Submitted to *IEEE Transactions on Parallel and Distributed Systems*. (CSRD Technical Report 1383.)

The Privatizing DOALL Test:
A Run–Time Technique for DOALL Loop Identification and Array Privatization[†‡]

Lawrence Rauchwerger and David Padua
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801

Abstract

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 important issue because a large class of complex simulations used in industry today have irregular domains and/or dynamically changing interactions. To handle these types of problems methods capable of automatically extracting parallelism at *run–time* are needed. For this reason, we have developed the Privatizing DOALL test – a technique for identifying fully parallel loops at run–time, and dynamically privatizing scalars and arrays. The test is fully parallel, requires no synchronization, is easily automatable, and can be applied to any loop, regardless of its access pattern. We show that the expected speedup for fully parallel loops is significant, and the cost of a failed test (a not fully parallel loop) is minimal. We present experimental results on loops from the PERFECT Benchmarks which confirm our conclusion that this test can lead to significant speedups.

Index Terms – Doall, Run-time, Parallel, Privatization, Speculative, Race Detection

[†]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.

[‡]A preliminary version of this paper [26] was presented at the *8th ACM International Conference on Supercomputing*, July 1994, Manchester, England.

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, increases the possibility of error over sequential programming and the resulting code may not be portable to other 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 [24, 35], 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 [10].

Thus, since the available parallelism in theses types of applications cannot be determined statically by present parallelizing compilers [8, 10, 13], it has become clear that compile-time analysis must be complemented by new methods capable of automatically extracting parallelism at *run–time* in order to realize the potential of parallel computing. 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 [19, 29, 37].

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 [31, 32]. Some of today's parallelizing 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, new techniques have been developed for the run–time analysis and scheduling of loops with cross-iteration dependences [7, 11, 16, 19, 22, 25, 27, 28, 29, 36, 37]. 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. Most of these schemes partition 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. The wavefronts themselves are executed sequentially by placing a synchronization barrier between each wavefront.

1.1 Our approach

In this paper we approach the problem of determining the parallelism of loops at run–time from a different viewpoint – instead of finding a valid parallel execution schedule for the loop, we focus on the problem of simply deciding if the loop is fully parallel, that is, determining whether or not the loop has cross-iteration dependences. Our interest in identifying fully parallel loops is motivated by the fact that they arise frequently in real programs. In addition, an efficient test for full parallelism of a loop could be used as a building block for more complex run–time dependence analysis techniques in order to produce valid execution schedules for partially parallel loops.

As we show, the analysis needed to test whether or not a loop is fully parallel can be done very efficiently at run–time without the use of any synchronization, and can produce scalable speedups. User directives or execution statistics can be used to identify the loops to which this test is to be applied. The techniques presented are also capable of eliminating some memory–related dependences by dynamically privatizing scalars and arrays. Our methods are fully parallel, require no synchronization, and can be applied to any loop.

Our methods can also be used for detecting *access anomalies* [30, 12] or *race conditions* [14] in parallel programs,

```
S1:  DO i = 2, n
S2:   A[i] = A[i] + A[i-1]
S3:  ENDDO
```

(a)

```
S1:  DO i = 1, n
S2:   A[i] = 2*A[i]
S3:  ENDDO
```

(b)

```
S1:  DO i = 1, n/2
S2:   tmp = A[2*i]
S3:   A[2*i] = A[2*i-1]
S4:   A[2*i-1] = tmp
S5:  ENDDO
```

(c)

Figure 1:

i.e., when the same memory location is accessed by more than one concurrent thread without synchronization, and it is written in at least one of the threads. In fact, our methods are more closely related to some of the techniques for detecting access anomalies [30], than they are to the previous work mentioned above for finding valid execution schedules for a partially parallel loops. After presenting our methods, we discuss their relationship to the work done on detecting access anomalies in Section 8.

We discuss our analysis techniques in Sections 2–7. In Section 8, we describe how these techniques can also be used for debugging parallel programs and for speculative parallelization. In Section 9, we discuss some important compile-time issues. In Section 10, we describe related work for constructing valid parallel execution schedules for partially parallel loops. Finally, in Section 11, we present some experimental measurements of loops from the PERFECT Benchmarks executed on the Alliant FX/80 and 2800 that show that our techniques are effective in producing speedups – even though the analysis is done without the help of any special hardware devices. It is conceivable, and we believe desirable, that future machines would include special hardware devices to accelerate the run–time analysis and in this way widen the range of applicability of the techniques and increase potential speedups.

2 Detecting Parallelism at Run–Time

A loop can be executed in fully parallel form, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the execution ordering of the data accesses from different iterations. In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the *data dependence* relations between the statements in the loop body must be analyzed [5, 17, 24, 35, 38]. 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 are data producer and consumer dependences, i.e., they 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 if the loop is executed in fully parallel form. The iterations of such a loop are not independent because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration of the loop. For example, the iterations of the loop in Fig. 1(a), which computes the prefix sums for the array A, must be executed in order of iteration number because iteration $i+1$ needs the value that is produced in iteration i, for $2 \leq i \leq n$.

In principle, if there are no flow dependences between the iterations of a loop, then the loop may be executed in fully parallel form. The simplest situation occurs when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and the loop, as is, can be executed in parallel. For example, there are no cross-iteration dependences in the loop shown in Fig. 1(b), since iteration i only accesses the data in $A[i]$, for $1 \leq i \leq n$. If there are no flow dependences, but there are anti or output dependences, then the loop must be modified to remove all these dependences before it can be executed in parallel. Unfortunately, not all situations can be handled efficiently. In order to remove certain types of anti and output dependences a transformation called *privatization* can be applied to the loop. Privatization creates, for each processor cooperating on the execution of the loop, private copies of program variables that give rise to anti or output dependences (see, e.g., [24, 9, 20, 21, 33, 34]). The loop shown in Fig. 1(c), which, for even values of i, swaps $A[i]$ with $A[i-1]$, is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S4` of iteration i and statement `S2` of iteration $i+1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`.

In the next sections, we describe run–time techniques that can be used to determine if a loop can be executed in

parallel. We first describe the *DOALL test* which determines if the loop, in its original form, can be executed in parallel, i.e., it decides if there are any cross-iteration dependences in the loop. We then explain how the DOALL test can be augmented to determine at run–time whether all existing memory–related dependences can be removed by privatization. If it is found that all dependences can be eliminated, then the *Privatizing DOALL test* transforms the loop by privatizing the variables which give rise to the anti and/or output dependences. In order to identify the dependence relations among the iterations of the loop, both tests inspect the accesses to the variables that cannot be analyzed at compile time. Briefly, the inspection is done by using shadow versions of the variables under scrutiny to follow (keep a record of) the data access pattern of the original loop. After processing all the accesses contained in the original loop, some additional computation determines whether all iterations of the loop can be performed in parallel while guaranteeing the semantics of the loop. For the Privatizing DOALL test, an additional phase may be required to actually allocate the private variables.

It must be emphasized that both DOALL tests are designed to be used only on loops for which the compiler could not evaluate with certainty the data dependence relations. We recall that there are several possible situations in which it is either difficult or impossible to determine the data dependence relationships at compile time: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, nonlinear subscript expressions (a fairly rare case) and, most often, subscripted subscripts. Another important point is that these run–time tests must be fully parallel. If the tests cannot be executed in parallel, then they would not scale with the number of processors or the data size, and the overhead associated with the tests could become a sequential bottleneck of the loop.

3 The DOALL Test

The DOALL test described below is a pass/fail test for full parallelism of a loop, i.e., it detects if there are any cross-iteration dependences in the loop. If there are any dependences, then this test does not identify them, it only flags their existence. We first describe the DOALL test, as applied to a shared array A, and then give a few examples illustrating its use.

DOALL Test

1. *Marking Phase.* For each shared array $A[1:s]$ whose dependences cannot be determined at compile time, we declare read and write shadow arrays, $A_r[1:s]$ and $A_w[1:s]$, respectively. The shadow arrays are initialized to zero, and are marked as follows.

   In parallel, for each iteration i of the loop, process all accesses to the shared array A:

   (a) Writes: If this is the first write to this array element in this iteration, then set the corresponding element in A_w.
   (b) Reads: If this array element is *never* written in this iteration, then set the corresponding element in A_r.
   (c) Count the total number of write accesses to A that are marked in this iteration, and store the result in $tw_i(A)$, where i is the iteration number.

2. *Analysis Phase.* For each shared array A under scrutiny:

   (a) Compute (i) $tw(A) = \sum tw_i(A)$, i.e., the total number of writes that were marked by all iterations in the loop, and (ii) $tm(A) = sum(A_w[1:s])$, i.e., the total number of marks in $A_w[1:s]$.
   (b) If $any(A_w[1:s] \wedge A_r[1:s])$[1], i.e., if the marked areas are common *anywhere*, then the loop *is not a DOALL* and the phase ends. (Since we read and write from the same location in different iterations, there is at least one flow or anti dependence.)
   (c) Else if $tw(A) = tm(A)$, then the loop *is a DOALL* and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)
   (d) Otherwise, it *is not a DOALL*. (There are output dependences since we overwrite at least one memory location.)

[1] any returns the "OR" of its vector operand's elements, i.e., $any(v[1:n]) = (v[1] \vee v[2] \vee \ldots \vee v[n])$.

```
S1: DO i = 1, 8
S2:   A[W[i]] = ...
S3:        ... = A[R[i]]
S4: ENDDO
```

```
W[1:8] = [ 1 3 2  3 7 5 6 12]
R[1:8] = [ 1 9 2  2 7 8 8 12]

W'[1:8] = [ 1 3 2  4 7 5 6 12]
R'[1:8] = [ 1 9 2 10 8 8 8 12]
```

	Position in shadow arrays												Written	Counted
	1	2	3	4	5	6	7	8	9	10	11	12	$tm(A)$	$tw(A)$
$A_w[1:12]$	1	1	1	0	1	1	1	0	0	0	0	1	7	8
$A_r[1:12]$	0	1	0	0	0	0	0	1	1	0	0	0		
$A_w[1:12] \wedge A_r[1:12]$	0	1	0	0	0	0	0	0	0	0	0	0		
$A'_w[1:12]$	1	1	1	1	1	1	1	0	0	0	0	1	8	8
$A'_r[1:12]$	0	0	0	0	0	0	0	1	1	1	0	0		
$A'_w[1:12] \wedge A'_r[1:12]$	0	0	0	0	0	0	0	0	0	0	0	0		

Figure 2: Results of the DOALL test.

To illustrate the DOALL test we consider the loop shown in Fig. 2. Assume that the shared array $A[1:12]$ is accessed in a manner such that the dependences cannot be determined at compile time. In the first example, the reference pattern of A within the loop is given by the subscript arrays $W[1:8]$ and $R[1:8]$. The DOALL test allocates, and initializes to zero, the write and read shadow arrays, $A_w[1:12]$ and $A_r[1:12]$, respectively. After marking and counting, we obtain the results depicted in the table. Because $A_w[2] = A_r[2] = 1$, we know there exists at least one flow or anti dependence. Since the number marked does not equal the number written, we know that there are output dependences. Therefore, the loop cannot be executed in parallel. In the second example, we use the subscript arrays $W'[1:8]$ and $R'[1:8]$, and the shadow arrays $A'_w[1:12]$ and $A'_r[1:12]$. Because the number of write accesses marked equals the number written, and since $A_w[1:12] \wedge A_r[1:12]$ is zero everywhere, we conclude that there are no cross-iteration dependences, i.e., the loop can be executed in parallel.

An implementation of the DOALL test will usually not adhere exactly to the above description. In particular, is should be optimized to take advantage of the target machine's architecture (e.g., private storage for the processors), and any special operations available on it. We discuss some of these implementation details in Section 5, and then analyze the complexity of the DOALL test in Section 7.

4 The Privatizing DOALL Test

The DOALL test described above is only able to detect the presence of dependences among the iterations of the loop. In this section we explain how to augment the test so that it can determine if all dependences caused by a particular array can be removed by privatizing the array. If it finds that all the dependences can be eliminated, then the Privatizing DOALL test (PD test) transforms the loop by allocating the private variables.

We now define a private variable, and state the criterion that must be satisfied in order for a variable to be determined as privatizable by the PD test.

Definition. A variable is *private* if its scope is the loop body. Therefore a *private variable* can only be accessed by the loop iteration to which it belongs.

Privatization Criterion. Let A be a shared array that is referenced in a loop L. A can be *privatized* by the PD test 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, there are some types of dependences that the PD test does not handle. Specifically, if a shared variable is initialized by reading a value that is computed outside the loop, then we will not privatize that variable. Such variables could be privatized if a *copy–in* mechanism for the external value is provided. The techniques dealing with this situation will be discussed in Section 6.3. The *last value assignment* problem is the conceptual analog of the copy–in problem. If a privatized

```
S1:  DO i = 1, 8
S2:      ...        = A[R1[i]]          R1[1:8] = [ 2 2 2 10 8 8 8 10]
S3:     A[W[i]]     = ...                W[1:8] = [ 1 3 5  4 7 3 6 12]
S4:      ...        = A[R2[i]]          R2[1:8] = [ 1 3 2 10 7 3 8 12]
S5:  ENDDO
```

	Position in shadow arrays												Written	Counted
	1	2	3	4	5	6	7	8	9	10	11	12	$tw(A)$	$tm(A)$
$A_w[1:12]$	1	0	1	1	1	1	1	0	0	0	0	1	8	7
$A_r[1:12]$	0	1	0	0	0	0	0	1	0	1	0	0		
$A_{np}[1:12]$	0	1	0	0	0	0	0	1	0	1	0	0		
$A_w[1:12] \wedge A_r[1:12]$	0	0	0	0	0	0	0	0	0	0	0	0		
$A_w[1:12] \wedge A_{np}[1:12]$	0	0	0	0	0	0	0	0	0	0	0	0		

Figure 3: Results of the Privatizing DOALL test.

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. One way this problem can be handled is to associate with each private variable a time stamp (iteration number) that is updated at every write. After the loop has been executed, the value of the private variable with the latest time stamp is copied to the original version of the variable. If all iterations of the loop are writing the same elements of the array then the last value assignment problem can be solved by writing to the original version of the variable during the last iteration. It should be noted that private loop variables are seldom live after the loop.

In order to simplify the description of the PD test given below, the last value assignment problem is not addressed. (The additions to the DOALL test are italicized.)

Privatizing DOALL Test (PD Test)

1. *Marking Phase.* For each shared array $A[1:s]$ whose dependences cannot be determined at compile time, in addition to the shadow arrays, $A_r[1:s]$ and $A_w[1:s]$, *we declare a shadow array* $A_{np}[1:s]$ *that will be used to flag array elements that* cannot *be privatized.* As before, the shadow arrays are initialized to zero. *Initially, the test assumes that all array elements* are *privatizable, and if it is found in any iteration that an element is read before it is written, then it is marked as non–privatizable.*

   In parallel, for each iteration i of the loop, process all accesses to the shared array A:

   (a) Writes: If this is the first write to this array element in this iteration, then set the corresponding element in A_w.

   (b) Reads: If this array element is never written in this iteration, then set the corresponding element in A_r. *If this array element has not been written in this iteration before this read access, then set the corresponding element in* A_{np}*, i.e., mark it as* non–privatizable.

   (c) Count the total number of write accesses to A that are marked in this iteration, and store the result in $tw_i(A)$, where i is the iteration number.

2. *Analysis Phase.* For each shared array A under scrutiny:

   (a) Compute (i) $tw(A) = \sum tw_i(A)$, i.e., the total number of writes that were marked by all iterations in the loop, and (ii) $tm(A) = sum(A_w[1:s])$, i.e., the total number of marks in $A_w[1:s]$.

   (b) If $any(A_w[1:s] \wedge A_r[1:s])$, i.e., if the marked areas are common *anywhere*, then the loop *is not a DOALL* and the phase ends. (Since we read and write from the same location in different iterations, there is at least one flow or anti dependence.)

   (c) Else if $tw(A) = tm(A)$, then the loop *is a DOALL* and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)

   (d) *Else if* $any(A_w[1:s] \wedge A_{np}[1:s])$*, then the loop* is not a DOALL *and the phase ends. (There is at least one dependence that cannot be removed by privatization).*

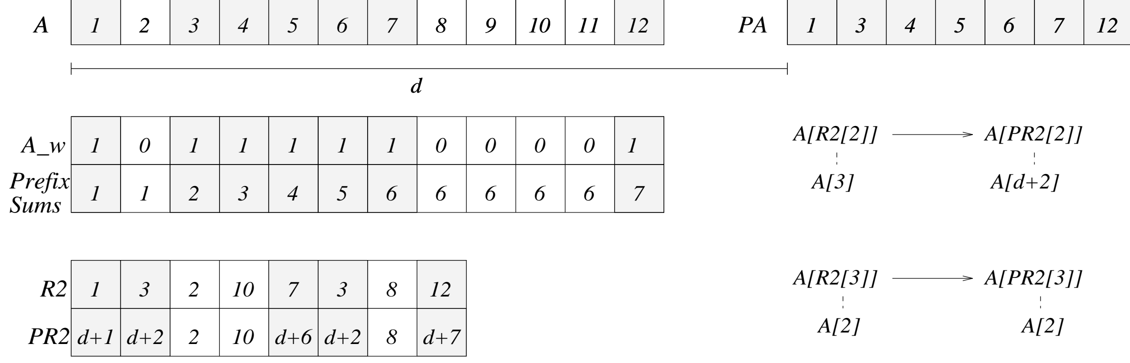

Figure 4: Allocating the private variables; privatized elements are shaded.

(e) *Otherwise, the loop* can be made into a DOALL *by privatizing all elements of the shared array that are written in the loop. (We remove all memory–related dependences by privatizing these array elements.)*

In order to illustrate the differences between the DOALL test and the PD test, we consider the loop shown in Fig. 3, which contains memory–related dependences that can be removed by privatization. Assume the loop has 8 iterations, accesses a vector of dimension 12, and that the access pattern is given by the subscript arrays $R1$, $R2$ and W. After marking and counting we obtain the results depicted in the table. Since $A_w[1:12] \wedge A_r[1:12]$ and $A_w[1:12] \wedge A_{np}[1:12]$ are zero everywhere, the loop can be made into a DOALL, but only after privatization because $tw(A) \neq tm(A)$.

4.1 Allocating the private variables

If the PD test determines that privatization can be used to transform the loop into a DOALL, then there are two choices: privatize the entire shared variable, or only privatize the shared memory locations (e.g., elements of the array) that are *written* in the loop. If only the elements that are written are privatized, it is possible that this technique might yield performance gains over privatizing the whole array (which is the traditional compile time privatization technique [20, 21, 33, 34]) for two reasons: First it may save work in the last value assignment step and second, if the data access pattern is sparse enough, the reduction in the size of the working set could improve the cache performance perhaps to the point of superlinear speedups.

If it is decided to privatize only the elements that are written, then the private variables can be allocated as follows. Consider the array A from the previous example. The elements of A that are privatized are exactly those elements k for which $A_w[k] = 1$. Since $tm(A) = 7$, we allocate enough space for seven elements of A; denote this space by $PA[1:7]$. We can determine the positions of the privatized elements of A in PA from the prefix sums of $A_w[1:12]$, e.g., the private version of $A[5]$ is contained in $PA[4]$ since the prefix sum value of $A_w[5] = 4$ (see Figure 4).

In general, on each access to a shared array element $A[k]$, it must be determined whether or not $A[k]$ has been privatized, e.g., by checking $A_w[k]$. However, in the case of subscripted subscripts, we can remove the need for this check as follows. Each iteration is provided with a private subscript array (of the same dimension as the original subscript array), and all references to the subscript array use the private version. If $A[k]$ is not privatized, then references to it in the subscript array remain the same. If $A[k]$ is privatized, then occurrences of k in the subscript array are replaced by the prefix sum value in $A_w[k]$ plus the offset between the starting addresses of A and PA. The private version $PR2$ of the subscript array $R2$ is shown in Figure 4, where d is the offset between the starting addresses of PA and A, i.e., $d = \&PA[0] - \&A[0]$. Actually, we can handle other cases besides subscripted subscripts in a similar manner by constructing a private subscript array $PS[1:s]$, and and transforming references such as $A[i]$ into $A[PS[i]]$. The values of PS are set as follows: if $A[k]$ is not private, then $PS[k] = k$, and if $A[k]$ is private, then $PS[k] = s_k + d$, where s_k is the prefix sum value in $A_w[k]$, and d is again the offset between the starting addresses of A and PA. Since this technique requires indexing out of bounds, it may cause problems on certain architectures and would be best implemented in machine language.

Original Version

```
      DO 140 I=1,NP
            DO 130 J=1,I
               IJ=IA(I)+J
      ....
               DO 120 K=1,I
                  MAXL=K
                  IF(K.EQ.I) MAXL=J
                  DO 110 L=1,MAXL
                     KL=IA(K)+L
           .....
C **DOALL Test checks the writes to X**
                     X(IJ,KL)=....
 110              CONTINUE
 120           CONTINUE
 130        CONTINUE
 140        CONTINUE
```

Extracted Loop for Marking

```
    integer pX_w(0:numproc-1,:,:), ptw(0:numproc-1)
    integer X_w(:,:),tw,tm
    DOALL  I = 1,NP
C **-- private variables
       integer IJ, MAXL, J, K, L, KL
C **-- do once per processor**
       pX_w(my_rpoc,:,:) = 0
       ptw(my_proc) = 0
    LOOP ** do once per iteration (I) **
       DO J=1,I
          IJ=IA(I)+J
          DO K=1,I
             MAXL=K
             IF(K.EQ.I) MAXL=J
             DO L=1,MAXL
                KL=IA(K)+L
C **mark shadow array X_iw (if not already marked)**
                IF (pX_w(my_proc,IJ ,KL) .NE. I) THEN
                   pX_w(my_proc,IJ,KL) = I
                   ptw(my_proc) = ptw(my_proc) + 1
                ENDIF
             ENDDO
          ENDDO
       ENDDO
    ENDDOALL
C **transfer private arrays to global shadow array**
    DOALL i=1,nproc
      if (pX_w(i,:,:) .ne. 0) X_w(:,:) = pX_w(i,:,:)
    ENDDO
C **accumulate global number of marks attempted   **
    tw = sum(ptw(0:numproc-1))
C **accumulate number of marks in global shadow array**
    tm = sum(X_w(:,:))
C **execute loop appropriately**
    IF (tw .eq. tm) THEN
       execute loop as a DOALL
    ELSE
       execute loop sequentially
```

Figure 5:

5 Implementing the Tests on a Shared Memory Machine

As mentioned above, implementations of the DOALL tests should be optimized to take advantage of any features available on the target architecture. For example, on shared memory multiprocessor machines, each processor may have some private memory. In this case, contention in the global shared memory can often be reduced if a significant portion of the computation can be performed in the processors' private memories, and then combined to give the final result.

5.1 Private shadow structures

The DOALL tests can take advantage of the processors' private memories by using private shadow structures for the marking phase of the tests. Then, at the conclusion of the private marking phase, the contents of the private shadow structures are merged into the global shadow structure referred to in the description of the tests (Sections 3 and 4). Note that since the order of the writes (marks) to an element of the shadow structure is not important, all processors can transfer the contents of their private shadow structures to the global shadow structure without synchronization. Also, the transfer from the private to the global shadow structures could be done in vector–concurrent mode if available.

In fact, using shadow structures that are private to each processor enables some additional optimization of the DOALL tests as follows. Since the shadow structures are private to each processor, the iteration number can be used as the

"mark" and assigned to them. In this way, no re-initialization of the shadow structures will be required between successive iterations, and checks such as "is this the first write access to this memory location in this iteration?" simply require checking if the corresponding element in the write shadow structure A_w is marked with the iteration number. Another benefit of the iteration number "marks" is that they can double as time–stamps, which are needed for performing the last–value assignment to any shared variables that are live after loop termination.

An example which uses private shadow arrays that are marked with the iteration number is shown in Figure 5. The original loop (shown on the left) is extracted from loop 140 in subroutine INTGRL of TRFD from the PERFECT Benchmarks. The DOALL test is applied to the shared array X. In the inspector loop (shown on the right), writes are marked in the private shadow array `X_w` by iteration number. The number of writes marked is recorded in the private variable `tw_i`. At loop termination, the total number of marks that were attempted is accumulated (with synchronization) into the global variable `tw(X)`, and the contents of the private shadow arrays are transferred to the global shadow array. Then, the loop is executed appropriately.

5.2 Hash tables

Thus far, we have assumed that the shadow structures are structurally equivalent to the original shared variable, and that each processor has its own private copy of the entire shadow structure. A potential problem with this strategy is that the number of operations in the DOALL tests would not scale, i.e., although, as the number of processors increases, each processor is responsible for fewer accesses, each processor must still inspect every element of its private shadow structure when transferring it to the global shadow structure. Another related issue is that the resource consumption (memory) would not scale.

To combat these problems we would like a processor to have private shadow copies only of the *elements* accessed in the iterations assigned to that processor for the PD test. Unfortunately, the shadow elements needed by a processor will generally not form a contiguous subset of the shadow structure because most of the loops that require run-time analysis have irregular access patterns. However, these issues can be addressed by using hash tables for the shadow structures. Of course, the cost per access would increase since array accesses would be replaced by accesses to a hash table. Thus, there exists a trade-off that should be resolved using a cost model.

6 Variants of the PD Test

6.1 A processor–wise version of the PD test

The PD Test, as described in Sections 3 and 4, determines whether a loop has any cross-*iteration* data dependences that cannot be removed by privatization. It turns out that essentially the same method can be used to test whether the loop, as executed, has any cross-*processor* data dependences that cannot be removed by privatization.[2] Assuming each processor has its own private shadow structures as discussed in Section 5.1, the only difference is that all checks in the test refer to processors rather than to iterations, i.e., replace "iteration" by "processor" in the description of the PD test so that all iterations assigned to a processor are considered as one "super–iteration" by the test.

Note that a loop that is not fully parallel could potentially pass the processor-wise version of the PD test because data dependences among iterations assigned to the same processor will not be detected. However, this is acceptable (even desirable) as long as these iterations are guaranteed to be executed on the same processor in the same order. Therefore, if each processor keeps a record of the iterations that it executes during the PD test, this information could be used to statically schedule the future parallel execution of the loop.

In Section 5.1 we noted that reinitialization of the private shadow structures could be avoided if iteration numbers were used for marking in the PD test. A potential advantage of the processor–wise version of the PD test is that no reinitialization is needed since we are only interested in cross–processor dependences. Therefore, memory requirements could be reduced by using boolean shadow structures. Of course, if any variable requires a last–value assignment, then it might still be desirable to use iteration numbers for marking.

[2] This fact was noted by Santosh Abraham[1].

6.2 Distinguishing between fully independent and privatizable accesses

Since privatization generally implies replication of program variables (i.e., an increase in memory requirements), this transformation should be avoided when there is no benefit to be gained. In Section 4.1 we noted that the PD test offers potential gains in this regard over privatizing the entire shared array, i.e., the PD test can identify and privatize only the elements that are actually written. However, if an element is written only once in the loop, then there is no need for it to be privatized and replicated on multiple processors.

Although the standard version of the PD test described in Section 4 does not determine if an element is written more than once in the loop, it can easily be augmented to provide this information. The simplest approach is to use another shadow structure A^p_{mw} to flag the array elements which have been written multiple times. On a write to an element during the marking phase, the corresponding entry of A^p_{mw} is marked if it is known that that element has been written before (in any iteration), which can be determined by checking the corresponding entry in A_w. The global shadow structure A_{mw} is now constructed from the private (per processor) shadow structures A^p_{mw} and A_w. First, the marked elements in the private shadow structures A^p_{mw} can be transferred directly, without synchronization, into A_{mw}. If the private shadow structures A_w are merged pairwise into the global shadow structure A_w, then during this process it can be determined if an element is marked in more than one A_w, i.e., if it was written more than once. We note that the need for the pairwise merges can be eliminated if the accesses to the global shadow structure are placed in critical sections. If it is found that the loop can be transformed into a DOALL by privatization, then every element that is written more than once, i.e, with $A^p_{mw} = 1$, is privatized, and those elements that are only written once are not. It is simple to see that the need for the additional structure A^p_{mw} can be eliminated by using three states for the structure A_{np}, e.g., negative values for multiple writes and privatizable, 0 for at most one write and privatizable (initial value), and positive values for not privatizable (once set, never unset).

If the processor–wise version of the PD test is used then the elements that are written more than once can be identified in essentially the same manner. Note that for the processor–wise version it is possible that the number of private variables could be reduced even more since only the processors that actually write the elements need copies, and the private structures identify these elements.

6.3 Supporting copy-in of external values

Suppose that a loop is determined as fully parallel by the PD test except for the accesses to one element a. If the first time(s) a is accessed it is read, and for every later iteration that accesses a it is always written before it is read, then the loop could actually be executed as a DOALL by having the initial accesses to a *copy–in* the global value of a, and the iterations that wrote a used private copies of a. The PD test can be augmented to detect this situation by keeping track of the maximum iteration i^+_r that read a (before it was ever written), and the minimum iteration i^-_w that wrote a. Then, if $i^+_r \leq i^-_w$, the loop can be executed in parallel. In order to collect this information we need two additional private shadow structures, which are merged, pairwise, into the global shadow structure (as in Section 6.2).

We can remove the need for these additional shadow structures in a restricted version of the processor–wise PD test where the iteration space is assigned to the processors in contiguous chunks, i.e., processor i gets iterations $(n/p) * i$ through $(n/p) * (i+1) - 1, 0 \leq i < p$. Then, we need only check that the first write to a appears in a chunk that is not less than the last chunk in which a is marked as non-privatizable. The potential disadvantage of this chunking method is that the assignment of iterations to processors may produce an imbalanced workload.

7 Complexity of the DOALL Tests

We now show that the time required by the DOALL tests is $T(n, s, a, p) = O(na/p + \log p)$ where p is the number of processors, n is the total iteration count of the loop, s is the number of elements in the shared array, and a is the (maximum) number of accesses to the shared array in a single iteration of the loop. We assume that the implementations of the tests have been optimized as discussed in Section 5, i.e., the tests use private shadow structures and, if necessary, hash tables. The analysis below is also valid for the variants of the PD test discussed in Section 6.

The marking phase (Step 1) takes $O(na/p + s + \log p)$ time, i.e., proportional to $\max(na/p, s, \log p)$ time. We record the read and write accesses, and the privatization flags in private shadow arrays. In order to check whether for a read of an element there is a write in the same iteration, we simply check that element in the shadow array – a constant

time operation. All accesses can be processed in $O(na/p)$ time, since each processor will be responsible for $O(na/p)$ accesses. After all accesses have been marked in private storage, the private shadow arrays can be merged into the global shadow arrays in $O(s + \log p)$ time; the $\log p$ contribution arises from the possible write conflicts in global storage that could be resolved using software or hardware combining. When using a variant of the test that requires pairwise merging of the private shadow arrays into the global shadow array, we can also perform the merge in $O(s+\log p)$ time, i.e., in each subsequent merge we use twice as many processors so the time is $O(s(\frac{1}{2^0} + \frac{1}{2^1} + \ldots + \frac{1}{2^{\log p}})) = O(s)$. In this case the $\log p$ contribution is due to the $\log p$ stages of the merge, and there will be no write conflicts as was possible in the one stage merge.

If $s \gg na/p$, then the time required to merge the private shadow arrays into the global shadow arrays will dominate the time required for the actual marking. As mentioned in Section 5.2, this can be avoided by using private hash tables of size $O(na/p)$ instead of the private shadow arrays. The hash tables can be transferred to the global shadow arrays in $O(na/p + \log p)$ time, and the check needed to avoid marking both a read and a write in the same iteration remains a constant time operation (although slightly more expensive). When using hash tables, the pairwise merges may become slightly more expensive because each processor will be responsible for $O(na/p)$ accesses in each of $\log p$ merging phases, i.e., the total cost of the pairwise merges using hash tables is $O(na \log p/p)$. If the pairwise merge for hash tables is eliminated by placing the accesses to the global shadow structure in critical sections (Section 6.2), then the complexity becomes $O(na/p + \log p)$, where the $\log p$ contribution comes from the possible contention in the critical sections.

The counting in Step 2(a) can be done in parallel by giving each processor s/p values to add within its private memory, and then summing the p resulting values in global storage, which takes $O(s/p + \log p)$ time [18]. The comparisons in Step 2(b) (2(d)) of the A_w and A_r (A_{np}) shadow arrays take $O(s/p + \log p)$ time. Again, if $s \gg na$, then the complexity can be reduced to $O(na/p + \log p)$ by using hash tables.

From the above analysis, we conclude that the DOALL tests require little communication and should scale well with all of the parameters, i.e., the number of processors, the size of the shared variable, and the number of references to the shared variable (encompassing both the number of iterations, and the number of accesses within an iteration).

7.1 Complexity of run–time privatization

If the Privatizing DOALL test determines that privatization is needed, then we either privatize the entire shared array A, or we privatize only the elements of the shared array that are *written* during the loop. If the entire array is privatized, then the allocation can be done in constant time. When privatizing only the elements that are written, the information needed is $tm(A)$, the number of elements written, and the prefix sums of A_w. Since A_w and $tm(A)$ are computed during the test itself, the only additional information needed is the prefix sums, which can be computed in time $O(s/p + \log p)$ by recursive doubling [18]. In fact, the prefix sums can be computed at the same time that $tm(A)$ is accumulated without much extra work. A similar computation can be performed on the array A_{mw} if only the elements that are written more than once are privatized (as discussed in Section 6.2). If the entire array is not privatized, then in the case of subscripted subscripts, private copies of the subscript arrays are also created. Given the original subscript array and A_w, each private subscript array can be created in time $O(m)$, where m is the size of the original subscript array.

If a private variable is live after the loop terminates, then we also need to perform a last value assignment. In this case, we keep time–stamps (iteration numbers) with the private variables. As mentioned in Section 5, if the shadow structures are "marked" with the iteration number, then the marks can double as the time–stamps. Then, after loop termination, the private variable with the latest time stamp is copied to the original (non–privatized) version of the variable. The private variable with the latest time–stamp can be selected from pairwise merges of the private A_w shadow structures in time $O(s + \log p)$; when using hash tables the complexity is $O(na \log p/p)$, or $O(na/p + \log p)$ if accesses to the global shadow structures are placed in critical sections.

7.2 Schedule reuse

Thus far, we have assumed that a DOALL test is run *each* time a loop is executed in order to determine if the loop is parallel. However, if the loop is executed again, with the same data access pattern, the first test can be reused amortizing the cost of the test 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.* [29]). 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. We remark that most programs are of a repetitive nature, and thus there exists the potential for schedule reuse. A simple example in which schedule reuse could be considered is for multiply nested loops. If possible, it is generally best to parallelize the outer loop in the nesting. However, if this is not possible, then it may be the case that schedule reuse could be attempted when parallelizing the inner loops.

8 Verifying Parallelized Loops

We have presented the DOALL and the Privatizing DOALL tests as run–time techniques that can be used to detect the parallelism of a loop before executing it. As mentioned before, another important application for these tests is in the area of debugging parallel programs to detect *access anomalies* or *race conditions* i.e., when the same memory location is accessed by more than one concurrent thread without synchronization, and it is written in at least one of the threads. In fact, the DOALL test can be used as an efficient run–time test for cases in which there are no synchronization operations between parallel loop iterations. When used for this purpose, the marking phase could be run in parallel by incorporating it in the body of the parallel loop, and the analysis phase could be done after the completion of the loop. It is important to note that since the DOALL test itself is fully parallel, the outcome of the test will be valid for any loop, regardless of its data dependence structure.

Generally, access anomaly detection techniques seek to identify the point in the parallel execution in which the access anomaly occurred. Padua *et al.* [2, 14] discuss methods that statically analyze the source program, and methods that analyze an execution trace of the program. Since not all anomalies can be detected statically, and execution traces can require prohibitive amounts of memory, *run–time* access anomaly detection methods that minimize memory requirements are desirable [30, 12, 23]. In fact, a run–time anomaly detection method proposed by Snir, and optimized by Schonberg [30], bears similarities to the version of the DOALL test presented in Section 3 (i.e., the version without the privatization). However, in order to identify the point in the execution in which the anomaly occurred, their methods [30] require much more memory than the DOALL test, e.g., viewed in the framework of the DOALL test, a separate shadow array for each *iteration* in a loop must be maintained. Another difference is that the DOALL test is optimized especially for loops, and access anomaly detection methods must handle any type of concurrent thread in a parallel program.

There are other situations in which it may be useful to have a run–time test that can determine, for a particular input set, whether or not a loop has been validly parallelized. For example, such a test can be used to *check manually parallelized* loops. Writing and especially debugging parallel programs is a very complex task, and there are many cases in which it may be difficult to verify a program's correctness by only analyzing its output. For example, in loops where access to a variable is guarded by a branch condition the DOALL test can be used to verify that no illegal concurrent accesses to that variable are made. In fact, using the DOALL test, we have discovered an instance in which the programmer incorrectly identified a loop as a DOALL for this reason.

8.1 Speculative parallel execution

Thus far, we have advocated the DOALL test as a method which should be used at run–time to determine whether a loop should be be executed sequentially or in parallel. There are, however, certain circumstances under which it may be preferable to go even further and actually *execute the loop in parallel, and determine later if, for the given input, the loop was in fact fully parallel*. If it was not, the loop is re-executed sequentially. One example of such a situation is when it is known (from, e.g., static analysis, run–time statistics, or compiler directives) that the loop is *usually* fully parallel. Another case is when the work required to extract the data access pattern (for study by the tests) is comparable to the work performed by the loop itself; in this scenario, not too much extra work (besides the test itself) is performed, and, at the conclusion, both the status of the pass/fail test and the results of the, possibly invalid, parallel execution are known. Speculative use of the tests would be implemented in essentially the same way as discussed in Section 8. In addition, the prior state of any variables modified by the parallel execution must be available for the sequential re-execution of the loop which will be required if the test fails. One alternative to saving/restoring the state of these variables is to privatize them, and to copy–in any needed external values. Then, if the test passes, only the

usual copy–out of the live variables is necessary. In any case, the cost of the solution adopted for this problem must be factored into the decision of whether or not to use speculative execution. Another issue that must be dealt with is exception handling. A simple solution is to abandon the parallel execution if an exception occurs, and execute the loop sequentially.

As a final remark, we note a method that can be used to minimize the risks of speculative parallel execution: one processor executes the loop sequentially, and the rest of the processors, speculatively, execute the loop in parallel. Of course, the sequential and the parallel executions would need separate copies of the output data for the loop. As long as the cost of creating these copies is not too great, this technique should maximize the potential gains attainable from parallel execution, while, at the same time, minimizing the costs incurred by failed tests, i.e., from testing loops that are, in fact, not parallel.

9 Automatic Application of the Tests

In the previous sections we have discussed run–time data dependence and privatization techniques. These techniques are automatable and a good compiler could easily insert them in the original code. In this section, we address some of the issues that are involved with the automatic utilization of these tests. We begin with a brief outline of how a compiler might apply the DOALL test.

1. *At Compile Time.*
   (a) Generate an *inspector loop* for the marking phase of the test. This is done by collecting all accesses to the shared variables under study into a separate loop, where they are replaced by accesses to the appropriate shadow variables. An example is shown in Fig. 5, where there are only output dependences.
   (b) A cost/performance analysis determines whether the test should be applied. If not, the inspector is discarded and a sequential version of the loop is generated.
   (c) Generate a multi-version loop with options for sequential and parallel executions of the original loop. The run–time selection among the versions of the loop depends upon the outcome of the analysis of the shadow variables marked by the inspector loop. The analysis can be done by calls to a run–time library.
2. *At Run–Time.*
   (a) Execute the marking phase of the test, i.e., run the inspector generated in Step 1(a).
   (b) Execute the analysis phase of the test, which gives the pass/fail result of the test.
   (c) Execute the selected version of the loop indicated by the result of Step 2(b).
   (d) Collect statistics for use in future runs, and/or for schedule reuse in this run.

Generating the inspector loop

An inspector loop can be formed from the original loop by replacing accesses to the shared variables with accesses to the appropriate shadow variables; also, to avoid side effects, all other program variables written in the loop must be privatized. However, to make the inspector to run as fast as possible, we want to exclude any computation from the original loop that does not affect the pattern of access to the variables under study. In the best case, the access pattern of the shared variable is known before entering the loop, e.g., a pre-determined subscript array. In this simple case, the access pattern of the array can be processed in complete isolation from the rest of the loop. However, there are cases in which it is not possible for the data access operations to be completely isolated from the other computation in the loop. Specifically, if the address (subscript) computation of an array element is performed in the same π-block (strongly connected component in the dependence flow graph) as the statement that references the array using that subscript, then the address computation cannot be removed from the inspector loop. For example, the access pattern may be computed inside the loop itself, perhaps in a preparation phase, e.g., a loop first collects in a subscript array the indices of the elements of a much larger data structure that will be processed subsequently in the loop. In this case, the inspector loop must include the access pattern computation. Another difficult case is when the statements accessing the array in question are control flow dependent upon data computation performed inside the loop (with the exception of loop indices and loop invariant variables). In this case, the inspector must either execute the computation that defines the predicates, or assume conservatively that all branches are taken (possibly introducing false dependences). In order

to minimize such situations, the compiler should reduce the control flow affecting data accesses as much as possible by using known techniques such as *loop distribution*, *loop invariant hoisting*, and, generally, *code motion*.

Determining when to apply the test

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of the tests will be enhanced if their run–time overhead is avoided when the test is likely to fail. There are three main factors that the compiler should consider when deciding whether or not to apply the test: the probability that test will pass (i.e., that the loop is in fact a DOALL), the speedup that would be obtained if the test passes, and the slowdown incurred if the loop is not a DOALL.

In order to predict the outcome of the test, the compiler should use both static analysis and run–time statistics (collected on previous executions of the loop). In addition, directives about the parallelism of the loop might prove useful.

Given a fully parallel loop L, the *ideal speedup*, Sp_{id}, is the ratio between its sequential and its parallel execution times, T_{seq} and T_{doall}, respectively. However, when L is parallelized using the DOALL test, the *attainable speedup*, Sp, must account for the overhead required by the marking and analysis phases of the test, T_{mark} and $T_{analysis}$, respectively.

$$Sp_{id} = \frac{T_{seq}}{T_{doall}} \qquad\qquad Sp = \frac{T_{seq}}{T_{mark} + T_{analysis} + T_{doall}}$$

Using static analysis, the compiler can compute an estimate for Sp by estimating $T_{seq}, T_{doall}, T_{mark}$ and $T_{analysis}$. The values T_{seq} and T_{doall} can be estimated using some architectural model, e.g., instruction counting. Our analysis in Section 7 predicts that $T_{mark} = T_{analysis} = O(na/p + \log p)$, where p is the number of processors, n is the number of iterations of the loop, and a is the maximum number of accesses to the shared array in a single iteration of the loop. In practice, $T_{analysis}$ should be fairly well modeled by this expression, i.e., $T_{analysis} \approx c(na/p + \log p)$, where c is some small constant. However, the estimate of T_{mark} may not always be as good. The reason for this is that our analysis implicitly assumed that the data access pattern is known before loop entry. As discussed above, in the worst case $T_{mark} \approx T_{doall}$, i.e., the inspector loop is computationally equivalent to the original loop. However, in all cases, an estimate of T_{mark} can be obtained by static analysis of the inspector loop.

Note that in the worst possible case $T_{mark} \approx T_{analysis} \approx T_{doall}$, and even then the attainable speedup predicted for the DOALL test is $\approx \frac{1}{3} Sp_{id}$. Although 33% of ideal speedup may not appear impressive on an eight processor machine, these tests were designed for massively parallel processors (MPPs), and on such a machine this in an excellent performance when compared to the alternative of sequential execution.

It is also instructive to examine the *slowdown* incurred by a failed test, i.e., when the loop must be executed sequentially. In this case, T_{seq} is increased by $T_{mark} + T_{analysis}$. Note that since the DOALL test is fully parallel, in the worst case we have $T_{mark} \approx T_{analysis} \approx \frac{1}{p} T_{seq}$, i.e., when the marking phase is work-equivalent to the loop. Thus, the cost of performing a failed test is proportional to $\frac{1}{p} T_{seq}$:

$$T_{seq} + T_{mark} + T_{analysis} \approx T_{seq} + \frac{2}{p} T_{seq} = (1 + \frac{2}{p}) T_{seq}$$

Therefore, unless it is known a priori with a high degree of confidence that the loop is not parallel, the test should probably be applied, i.e., the potential payoff is worth the risk of slightly increasing the sequential execution time.

Based on the outcome of the cost/performance analysis, the compiler determines whether the test should be performed, and if it decides to use the test, it must also decide how the test should be applied: using the inspector/executor paradigm (i.e., first test, and then execute) or in a speculative manner (i.e., test and execute simultaneously, as described in Section 8.1). In practice, when T_{mark} approaches T_{doall}, speculative use of the tests may be beneficial. The overhead needed to save/restore state (Section 8.1), must also be considered when deciding whether to use speculative parallel execution.

10 Previous Run–Time Techniques for Parallelizing Loops

As mentioned in Section 1, there has been some work on developing techniques for the run–time analysis and scheduling of loops with cross–iteration dependences. Note that this is a more general problem than the one studied in this paper since the construction of a valid parallel execution schedule requires *finding and analyzing all* cross–iteration dependences in the loop whereas we have been concerned with simply *detecting the presence of any* dependences that prohibit parallelization. Therefore, these techniques are necessarily more complex than the PD test, and in almost all cases use global synchronization primitives, make conservative assumptions regarding cross–iteration dependences, have significant sequential components, and/or do not find an optimal parallel execution schedule for the iterations of the loop. Most of these schemes partition the set of iterations into subsets called *stages*, so that the iterations in each stage can be executed in parallel, i.e., there are no data dependences between iterations in a stage. Stages formed by a regular pattern of iterations are named a wavefronts. The wavefronts themselves are executed sequentially by placing a synchronization barrier between each wavefront.

One of the first run–time methods for scheduling partially parallel loops was proposed by Zhu and Yew [37]. It computes the stages one after the other in successive phases. In a phase, an iteration is added to the current stage if it is the lowest iteration (not assigned to a previous stage) that accesses (reads or writes) any of the data (variables) used in that iteration, i.e., none of the data accessed in that iteration is accessed by any lower unassigned iteration. In each phase, the lowest unassigned iteration to access any variable (e.g., array element) is found using atomic *compare-and-swap* synchronization primitives to record the minimum such iteration in a shadow version of that variable. By using separate shadow variables to process the read and write operations, Midkiff and Padua [22] improved this basic method so that concurrent reads from a memory location are allowed in multiple iterations. Recently, Chen, Yew and Torrellas [11] proposed another variant of the Zhu and Yew method which improves performance in the presence of *hot-spots* (i.e., many accesses to the same memory location) by first doing some of the computation in private storage. All of the above mentioned methods construct maximal stages in the sense that each iteration is placed in the earliest possible stage, giving a minimal depth schedule, i.e., a minimal number of stages.

Polychronopolous [25] gives a method that assigns iterations to stages in a different way: each wavefront consists of a maximal set of contiguous iterations which contain no cross-iteration dependences. It is easy to see that this method may not yield a minimum depth schedule. Like the method of Zhu and Yew, and its variants, this method uses shadow versions of the variables to detect possible dependences. The wavefronts can be constructed sequentially by inspecting all the shared variable accesses, or in parallel with the aid of critical sections. Note that since all computations are performed at run–time, it is important for them to be efficiently parallelizable.

Krothapalli and Sadayappan [16] proposed a run–time scheme for removing anti (write-after-read) and output (write-after-write) dependences from loops. These types of dependences are also known as *memory–related* dependences because they arise from the reuse of storage and are not essential to the computation, as are flow (read-after-write) dependences which express a fundamental relationship about data flow in the program. Their scheme includes a parallel preprocessing phase which uses critical sections as in the method of Zhu and Yew, to determine the number and types of accesses to each memory location. Next, for each memory location they build a flow graph, allocate any additional storage needed to remove the anti and output dependences, and explicitly construct the mapping between all the memory accesses in the loop and the storage (both old and new). Then, the loop is executed in parallel using synchronization (locks) to enforce the flow dependences. This scheme relies heavily on synchronization, inserts an additional level of indirection into all memory accesses, and calls for dynamic shared memory allocation.

The problem of analyzing and scheduling loops at run–time has been studied extensively by Saltz *et al.* [7, 27, 28, 29, 36]. In most of these methods, the original source loop is transformed into an *inspector*, which performs some run–time data dependence analysis and constructs a (preliminary) schedule, and an *executor*, which performs the scheduled work. The original source loop is *assumed to have no output dependences.* In [29], the inspector constructs stages that respect the flow dependences by performing a *sequential* topological sort of the accesses in the loop. The executer enforces any anti dependences by using old and new versions of each variable. Note that the anti dependences can only be handled in this way because the original loop does not have any output dependences, i.e., each variable is written at most once in the loop. The inspector computation (the topological sort) can be parallelized somewhat using the *DOACROSS parallelization technique* of Saltz and Mirchandaney [27], in which processors are assigned iterations in a wrapped manner, and busy-waits are used to ensure that values have been produced before they are used (again, this is only possible if the original loop has no output dependences).

Benchmark Subroutine Loop	Experimental Results				Data Access Description	Inspector
	Technique	Speedup obtained	Speedup ideal	#Proc		
OCEAN FTRVMT Loop 109	insp/exec	2.1	6.0	8	kernel-like loop accesses a vector with run–time determined strides	data accesses replicates loop
MDG INTERF Loop 1000	insp/exec private	8.8	12.4	14	accesses to a privatizable vector guarded by loop computed predicates	data accesses branch predicates
BDNA ACTFOR Loop 240	insp/exec private	7.6	11.6	14	accesses a privatizable array indexed by a subscript array computed inside loop	data accesses subscript array
TRFD INTGRL Loop 140	insp/exec sch. reuse	1.5 2.2	4.6	8	small triangular loop accesses a vector indexed by a subscript array computed outside loop	data accesses replicates loop
TRACK NLFILT Loop 300	speculative	4.2	4.4	8	accesses an array indexed by subscript array computed out of loop, access pattern guarded by loop computed predicates	not applicable

Table 1: Summary of Experimental Results

Recently, Leung and Zahorjan [19] have proposed some other methods of parallelizing the inspector of Saltz *et al.* These techniques are also restricted to loops with no output dependences. In *sectioning*, each processor computes an optimal parallel schedule for a contiguous set of iterations, and then the stages are concatenated together in the appropriate order. Thus sectioning will usually produce a suboptimal schedule since a new synchronization barrier is introduced into the schedule for each processor. In *bootstrapping*, the inspector of Saltz *et al.* (i.e., the sequential topological sort) is parallelized using the sectioning method. Although bootstrapping might not optimally parallelize the inspector (due to the synchronization barriers introduced for each processor), it will produce the same minimum depth schedule as the sequential inspector of Saltz *et al.*

11 Experimental Results

In this section we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [3]) and 14 processors (Alliant FX/2800 [4]) using a Cedar Fortran [15] implementation of the DOALL tests. We considered five loops contained in the PERFECT Benchmarks [6] that could not be parallelized by any compiler available to us. A summary of our results is given in Table 1. For each loop, the methods applied and the speedup obtained are reported. As a reference, we give the ideal speedup, which was measured using an optimally parallelized (by hand) version of the loop. In addition, we mention the computation performed by the inspector; the notations *branch predicates* and *subscript array* mean that the inspector computed these values, and *replicates loop* means that the inspector was work-equivalent to the original loop.

In addition to the summary of results given in Table 1, we show in Figures 8 through 6 the speedup measured for each loop as a function of the number of processors used. These graphs show that in most cases the speedups scale with the number of processors and are a significant percentage of the ideal speedup. Recall that in order for our methods to scale with the number of processors and the data size, the shadow arrays should be distributed over the processor space, rather than replicated on each processor. As discussed in Section 5, one way to accomplish this is to use hash tables for the shadow structures. However, since our implementation does not yet optimize with hash tables, in some cases the speedups shown do not appear to scale. In particular, in cases in which the size of the array under test is $> na/p$, our implementation displays $T_{mark} \approx T_{analysis} \approx s$ instead of $\approx na/p$ as predicted, (i.e., the marking and analysis phases of the test touch the entire shadow array, regardless of the access pattern in the loop under study). This situation is encountered with the loops from Ocean and TRFD. We believe that the use of hash tables (for MPPs) would preserve the scalability of our methods in these cases as well.

For all of the loops studied we have obtained a very good fit of our experimental data to the speedups predicted by the modeling described in Section 9. Our estimates were made using a simple instruction counting model: for loops (vectors), we used the product of the number of instructions and the number of iterations (elements). In the following,

we discuss each loop in more detail and discuss the accuracy of our performance prediction modeling.

OCEAN–FTRVMT–Loop 109. This loop is utilized in the computation of a 2-dimensional FFT. It is invoked 26,000 times, and accounts for 40% of the sequential execution time of the program. For this loop we estimate $T_{mark} \approx T_{analysis} \approx T_{doall}$, and predict $Sp \approx \frac{T_{seq}}{3T_{doall}} = \frac{1}{3}Sp_{id}$. In fact, as shown in Fig. 8, the speedup obtained by the DOALL test is about 1/3 of the ideal speedup.

MDG–INTERF–Loop 1000. This loop calculates inter-molecular interaction forces. In order to avoid false dependences, our inspector computes the branch predicates and has an estimated complexity $T_{mark} \approx .2T_{doall}$. Since $T_{analysis}$ is small (the shadow vector has only 14 elements), we expect the Privatizing DOALL test to obtain $Sp \approx .7Sp_{id}$. The results shown in Fig. 9 display a pretty good fit to this estimation.

BDNA–ACTFOR–Loop 240. This loop selects certain elements from a large array, and processes the selected elements later in the loop. The speedup obtained for this loop using the Privatizing DOALL test is almost 2/3 of the ideal speedup (see Fig. 7). This speedup cannot be accurately predicted at compile time because the number of selected elements is not known until run–time, and the inspector executes the selection phase (computing the subscripts), but not the subsequent processing phase.

TRFD–INTGRL–Loop 140. For this small triangular loop, $Sp_{id} \approx 4.5$ on the Alliant FX/80 because of load imbalance. Statically, we would predict $T_{doall} \approx T_{mark} \approx T_{analysis}$, and $Sp \approx \frac{1}{3}Sp_{id}$. However, since our implementation does not yet optimize with hash tables, the access pattern of our shadow arrays is rectangular (versus the triangular pattern of the loop) and $T_{mark} \approx T_{analysis} \approx 2T_{doall}$, so that $Sp \approx \frac{1}{5}Sp_{id} < 1$.

However, Loop 140 is executed seven times (within an outer loop) and uses a larger subscript array on each subsequent execution. We could use *schedule reuse* (Section 7.2) if the current subscript array were a subset of the one from the previous invocation. In order to obtain this subset relation, we executed the calling loop in reverse order – and verified our action with the DOALL test. For the seven invocations of Loop 140, we obtained $Sp = \frac{1}{2}Sp_{id}$ – including the subscript array comparisons (see Fig. 10). The example shows that in the most disadvantageous cases we can obtain significant speedups by using a simple schedule reuse technique.

TRACK–NLFILT–Loop 300. Since the inspector would have had to perform almost all of the computation of the loop, we decided to use *speculative execution* as described in Section 8.1. Of the 59 executions of the loop, only five times it was not a DOALL. In these cases we restored the values of the variables modified by the loop (which were saved before invoking the loop) and re-executed the loop sequentially. The results are depicted in Fig. 6. Note that the speedup reported here includes both the parallel and the sequential instantiations. Of course, such a speculative technique can be costly if the test fails most of the time, but in cases such as this where extracting the access pattern is not possible without executing the whole loop, it is an attractive alternative.

12 Conclusion

In this paper we approach the problem of determining the parallelism of loops at run–time from a different viewpoint – instead of finding a valid parallel execution schedule for a loop, we solve the problem of simply deciding if the loop is fully parallel – a frequent occurrence in real programs. Our methods are also capable of eliminating some memory–related dependences by dynamically privatizing scalars and arrays.

We have shown that the concept of run–time data dependence checking is a useful solution for loops that cannot be sufficiently analyzed by a compiler. We would like to re-emphasize that our methods are applicable to all loops, without any restrictions on their data or control flow. Both inspector/executor and speculative strategies have been shown to be viable alternatives for even modestly parallel machines like the Alliant FX/80 and 2800. However, we believe that the true significance of these methods will be the increase in real speedup obtainable on massively parallel processors (MPPs). As we have shown, the cost associated with the DOALL test is proportional to $\frac{1}{p}T_{seq} + \log p$, where p is the number of processors available. If the target architecture is an MPP with *hundreds* or, in the future *thousands*, of processors, then this cost will become a very small fraction of sequential execution time (T_{seq}). When applying the DOALL test to a loop, our performance gain/loss will range from at least $1/3$ of the ideal speedup when the test passes (which can reach into the hundreds for MPPs), to an additional few percentage points of the sequential execution time

if the test fails. In other words, the test has the potential to offer large gains in performance (speedup), while at the same time risking only small losses. To bias the results even more in our favor, the decision on when to apply the test should make use of run–time collected information about the fully parallel/not parallel nature of the loop. In addition, specialized hardware features could greatly reduce the overhead introduced by the test.

In the near future we plan to implement these methods in POLARIS, a restructuring compiler currently being developed at the University of Illinois that targets the latest generation of massively parallel architectures.

Acknowledgement

We would like to thank William Blume for identifying applications for our test and other useful advice, and Paul Peterson for many fruitful discussions. We would especially like to thank Nancy Amato for carefully reviewing the manuscript.

References

[1] S. Abraham. Private communication, 1994.

[2] T. Allen and D. A. Padua. Debugging fortran on a shared-memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, St. Charles, IL, 1987.

[3] Alliant Computer Systems Corporation, 42 Nagog Park, Acton, Massachusetts 01720. *FX/Series Architecture Manual*, 1986. Part Number: 300-00001-B.

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

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] D. K. Chen, P. C. Yew, and J. Torrellas. An efficient algorithm for the run-time parallelization of doacross loops. manuscript, 1994.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proc. ACM ???*, pages 1–10, 1990.

[13] 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.

[14] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Soft.*, pages 69–77, January 1992.

[15] 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.

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

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

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

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

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

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

[23] I. Nudler and L. Rudolph. Tools for the efficient developement of efficient parallel programs. In *Proc. 1st Israeli Conference on Computer System Engineering*, 1988.

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

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

[26] 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 –, July 1994.

[27] 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.

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

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

[30] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 285–297, Portland, Oregon, 1989.

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

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

[33] 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.

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

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

[36] 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.

[37] 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.

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

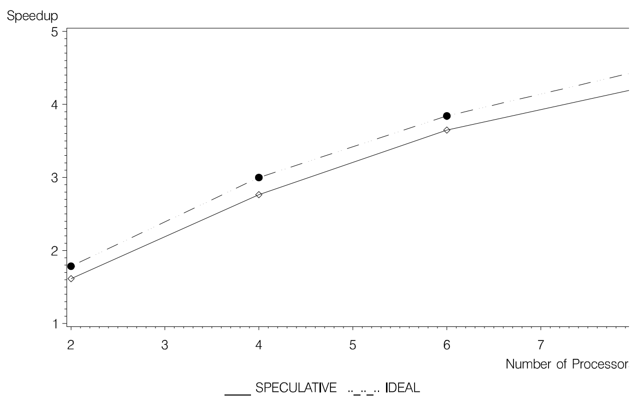

Figure 6:

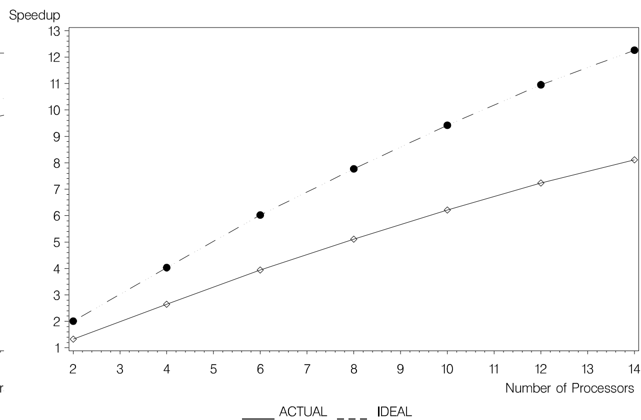

Figure 7:

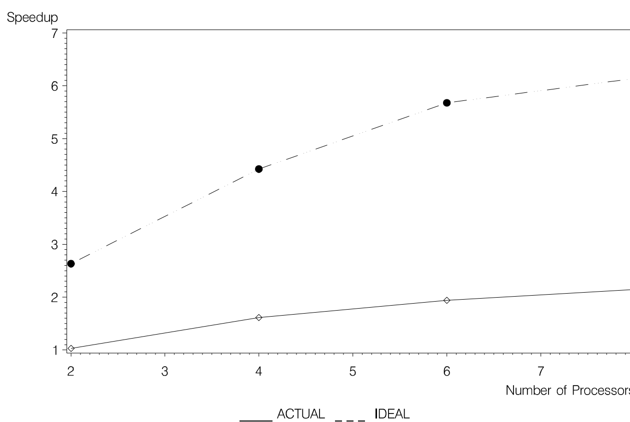

Figure 8:

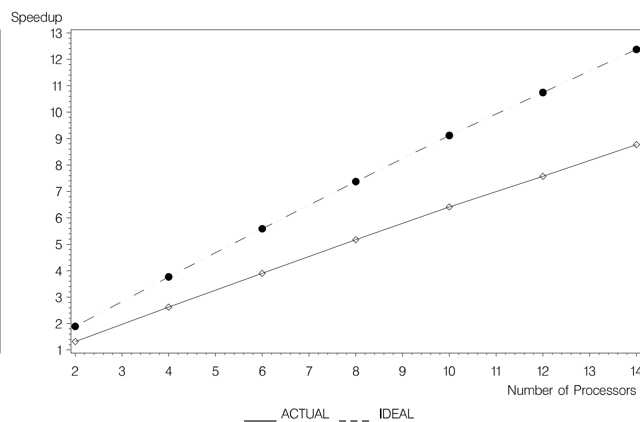

Figure 9:

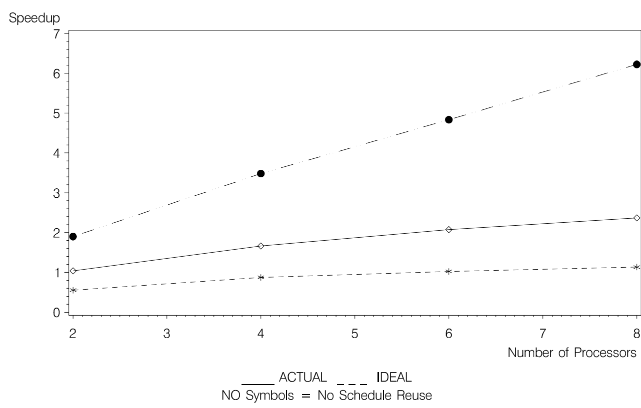

Figure 10: