Run–Time Methods for Parallelizing Partially Parallel Loops§

Lawrence Rauchwerger† University of Illinois
Nancy M. Amato‡ Texas A&M University
David A. Padua† University of Illinois

Abstract

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. 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). We also describe a new scheme for constructing an optimal parallel execution schedule for the iterations of the loop.

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 [22, 32], 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. Examples include 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].

§Due to space limitations, this paper is an extended abstract of [24].

†Center for Supercomputing Research & Development, 1308 W. Main St., Urbana, IL 61801, email: rwerger,padua@csrd.uiuc.edu. Research supported in part by Intel and NASA Graduate Fellowships, and Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

‡Department of Computer Science, Texas A&M University, College Station, TX 77843-3112, email: amato@cs.tamu.edu. Research supported in part by an AT&T Bell Laboratories Graduate Fellowship, NSF Grant CCR-9315696, and the International Computer Science Institute, Berkeley, CA.

ermission to make digital/hard copies of all or part of this material with-
ıt fee is granted provided that the copies are not made or distributed
ır profit or commercial advantage, the ACM copyright/server
otice, the title of the publication and its date appear, and notice is given
ıat copyright is by permission of the Association for Computing Machinery,
ıc. (ACM). To copy otherwise, to republish,to post on servers or to
edistribute to lists, requires specific permission and/or fee.
CS '95 Barcelona, Spain © 1995 ACM 0-89791-726-6/95/0007..$3.50

Thus, since the available parallelism in theses types of applications cannot be determined statically by present parallelizing compilers [6, 8], compile-time analysis must be complemented by new methods capable of automatically extracting parallelism at *run–time*. Run–time techniques are needed because the access pattern of some programs cannot be statically determined, either because of limitations of current analysis algorithms or because the access pattern is input data dependent. For example, most dependence analysis algorithms conservatively assume dependences when presented with non–linear or subscripted subscript expressions.

During the past few years, techniques have been developed for the run–time analysis and scheduling of loops [5, 9, 13, 17, 20, 23, 25, 26, 27, 28, 29, 30, 33, 34]. 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 [30].

Our Results. 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 and increases the available parallelism in the loop. The schedule partitions the set of iterations into subsets called *wavefronts*. 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 parallel and requires no synchronization. The scheduling can be dynamically overlapped with the parallel execution of the loop iterations to utilize the machine more uniformly. Our new method improves on the previous techniques since none of them has all of these properties (a comparison to previous work is contained in Section 4).

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 [22, 15, 3, 32, 35]. 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

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

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

Figure 1:

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. 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 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, 18, 19, 31]). The loop shown in Figure 1(a), 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.

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 operation of the form $x = x \otimes exp$, where $\otimes$ is the associative operator and x does not occur in exp or anywhere else in the loop. If the operator is not commutative then the implementation of the parallel equivalent reduction operation is more constrained.

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 Figure 1(b). 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 $A(:)$. Once reduction variables are identified, methods are known for performing the reduction operation in parallel (see, e.g., [11, 14, 16, 35]).

3 Run–Time Analysis of Loops

Given a do loop whose access pattern cannot be statically analyzed, compilers have traditionally generated sequential code. Since compile–time data dependence analysis techniques cannot be used on such programs, methods of performing the analysis at run–time are required. Several techniques have been developed for the run–time analysis and scheduling of loops with cross-iteration dependences [5, 9, 13, 17, 20, 23, 28, 29, 30, 33, 34]. 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 generates *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. 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 [30]. Although our methods can also interleave the scheduler and the executor, we treat them separately since they do tackle distinct tasks.

3.1 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 scheduler 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 dependences involving these variables. In particular, these transformations increase the available parallelism in the loop and also reduce 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 uses 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 (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 the inspector, we 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 iteration order. For each reference it stores the 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.

Below we discuss how the references to each array element can be collected and stored in the array (or list) R_x. Assuming R_x is available, we first 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

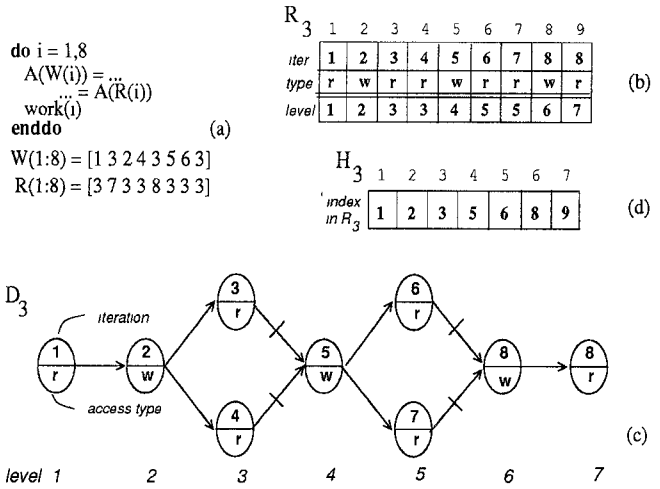


Figure 2: A (a) source loop, (b) the array R_3 for $A[3]$, (c) its dependence graph D_3, and (d) its hierarchy vector H_3.

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 6th iteration, which appears as the 6th entry in R_3. Its 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.

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 one 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

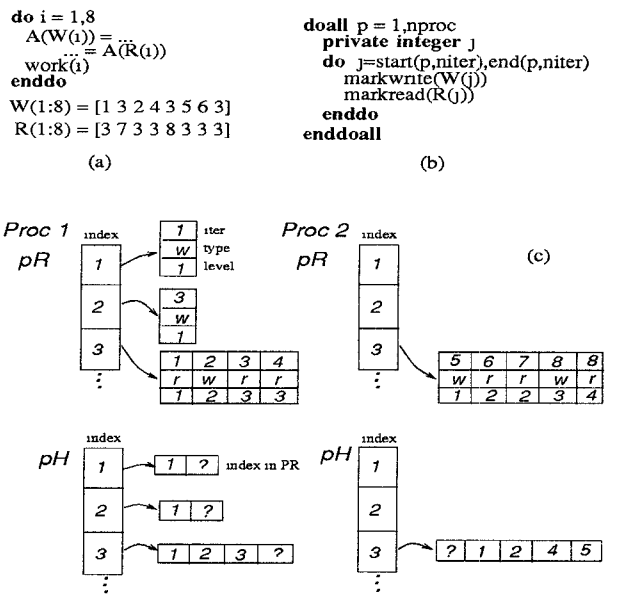


Figure 3: An example of the private element arrays pR and hierarchy vectors pH (c) when two processors are used in the inspector `doall` loop (b) for the source do loop (a).

R_x is to first place a record for each memory reference into an array R_A, and then sort the records lexicographically by array element number (first key) and iteration number (second key). After sorting, each array R_x will occupy a contiguous portion (a subarray) in the array R_A. In this case the marking operations 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 at most one read and one 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 cross-processor 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. 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 [16], where p is the total number of processors.

Privatization and Reduction Recognition. 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 determined to be read–only by every processor that referenced it. Similarly, an element is privatizable if and only if it was privatizable on every processor that accessed

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

      doall  i = 1, n
        markwrite(K(i))
        markredux(K(i))
S1:     A(K(i)) = .......
        markread(L(i))                           (b)
        markredux(L(i))
S2:       ........... = A(L(i))
        markwrite(R(i))
S3:     A(R(i)) = A(R(i)) + exp()
      enddoall
```

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 `markredux` operation invalidates the indicated array element as a reduction variable since it is accessed outside the reduction statement `S3`.

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 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 operator. The statement is validated as a reduction if it can be shown that x is neither referenced in exp nor anywhere in the loop body outside the reduction statement. For example, although statement `S3` in the loop in Figure 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 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 basic strategy can be extended to handle more complex reduction operations (refer to [24] for details).

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 [16]. 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.

3.2 The Scheduler

The scheduler derives the more restrictive iteration-wise dependence relations from the memory location dependence information found by the inspector. A valid parallel execution schedule for a loop is a partition of the set of iterations into ordered subsets called *wavefronts*, so that all cross-iteration 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 directed acyclic graph (dag) describing the cross-iteration dependences in the loop. We remark that the schedulers described below can be used to construct the full iteration schedule in advance (as described) or they can be interleaved with the executor, i.e., the iterations could be executed as they are found to be ready.

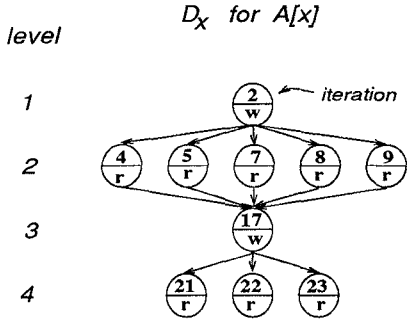
A simple scheduler. A simple scheduler that finds an optimal schedule is sketched in Figure 5(a). In the figure, an array `wf(i)` stores the wavefront found for iteration i, the global variable `done` flags if all iterations have been scheduled, `rdy(i)` signals if iteration i is ready to be executed, 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 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 7–16), and the predecessors of all references whose iterations have not been scheduled (line 10) are examined. An iteration is *not ready* if the iterations of any of its reference's predecessors were not assigned to previous wavefronts (line 11). After all the references are processed, all the iterations are examined (lines 17–19) to see which can be added to the current wavefront: an iteration i is ready (line 18) 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 each write to check all its "parents" (line 10). Fortunately, this problem is easily circumvented by requiring an unscheduled read access to inform its successor's iteration 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 7–16) requires time $O(n_a/p)$ to identify unscheduled iterations (line 9), 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. If $cpl \geq p$, 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 `do while` loop, we would like to examine only those references that are in the topmost unscheduled level of their respective dependence

```
    wf(1:numiter) = 0
    done = .false.
    cpl = 1
 4  do while (done.eq..false.)
      rdy(1:numiter) = .true.
      done = .true.
 7    doall i = 1, numaccess
 8      a = access(i)
 9      if (wf(a.iter) .eq. 0) then
10        for each (b in Pred(a))
11          if (wf(b.iter).eq.0) then
              done = .false.
              rdy(a.iter) = .false.
        endfor
        endif
16    enddoall
17    doall i = 1,numiter
18      if (wf(i).eq. 0 .and. rdy(i).eq..true.) wf(i) = cpl
19    enddoall
      cpl = cpl + 1
    enddo while
```

(a)



(b)

Figure 5: A simple scheduler (a), and the dependence graph for one of the memory locations accessed in the loop (b).

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. 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—which can easily be extracted in the marking phase. Then, in the scheduler, a count of the number of ready accesses for each iteration is computed on a per processor basis in the first `doall` (lines 7–16). In the second `doall` (lines 17–19), 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 the optimized version to outperform the original scheduler if there are multiple levels in the array element dependence graphs. Hence, the determination of which version to use should be made using knowledge gained about the access pattern by the inspector. In [24], we discuss ways to reduce scheduling overhead such as overlapping wavefront computation with actual loop execution and using dynamic ready queues [21].

4 A Comparison with Previous Methods

We now compare the methods described in this paper to several other techniques that have been proposed for analyzing and scheduling `do` loops at run–time. Most of this work has concentrated on developing inspectors. A high level comparison of the various methods is given in Table 1.

Methods utilizing critical sections. The method of Zhu and Yew [34] computes the wavefronts one after another using a method simi-

Method	obtains optimal sched	contains serial portions	requires global synch	restricts type of loop	privat or reduct
New	Yes	No	No	No	P,R
ZY [34]	No[1]	No	Yes[2]	No	No
MP [20]	Yes	No	Yes[2]	No	No
KS [13]	No[3]	No	Yes[2]	No	P
CYT [9]	No[1,3]	No	Yes	No	No
SM [28]	No[3]	No	Yes	Yes[5]	No
SMC [30]	Yes	Yes[4]	Yes	Yes[5]	No
LZ [17]	Yes	No	Yes	Yes[5]	No
P [23]	No	No	No	No	No
RP [25, 26]	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 identifies privatizable and reduction variables, respectively. The superscripts have the following meanings: 1, the method serializes all read accesses; 2, performance 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 applicable only to loops without output dependences (i.e., each memory location is written at most once); 6, the method identifies only fully parallel loops.

lar to the simple scheduler described in Section 3.2. 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 [20] extended this method to allow concurrent reads from a memory location in multiple iterations. These methods run the risk of a severe degradation in performance for access patterns containing *hot spots* (i.e., many accesses to the same memory location). A feature of them is that they use only a shadow version of the shared array whereas all other methods (except [23, 25, 26]) unroll the loop and store all accesses to the shared array.

Krothapalli and Sadayappan [13] proposed a run–time scheme for removing anti and output dependences from loops. For each memory location, their inspector counts the number references to it (using critical sections as in [34]), places them in a dynamically allocated array, and then sorts them by iteration number. After building a dependence graph for each memory location (similar to our arrays R_x), the inspector removes all anti and output dependences by redirecting the accesses to dynamically allocated storage (using an additional level of indirection). Flow dependences are enforced using full/empty bits. To our knowledge, this is the only other run–time privatization technique except for the one described in [25, 26].

Recently, Chen, Yew, and Torrellas [9] proposed an inspector that first builds (in private storage) access lists for each memory location referenced in a processor's assigned iterations (similar to [13] and our inspector's marking phase, except they serialize read accesses), and then links them across processors using a global Zhu/Yew algorithm [34]. Their scheduler/executor uses doacross parallelization [28] (see below). Although this scheme potentially has less communication overhead than [34], it is still sensitive to hot spots and there are cases (e.g., doalls) in which it proves inferior to [34].

Methods for loops without output dependences. This problem has also been studied extensively by Saltz *et al.* [5, 28, 29, 30, 33]. Most of their work assumes that there are no output dependences in the source loop. In doacross parallelization [28], 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 [30], 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 [17] proposed methods of parallelizing the sequential inspector of [30]. In theit *sectioning* method, the loop is chunked and each processor computes an optimal schedule for its chunk, and then these schedules are concatenated together separated by synchronization barriers. In *bootstrapping* technique, the inspector is parallelized (not optimally) using sectioning, but an optimal schedule is produced.

Other methods. In contrast to the above methods which place iterations in the lowest possible wavefront, Polychronopolous [23] 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 [34].

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 [25, 26]: 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.

5 Implementation and Experimental Results

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]. To analyze the overhead incurred by the methods we applied them to access patterns taken from actual programs and to synthetic access patterns.

The methods were implemented in Cedar Fortran [12]. The inspector was essentially as described in Section 3.1. In particular, we implemented the bucket sort version using separate pR and pH data structures for each processor. 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 have not yet included 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. An 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 sim-

ple scheduler described in Section 3.2, 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, but better speedups should be obtainable by interleaving these two tasks (see Section 3.2). We remark that there are other issues to be considered when applying these methods in a real application environment such as memory requirements and known bounds on the source loop's available parallelism (refer to [24] for more details).

Synthetic Loops

Using synthetic loops, we studied the sensitivity of the overhead of the methods to two characteristics of the source do loop: its *average parallelism* (#iterations/*cpl*) 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.

Average parallelism. To isolate the effect 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: each iteration had two accesses (a read followed by a write), and every array element was accessed approximately twice.

We would not expect the inspector's execution time to be dependent on the average parallelism in the source loop since it is fully parallel. However, as the scheduler runs in *cpl* steps, its execution time should be inversely correlated with the average parallelism. In Figures 6 and 7 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." 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 also studied how overhead speedup relates to average parallelism. The inspector's overhead is independent of the average parallelism since it is fully parallel. Although, the scheduler consists of *cpl* steps, it may still exhibit substantial speedups since each step is fully parallel. In fact, in Figures 8 and 9 we show that almost identical speedups are obtained for sequential, partially parallel, and fully parallel loops for both the inspector and scheduler. The slightly diminished slope of the inspector's speedup curve after about 10 processors is because our implementation did not use a "pair–wise" merge among the processors (Section 3.1).

Hotspots. To isolate the effect of the hotspot degree in the source loop on the overhead of the methods, we generated similar access patterns differing only in hotspot degree: all loops had 2048 iterations (each with two accesses), a critical path length of 40, and a loop with hotspot value h contained h references to each of $2048/h$ array elements. We would not expect the methods to be negatively affected by the hot spot degree. In fact, a larger hotspot degree implies fewer non-empty rows in the pR array, and thus we might see improved results in the analysis and scheduling phases. The results in Figure 10 show that in fact the total overhead (inspector + scheduler) is nearly the same for all hotspot degrees.

Loops from the MA28 Solver

We applied the new methods to loops from real applications, both to demonstrate the diversity of partially parallel access patterns and also to reconfirm the conclusions reached above using synthetic loops. For this purpose we chose Loop MA30cd/DO_120 from MA28 (a blocked sparse non-symmetric linear solver [10]). We selected this loop, which performs the forward–backward substitution in the final phase of the blocked sparse linear system solver, because it can generate many diverse access patterns when using the Harwell-Boeing matrices as input. Unfortunately, the loop itself is not a good candidate for parallelization since it performs very little work and is highly imbalanced.

We discuss 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 in [27]), we generated a parallel inspector and computed an optimal parallel execution schedule for the loop. The parallelism profiles obtained (Figures 11 and 12) show the wavefront sizes of the optimal parallel execution schedule and illustrate how the same loop can generate vastly different dependence graphs given different input. Figure 11 shows that most of the iterations of the loop can be executed in the initial wavefronts ($cpl = 114$), which suggests that interleaving the wavefront computation and execution would be more beneficial than overlapping them, so that parallelization can be abandoned when the sequential tail of the profile is reached. Although in Figure 12 most of the iterations are also executed in the initial 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 13 and 14 underscore the need for scheduling and execution strategies that can adapt dynamically depending upon the type of parallelism encountered. Figures 15 and 16 show that overhead speedup is invariant with the parallelism profile. Larger speedups were not obtained since the loop is heavily imbalanced due to the blocked nature of the algorithm used in MA28.

Perfect Benchmark Loops

We applied the methods to three loops contained in the PERFECT Benchmarks [4]. In the analysis phase 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. Figures 17 through 19 show 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. We note that these loops could also be identified by the LRPD test [25, 26], a run–time test for identifying fully parallel loops, i.e., loops that can be transformed into `doalls` using privatization and reduction parallelization. Although the LRPD test has a smaller overhead than the methods presented here, it cannot extract partial parallelism.

In *BDNA–ACTFOR–Loop 240*, the shared array under test is accessed through a subscript array computed inside the loop which is found to be privatizable in the analysis phase (Figure 17). In *MDG–INTERF–Loop 1000*, it is also found that the shared array under test is privatizable in the analysis phase (Figure 18). In *OCEAN–FTRVMT–Loop 109*, all accesses to the shared array are found to

be unique in the analysis phase. 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* [30]. 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 19, with schedule reuse we obtain scalable speedups that are comparable to the ideal speedup.

6 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 such as those described here can break this barrier and extract much of the available parallelism from even the most complex programs. The scalability of our methods ensures that their run–time overhead can be reduced to an insignificant fraction of the program's sequential execution time, which implies that their significance will only increase with the advent of massively parallel processors (MPPs).

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. 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 [25, 26, 27].

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 grateful to Richard Cole for 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 and others. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. TR. 827, Ctr. for Supercomputing R.&D., Univ. 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 Trans. on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[7] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *J. of Supercomputing*, pp. 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 *Proc. of Supercomputing 1994*, pp. 518–527, Nov. 1994.

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

[11] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Lecture Notes in Comp. Science 589. Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pp. 65–83, Aug. 1991.

[12] 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.

[13] V. Krothapalli and P. Sadayappan. An approach to synchronization of parallel computing. In *Proc. of the 1988 Int Conf. on Supercomputing*, pp. 573–581, June 1988.

[14] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proc. of the 1986 Int. Conf. on Parallel Processing*, pp. 869–876, Aug. 1986.

[15] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. of 8th ACM Symp. Princip. Prog. Lang.*, pp. 207–218, Jan. 1981.

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

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

[18] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of the 19th Int. Symp. on Comput. Arch.*, pp. 313–322, 1992.

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

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

[21] J. Moreira and C. Polychronopoulos. Autoscheduling in a distributed shared-memory environment . TR. 1373, Ctr. for Supercomputing R.&D., Univ. of Illinois, Urbana, June 1994.

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

[23] C. Polychronopoulos. Compiler optimizations for enhancing parallelism and their Impact on architecture design. *IEEE Trans. Comput*, C-37(8):991–1004, Aug. 1988.

[24] L. Rauchwerger, N. Amato and D .Padua. Run-time methods for parallelizing partially parallel loops. TR. 1400, Ctr. for Supercomputing R.&D., Univ. of Illinois, Urbana, IL, May 1989.

[25] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proc. of the 1994 Int Conf. on Supercomputing*, pp. 33–43, July 1994.

[26] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June, 1995.

[27] L. Rauchwerger and D. Padua. Parallelizing `while` loops for multiprocessor systems. In *9th Int. Parallel Process. Symp.*, April, 1995.

[28] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In Dr. H.D. Schwetman, editor, *Proc. of the 1991 Int. Conf. on Parallel Processing*, pp. 174–178. CRC Press, Inc., 1991. Vol. II - Software.

[29] J. Saltz, R. Mirchandaney, and K. Crowley. The doconsider loop. In *Proc. of the 1989 Int Conf on Supercomputing*, pp. 29–40, June 1989.

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

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

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

[33] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proc. of the 1991 Int. Conf. on Parallel Processing*, pp. 26–30. CRC Press, Inc., 1991. Vol. II - Software.

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

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

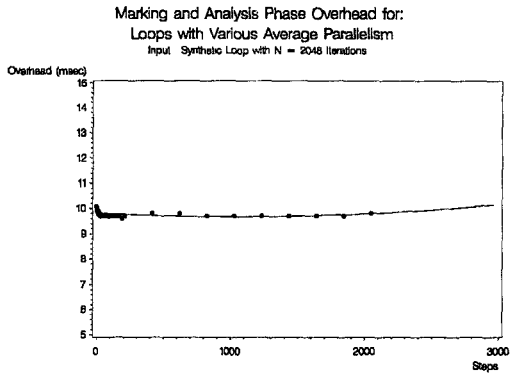


Figure 6:

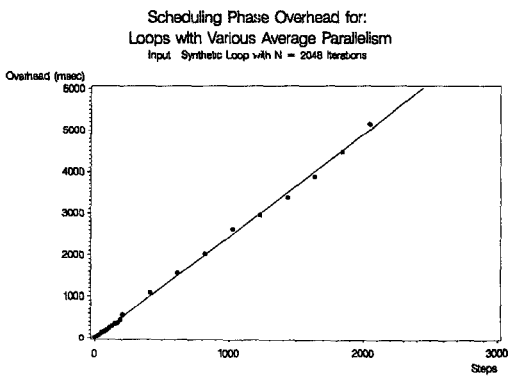


Figure 7:

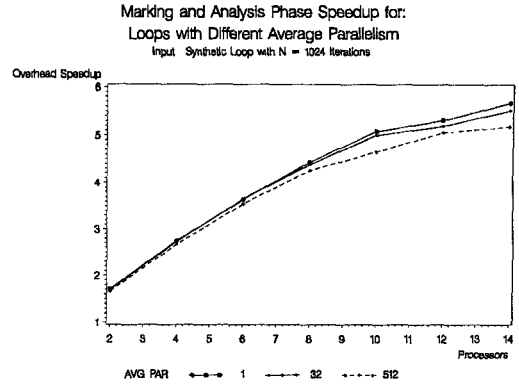


Figure 8:

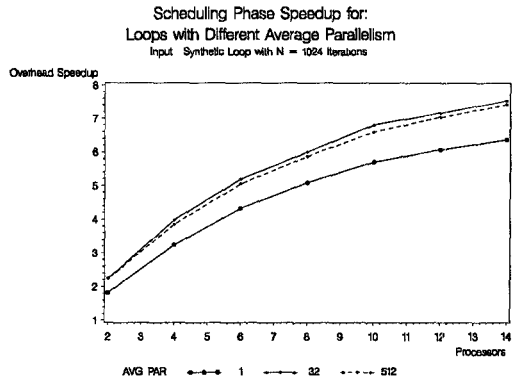


Figure 9:

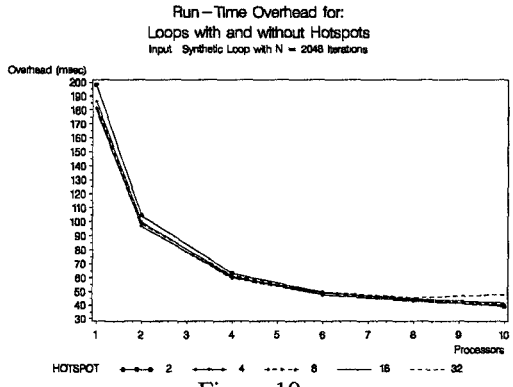


Figure 10:

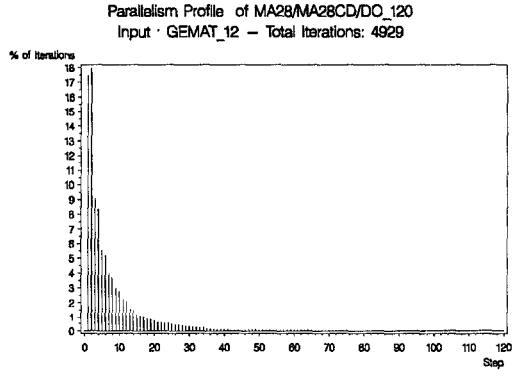


Figure 11:

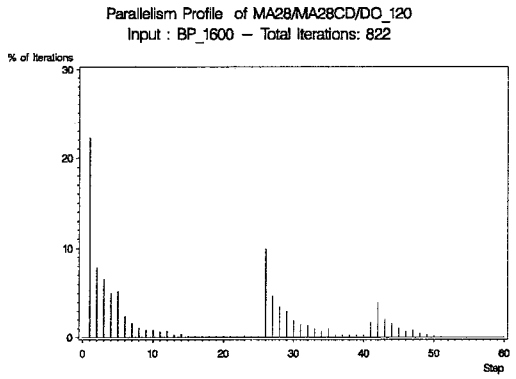


Figure 12:

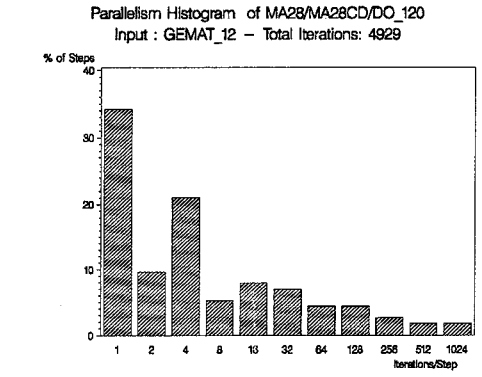


Figure 13:

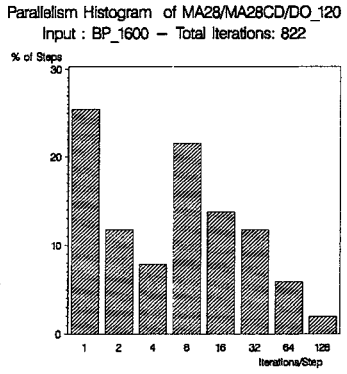


Figure 14:

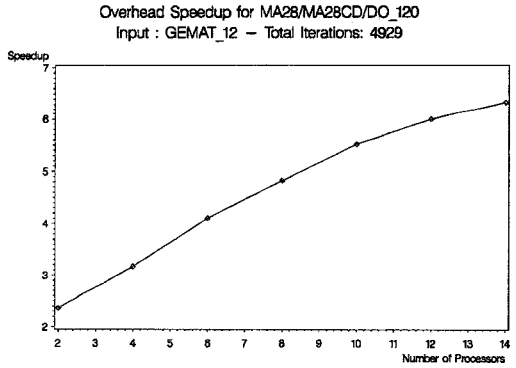


Figure 15:

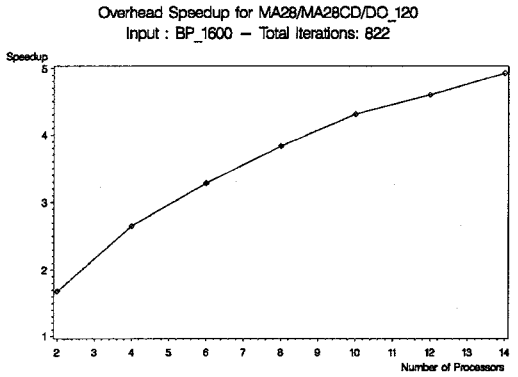


Figure 16:

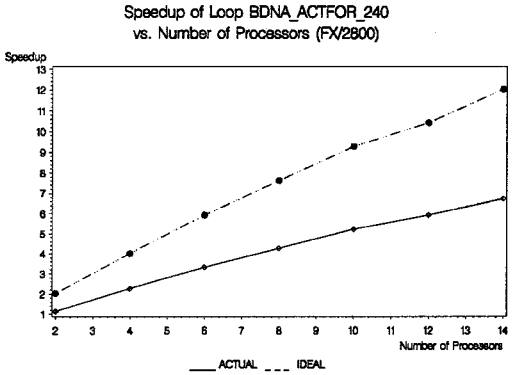


Figure 17:

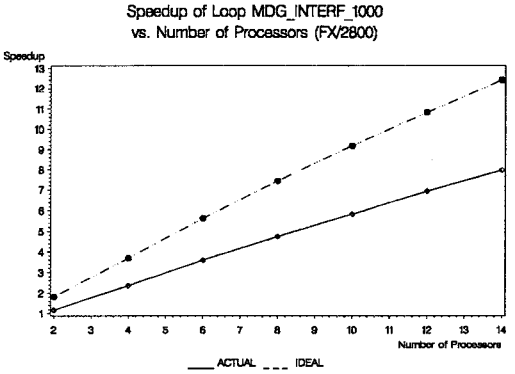


Figure 18:

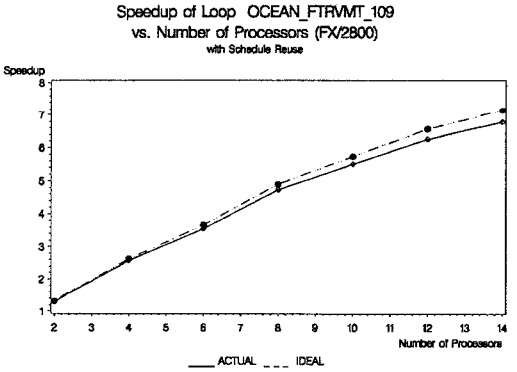


Figure 19: