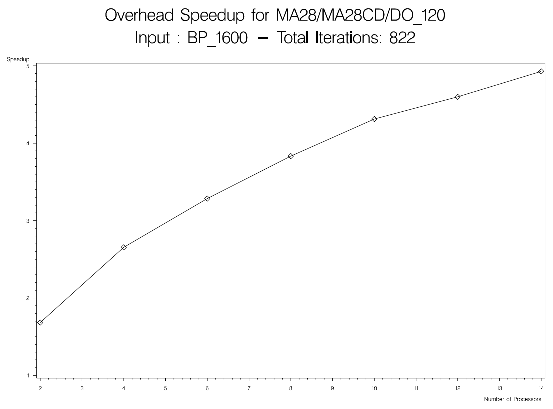


Figure 11:

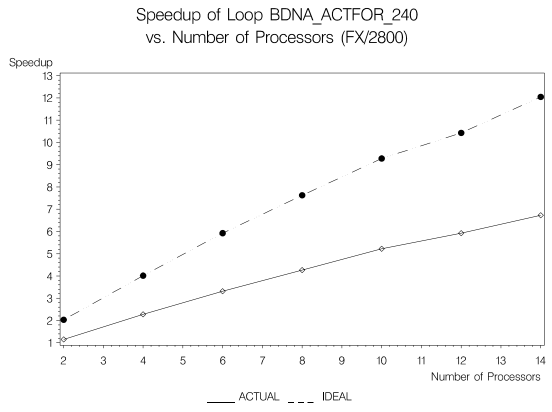


Figure 12:

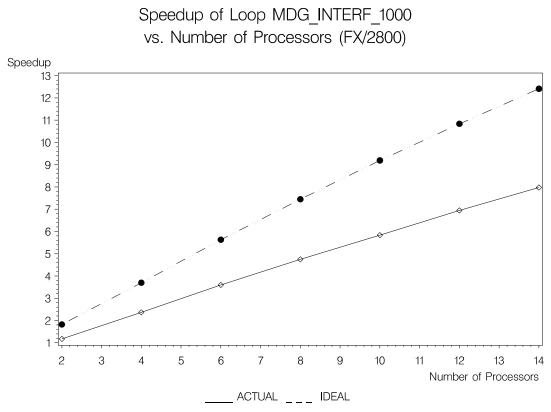


Figure 13:

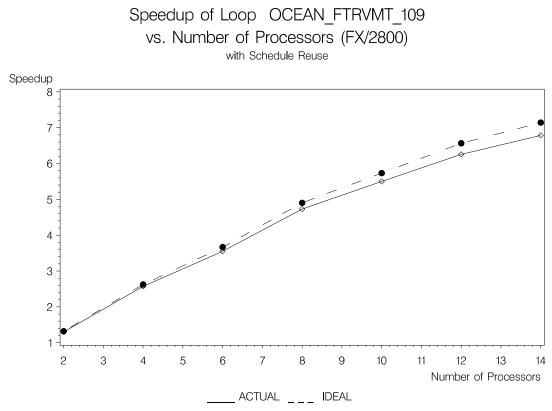


Figure 14:

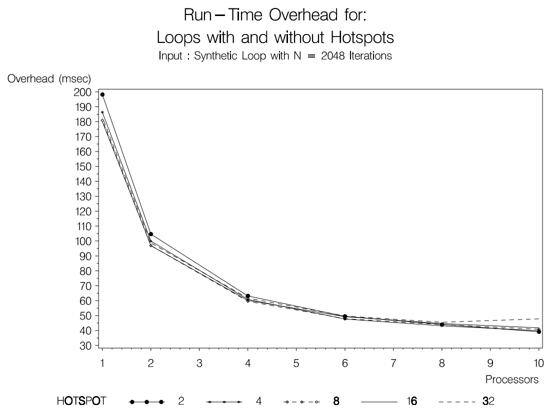


Figure 15:

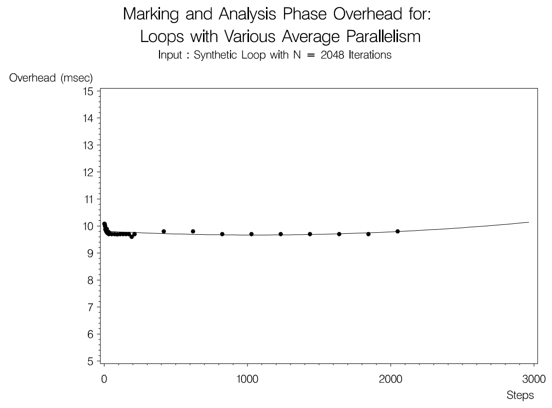


Figure 16:

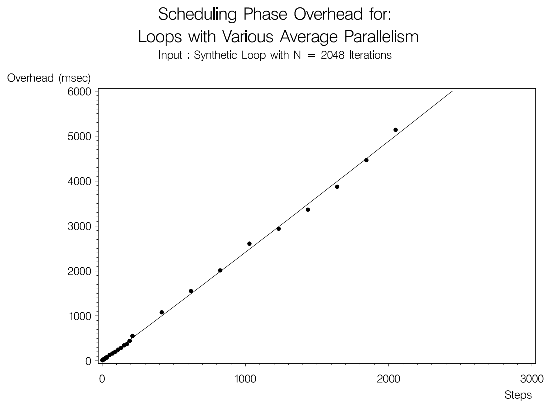


Figure 17:

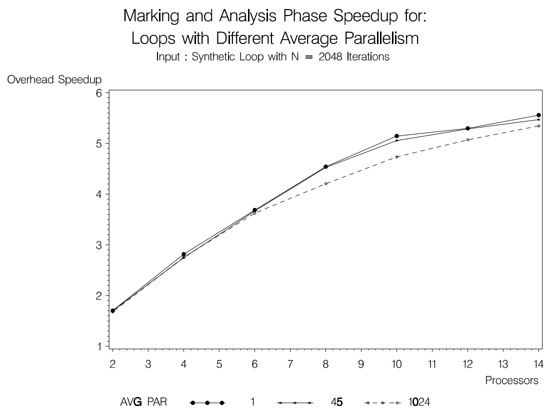


Figure 18:

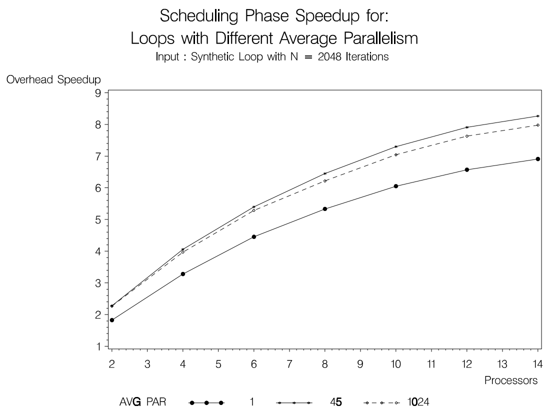


Figure 19:

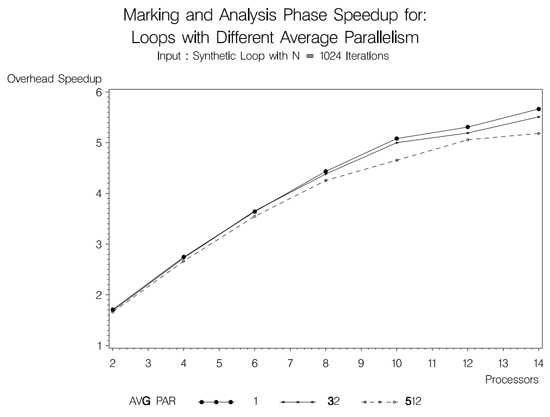


Figure 20:

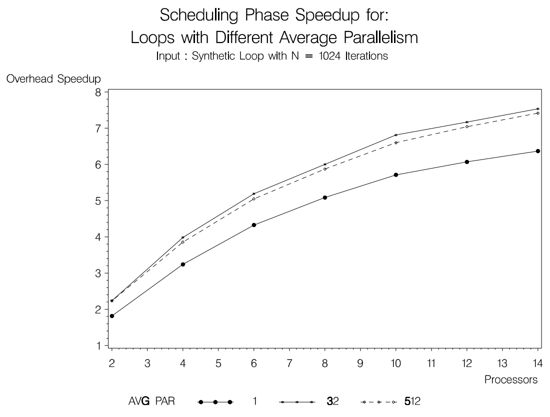


Figure 21: