An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops[1]

Ding-Kai Chen[2] Josep Torrellas Pen-Chung Yew
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801

Abstract

While automatic parallelization of loops usually relies on compile-time analysis of data dependences, for some loops the data dependences cannot be determined at compile time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, it is necessary to perform run-time analysis. In this paper, we present a new algorithm to parallelize these loops at run time. Our scheme handles any type of data dependence in the loop without requiring any special architectural support in the multiprocessor. Furthermore, compared to an older scheme with the same generality, our scheme significantly reduces the amount of processor communication required and increases the overlap among dependent iterations. We evaluate our algorithm with parameterized loops running on the 32-processor Cedar shared-memory multiprocessor. The results show speedups over the serial code of up to 14 with the full overhead of run-time analysis and of up to 27 if part of the analysis is reused across loop invocations. Moreover, the algorithm outperforms the older scheme in nearly all cases, reaching speedups of up to times when the loop has many dependences.

1 Introduction

Loop-level parallelism is an important source of speedup in multiprocessors that exploit medium-grain parallelism. To parallelize a loop, it is necessary to identify the data dependence relations between loop iterations via dependence analysis [1]. If the loop does not have any dependences across iterations, then it can run in parallel. Otherwise, we have to run it serially or, if we want to exploit some parallelism, insert proper synchronization to ensure that the memory accesses are performed in the correct sequence. The latter type of loop is called a DOACROSS loop.

Some DOACROSS loops can be transformed into fully parallel loops. For example, if the difference in iteration number between dependent iterations (called *dependence distance*) is known at compile time, we can use techniques like loop-alignment, loop skewing, or dependence uniformization [8, 13, 3, 11]. Then the loop can be run in parallel. Unfortunately, many times it is not possible to determine the dependence distance at compile time. An example is when array elements are accessed via subscripted subscripts. In these cases, the compiler cannot prove the independence of iterations. Therefore, even if the loop turns out to be very parallel, compile-time schemes fail to parallelize the loop.

For these loops we can still perform the dependence analysis at run-time and maybe execute the loop in parallel. This approach is called *run-time parallelization*. Much previous research has been done to design effective run-time parallelization algorithms [4, 6, 7, 10, 14]The main differences among the schemes proposed are the types of dependence patterns that are handled and the required system or architecture support. The key to success in these schemes is to minimize the time spent on the analysis of dependences and on process synchronization. Indeed, if too much time is spent on these overheads, it is better to run the loop serially.

In this paper, we describe and evaluate a new algorithm for the run-time parallelization of DOACROSS loops. Our scheme handles any type of dependence pattern without requiring any special architectural support. Furthermore, compared to an older scheme with the same generality [14], it speeds up execution by significantly reducing the amount of communication required and by increasing the overlap among dependent iterations. The effectiveness of this algorithm is evaluated via measurements of parameterized loops in the 32-processor Cedar shared-memory multiprocessor [5]. The results show speedups that reach up to 14 with the full overhead of the analysis and up to 27 if part of the analysis work is reused across loop invocations. Moreover, our algorithm outperforms the older scheme with the same generality in nearly all cases, reaching a 37-fold speedup when the loop has many dependences.

We organize this paper as follows. Section 2 provides background on run-time parallelization; Section 3 explains our scheme in detail; Section 4 describes our experimental methodology; and Section 4.2 presents the performance results. Section 5 concludes our study.

2 Background and Run-Time Parallelization Issues

In this section, we first present the basic concepts and then the previous work.

2.1 Loop Model and Basic Concept

Consider the loop shown in Figure 1(a). In order to determine if there is any loop-carried dependence [12] between statements S_p and S_q and, if so, compute its dependence distance, we have to solve the following integer equation

$$f(i) = g(i')$$

and evaluate the distance $(i' - i)$ within the constraints of the loop limits. Depending on the format of $f(i)$ and $g(i)$, there are several possible parallelization techniques to use (see Table 1). In the first and the second rows of Table 1, $f(i)$ and $g(i)$ are linear functions of the loop index with constant coefficients. In these cases, the dependence

[1] This work was supported in part by the National Science Foundation under grants NSF MIP-8920891, NSF MIP-9307910, NSF RIA MIP-9308098; the U.S. Department of Energy under grant DOE DE-FG02-85ER25001; a NASA ICLASS grant; and grant 1-1-28028 from the Univ. of Illinois Research Board.

[2] Currently at Silicon Graphics Computer Systems, Mountain View, CA 94043. E-mail: dkchen@mti.sgi.com

Dependence Type	Example		Method to Parallelize The Loop
	$f(i)$	$g(i)$	
Uniform Distance	2i+3	2i+1	Loop Alignment [8], Loop Skewing [13] (if loop level> 1)
Non-Uniform Distance	2i+9	3i+3	Dependence Uniformization [11, 3] (if loop level> 1)
Unknown Distance	I1(i)	I2(i)	Run-Time Parallelization

Table 1: Sample dependence types and parallelization methods.

distances are known or can be estimated at compile-time, and parallelizing compilers can generate proper synchronization instructions to preserve dependences. However, in the third row of Table 1, the dependence distance information is not available at compile-time: $f(i)$ and $g(i)$ are array expressions whose values are not known until run-time because they depend on the input data. Therefore, to parallelize this loop, we need to use a run-time parallelization scheme.

```
DO i=1,N                      DO i=1,N
   ...                           ...
Sp:A(f(i)) = ...              Sp:A(I1(i)) = ...
   ...                           ...
Sq:... =A(g(i)) + ...         Sq:...  =A(I2(i)) + ...
   ...                           ...
ENDDO                         ENDDO
         (a)                           (b)
```

Figure 1: Loop Example.

Without loss of generality, we use that loop in Figure 1(b) as our main example, where we assume the values of arrays *I1* and *I2* are not available until run-time. The values of both *I1* and *I2* have no restriction. Each array can have duplicated values, which implies that all three kinds of data dependences, namely flow (Read-after-Write), anti (Write-after-Read), and output (Write-after-Write) dependences, could occur between instances of statements S_p and S_q. For example, if $I1(1) = I2(3) = I1(4) = I1(5) = x$ in Figure 1(b), then we have the following dependences: $S_p(1) \xrightarrow{flow} S_q(3) \xrightarrow{anti} S_p(4) \xrightarrow{output} S_p(5)$. These accesses form a *dependence (or access) chain*. The values of arrays *I1* and *I2* are assumed to remain constant during the execution of one invocation of the loop, although they are allowed to change outside the loop.

In general, run-time parallelization schemes have two stages, namely *inspector* and *executor* [10, 14]. Both are performed at runtime. The inspector determines the dependence relations among the data accesses. The executor uses this information to execute the iterations in parallel in an order that preserves the dependences. If both the inspector and the executor are parallel algorithms, the scheme can take full advantage of parallel machines. The key to the success of these schemes, however, is to reduce the communication overhead in these two stages. Finally, note that, since index arrays for a loop such as *I1* and *I2* do not usually change between invocations of the loop, the inspector results could be reused across loop invocations.

2.2 Previous Work

Several run-time parallelization schemes have been proposed. The first one was proposed by Zhu and Yew [14]. Their scheme is general enough to handle any dependence pattern. Given a loop like that in Figure 1(b), the compiler transforms it into the code shown in Figure 2. In the figure, each element of array A has two fields, *data*, which contains the data value, and *key*, which is used to order the

```
Done(1:N)= .FALSE.
REPEAT UNTIL ((Done(i) .EQ. .TRUE) for all i)
C Inspector Phase
 DOALL i=1,N
  A(I1(i)).key= ∞
  A(I2(i)).key= ∞
 END DOALL
 DOALL i=1,N
  IF (Done(i) .EQ. .FALSE)
C the next two instructions are atomic
   compare&store{ if (A(I1(i)).key>i)
    { A(I1(i)).key ← i; } }
   compare&store{ if (A(I2(i)).key>i)
    { A(I2(i)).key ← i; } }
  END IF
 END DOALL
C Executor Phase
 DOALL i=1,N
  IF (Done(i) .EQ. .FALSE)
   IF ((A(I1(i)).key .EQ. i) .AND.
    (A(I2(i)).key .EQ. i)) THEN
    ...
    A(I1(i)).data = ...
    ...
    .=A(I2(i)).data + ...
    ...
    Done(i)=.TRUE.
   END IF
  END IF
 END DOALL
END REPEAT
```

Figure 2: Loop of Figure 1(b) transformed to use Zhu-Yew's scheme.

accesses. In this scheme, an iteration i is allowed to proceed at runtime only if all accesses to the array elements *A(I1(i))* and *A(I2(i))* by iterations $j<i$ have completed. To do so, the inspector determines the set of iterations that can proceed. This is done by having all unexecuted iterations visit the array elements they want to access and save in the *key* field of these elements their own iteration number if it is less than the value already stored in the *key* field. After that, the numbers remaining in the *key* field are the numbers of the iterations that can proceed. In the executor phase, iterations check if the *key* field of the elements that they want to access have a number equal to their own iteration number. If so, the iteration has no unexecuted predecessor and is allowed to proceed. Once iteration i is executed, we set *Done(i)* to *TRUE*. This process continues until all iterations are executed.

A good property of Zhu-Yew's algorithm is to have fully parallel inspector and executor phases. Unfortunately, it requires many memory accesses. Indeed, for each iteration of the repeat loop, each unexecuted iteration i requires at least $3r$ memory accesses, where r is the number of references to array A per iteration.

A second scheme, proposed by Midkiff and Padua [7], improved Zhu-Yew's scheme by allowing concurrent reads to the same array element by several iterations. To do so, each data element requires more than one *key* field. However, it still requires high communication as Zhu-Yew's scheme.

Other, less general schemes have been proposed [4, 9]. They restrict the problem in three ways. First, no two index elements of a given index array (*I1* and *I2* in Figure 1(b)) have the same values. This makes scheduling simpler. Second, they require a serial pre-processing loop, which can reduce speedup. Finally, if the index arrays are not one-to-one mappings, they require shared-memory allocation. Unfortunately, this can be slow and complicated in large-scale multiprocessors. How to do it is not described.

Lately, schemes for loops without output dependences have been proposed by Saltz *et al.* [10] and Leung and Zahorjan [6]. This assumption simplifies the problem. The emphasis of Saltz *et al.*'s work is on the scheduling issues. The results show that the best scheduling method is to statically partition the iterations among the processors and to rearrange the iteration order within each partition.

Finally, a common limitation of most of the previous work is the lack of performance results. It is therefore hard to evaluate the effectiveness of all these algorithms.

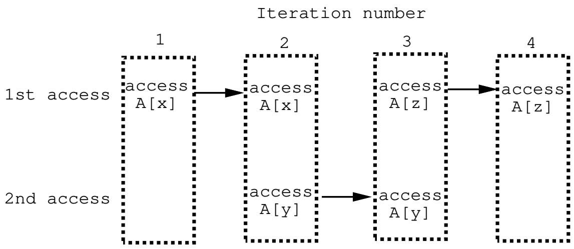


Figure 3: Example of dependence chains.

3 A New Algorithm for Run-Time Parallelization

Although Zhu-Yew's algorithm can handle all types of dependences, it has two limitations. First, the inspector cannot be reused across invocations of the same loop, even if the dependences do not change, because the inspector and the executor are tightly connected. Second, the execution of iterations with dependences cannot be overlapped. For example, assume that the dependences in the loop of Figure 1(b) are those in Figure 3. While the first access in iteration 3 does not depend on anything, Zhu-Yew's algorithm will not perform it until after the second iteration is finished, because the second access of iteration 3 depends on iteration 2. This limitation not only reduces part of the parallelism available; it also causes redundant traffic. Indeed, iteration 3 will be inspected by the inspector three times. Each time, it will generate extra traffic.

Our algorithm, instead, reuses the inspector results across loop invocations, and allows the overlap of dependent iterations. This is done by splitting the inspector and the executor. In the inspector, we gather all the dependence information and store it in a table called *Ticket*. Then, this information is used in one or more executor phases. Building this table, however, is very expensive, because it requires interprocessor communication. To reduce communication, our inspector algorithm exploits locality. The inspector and the executor phases are described next.

3.1 Inspector Phase

In this phase, we will order the references that access the same location while maintaining the original dependences. In this phase, processors share the *Ticket* table, whose x-dimension is the number of iterations in the loop and y-dimension the number of references to the shared locations per iteration. For example, the *Ticket* table for the loop of Figure 1(b) with $N = 12$ is shown in Figure 4(a). In the figure, we have numbered the accesses according to their serial execution order.

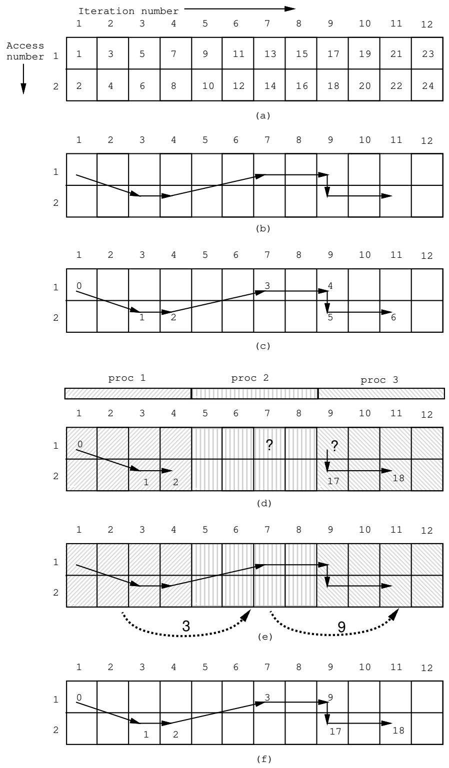


Figure 4: Example of the *Ticket* table.

To illustrate how the algorithm gathers the dependence information, assume that, for a given entry x of array A in Figure 1(b),

$$I1(1)=I2(3)=I2(4)=I1(7)=I1(9)=I2(9)=I2(11)=x.$$

This chain of dependences is represented in Figure 4(b) as a set of arrows on the *Ticket* table. The goal of the inspector is to store in the table entries sequence numbers that order the references involved in this chain of dependences from left to right. In the ideal case, the numbers assigned would be the ones shown in Figure 4(c).

In the executor phase, we will use an extra field called *key* assigned to each element of the shared data structure to enforce the ordering. For example, whatever processor is assigned in the executor phase to execute iteration 4 will wait for *A(I2(4)).key* to become 2. It will then access *A(I2(4))* and increment *A(I2(4)).key* by one. This, in turn, will trigger the execution of the first access in iteration 7, and so on. Of course, all other accesses not present in this chain of dependences can proceed in parallel if they do not belong to any other dependence chain.

To fill the table quickly, we use all processors to do it in parallel. Unfortunately, this requires some communication overhead. Our algorithm tries to minimize inter-processor communication by first filling most of the entries in the table in a communication-free phase called the *local inspector*, and then filling the few remaining ones with communication in the *global inspector* phase. We consider each phase in turn.

Local Inspector Phase

We divide the iteration space into as many groups of contiguous iterations as the number of processors, and then assign each group of iterations to a processor for inspection. Each processor, therefore, is in charge of updating a set of contiguous columns in the *Ticket* table. If we assume that we have 3 processors, the assignment is as shown in Figure 4(d).

In this phase, processors are trying to determine the dependence chain and to assign the sequence numbers. Note that they still do not know that the shape of the dependence chain is like the one shown in Figure 4(c). For each processor, the main difficulty is to know the sequence number of the first access in its local portion without communicating to other processors. Once the sequence number of the first access is known, the remaining accesses in the local portion could be determined accordingly. Note, however, that the sequence number need to be unique and in increasing order, but they do not have to be consecutive. Hence, we can leave the sequence number of the first access in each local portion open until the global inspection phase.

To allow each processor to assign sequence numbers in its local portion, we use a conservative approach. We assume that all of the accesses in all of the iterations prior to the local portion of the accesses are all in the same dependence chain. Therefore, we can come up with a starting sequence number for each local portion of the accesses. Figure 4(d) shows the result of such a scheme. Processor 1 can fill all its entries since its local starting sequence number is 0. Processor 2 has only one access in its local portion of the dependence chain. Hence, it will leave its sequence number open (shown as a question mark in Figure 4(d)). Processor 3 also has its first access open and we have to determine a safe starting sequence number for the second access of iteration 9. Since each iteration has 2 accesses, the conservative estimate for its starting sequence number will be $2 \times 8 + 1 = 17$ which is shown in Figure 4(d). Therefore, we set *Ticket(9,2)* to 17 and *Ticket(11,2)* to 18.

Global Inspector Phase

In the global inspector phase, we fill the entries that have a question mark in Figure 4(d). Each processor determines the sequence number that it will generate after executing its last access in the local dependence chain, and then sends it to the processor on its right. We call this number the *transition sequence number*. For example, processor 1 needs to pass 3 to processor 2 (Figure 4(e)). Processor 3 would pass 19 if there were more processors. For processor 2, its group of iterations has only one access in the chain, which has a question mark. In this case, as indicated in the local inspector phase, we assume that all accesses in processor 1 belong to the dependence chain. Therefore, the safe value for the access in *Ticket(7,1)* to set the *key* to is 9, which will be passed to processor 3 (Figure 4(e)).

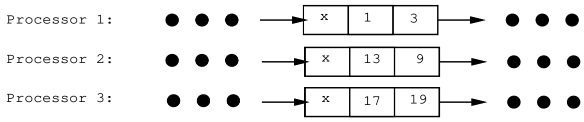


Figure 6: Unresolved lists for Figure 4.

To communicate these transition sequence numbers we use a fast parallel algorithm that can best be described using the *compare&store* atomic instruction, which in turn, can be implemented with primitives as simple as *lock*. We use a shared variable *Done*, and two shared arrays *Temp1* and *Temp2* which have as many entries as in the original shared data structure (array A in Figure 1(b)). In addition, each processor has a linked list called the *unresolved list* with nodes corresponding to its *Ticket* table entries that have question marks. Each node in the list has three fields. The first field is the index of the shared data structure element that is being accessed (x in our example). This is necessary because a processor will have one of these nodes per dependence chain it participates in. The second field identifies which entry in the *Ticket* table has the question mark (entry 13 for processor 2 in Figure 4(d)). The last field is the transition sequence number. The *unresolved lists* for the processors in our example are shown in Figure 6. In the figure, processor 1 has a node for the first access of the dependence chain (entry 1), even though we know its sequence number, namely 0. This is necessary to pass the transition sequence number 3 to the second processor. Of course, each list has as many nodes as dependence chains a processor participates in.

The algorithm is shown in the Single-Program-Multiple-Data form in Figure 5. The algorithm consists of an infinite loop where each iteration has two rounds, namely the *Temp1* and *Temp2* rounds. At the beginning of each round, in the first **foreach** statement, processors use *compare&store* to determine the transition sequence numbers in the nodes currently at the head of the dependence chains. These transition sequence numbers are stored in the correct entries of *Temp1* (in the *Temp1* round) or *Temp2* (in the *Temp2* round). Then, in the second **foreach** statement, processors check if their transition sequence numbers are the ones selected in the first **foreach** statement. If so, they read the transition sequence number determined in the *previous* round (not the current one) and save it in the *Ticket* table in place of the question mark. Note that the transition sequence number determined in the *previous* round was saved in the other *Temp* array. This other *Temp* array is reset and the node removed from the *unresolved list*.

We leave the while loop when all processors finish their *unresolved lists*. In our example, in the very first round (*Temp1* round), processor 1 saves its transition sequence number 3 in *Temp1(x)*. It then retrieves the value of 0 from *Temp2(x)* and saves it in *Ticket(1,1)*. In the next round (*Temp2* round), processor 2 saves 9 in *Temp2(x)* and retrieves 3 from *Temp1(x)* and stores it in *Ticket(7,1)*. Processor 3 will then update its question mark entry in the same way. The resulting *Ticket* table is shown in Figure 4(f). Note that, if the dependence chain has only one access, then the corresponding entry in the *Ticket* table will be set to zero. The reason is that, in the very first round in the algorithm of Figure 5, that access will be found at the head of its dependence chain. Therefore, its *Ticket* table entry will be set to the value of the corre-

```
Done ← true;
Temp1(:) ← ∞; /* this can be done in parallel by distributing it to all processors */
Temp2(:) ← 0;
barrier;

while (true) do
/* beginning of Temp1 round */
      foreach (x,y,z) in the unresolved list do
            compare&store{ if (Temp1(x)>z) { Temp1(x) ← z; } }/* atomic instruction */
      endfor;
      barrier;   /* at this point, Temp1(:) has the minimum values of the iteration sequence numbers */
      foreach (x,y,z) in the unresolved list do
            if (Temp1(x) == z) then /* Am I the processor currently at the head of the dependence chain for A(x)? */
               Ticket(y div number_access_per_iteration,y mod number_access_per_iteration) ← Temp2(x);
               Temp2(x) ← ∞;
               delete (x,y,z) from the unresolved list;
            else Done ← false; /* Not my turn yet, will try in the next round */
            endif;
      endfor;
      barrier;
      if (Done == true) stop; /* done, the Ticket table is finished for all processors */
      Done ← true; /* no need to worry about race because of the next barrier */
/* beginning of Temp2 round, very similar to the Temp1 round */
      ...
endwhile;
```

Figure 5: Global inspection algorithm.

sponding entry of *Temp2*. *Temp2* is initialized to 0 at the beginning of the algorithm.

3.2 Executor Phase

We now perform the accesses to the shared data structures. We perform all accesses in parallel except for the dependences specified in the *Ticket* table. As indicated before, we use an extra field (*key*) associated with each element of the shared data structures. This field is initialized to 0 and contains, at any time, the sequence number of the next access to be performed to that particular element. In the course of the executor, a processor will grab iteration i of the *DOACROSS* loop via self-scheduling and, for each access j in the iteration, obtain the sequence number S from *Ticket(i,j)*. Then, instead of accessing the corresponding entry *A(x)*, the processor will wait until *A(x).key* is equal to S. Once this is true, it will perform the access and then update *A(x).key* to the value expected by the next access in the dependence chain.

Usually, this update implies incrementing *A(x).key* by one. The exceptions are the accesses that had once belonged to the *unresolved* list. As indicated above, these accesses increment the *A(x).key* field by a larger amount. To always perform the right update, we can keep an extra number in each entry of the *Ticket* table: the value the *key* field should be set to after the access. However, this implies doubling the size of the table. To save space, we can increment the *key* by one by default unless we were waiting for what was a transition sequence number in the inspector. The latter will be the case if, for a given iteration i in a loop with N iterations and r accesses per iteration executed by P processors, the sequence number we are waiting for is smaller than $L = (i \textit{ div } N/P) * N/P * r + 1$. For example, *Ticket(7,1)* has a value of 3, which is less than $L = 9$, the serial access number in Figure 4(a) for the first entry in the iteration block containing iteration 7. When this occurs, we set *A(x).key* to L (value 9 in the example).

Figure 7 shows the resulting program in Single-Program-Multiple-Data form. While the assignment of iterations to

```
  integer pid,P,Bsize,L,i,N
C pid is the processor id,
C P is the # of processors,
C N is the # of iterations
  Bsize=N/P
C size of the block of iterations assigned
C to one processor in the inspector
  A(:).key=0
C this can be done in parallel by distributing
C to all processors
  call barrier()
  DO i=pid,N,P
C use cyclic scheduling
     L= (i / Bsize)*Bsize*number_refs_per_iter+1
C this value determines if the sequence number
C that we wait for is transition sequence number
     ...
     DO WHILE (A(I1(i)).key != Ticket(i,1))
C busy-waiting
     A(I1(i)).data= ...
C access the data
     IF (Ticket(i,1) <L) A(I1(i)).key=L
C update the key
     ELSE A(I1(i)).key++
     ...
     DO WHILE (A(I2(i)).key != Ticket(i,2))
C busy-waiting
     ... =A(I2(i)).data + ...
C access the data
     IF (Ticket(i,2) <L) A(I2(i)).key=L
C update the key
     ELSE A(I2(i)).key++
     ...
  ENDDO
```

Figure 7: The executor algorithm.

Param.	Explanation	What We Want to Measure	Values
N	Iteration count	Problem size scalability	1600,3200,6400,12800,25600
H_{frac}	Hot spot fraction	Consistency over dependence patterns	10%, 50%, 90%
H_{size}	Hot spot size		10%, 50%, 90%
W	Iteration grain size	Consistency over grain size	40μs (235 cycles), 160μs (941 cycles), 640μs (3765 cycles)
r	# references per iteration	Consistency over number of accesses	1, 2, 4, 8

Table 2: Loop parameters used in the experiments. The cycles in column 4 are 170 ns processor cycles.

processors can be self-scheduled, we choose static cyclic scheduling to reduce overhead. Note that iterations with dependencies can be overlapped now because we consider dependences between accesses instead of between iterations. For example, a processor can access *A(I1(i)).data* in Figure 7 even though *A(I2(i)).data* may not be ready yet. This ability to overlap dependent iterations enables more parallelism in the inspector and executor phases over the Zhu-Yew algorithm. In addition, it removes redundant operations in the inspector. It may, however, increase the spinlocking during execution since the processor may have to wait for *A(I2(i)).key* to become valid.

4 The Experimental System

Our experiments are timing runs on Cedar [5], a 32-processor scalable shared-memory multiprocessor designed at the Center for Supercomputing Research and Development. The machine has 4 clusters of 8 processors each. The latency to access the cache is 170 ns, while accessing the cluster memory takes ~1190 ns and accessing the global memory takes ~2210 ns. The workloads are measured with Cedar's high-resolution 10 μs clock and run in a single-user mode.

Each global memory module in Cedar has a custom-designed synchronization processor. This processor supports *Cedar_sync*, a synchronization primitive that operates on variables composed of two fields, namely *key* and *data*. A *Cedar_sync* operation on address *target*, first locks target and then performs a test on *target.key*. If the test fails, then the operation aborts. Otherwise, it performs operations on *target.key* and *target.data* and then unlocks target. The whole operation is specified in 1 instruction. Clearly, this primitive is useful to implement the *compare&store* instruction used in the inspector phase (Section 3.1). However, it is *not* necessary for our algorithm to run efficiently. In fact, our experimental data shows very similar results when our algorithm is implemented with simple *lock* and *unlock* primitives.

4.1 Workloads

The performance of our algorithm depends on some parameters of the loops that we try to parallelize. Table 2 shows what these parameters are (Column 2) and what they measure (Column 3). In our analysis, we use a loop as in Figure 8 as workload and try to cover as wide a range of values as possible for these parameters. Column 4 shows the range of values chosen.

In the table, iteration count N is assumed to be the same as the size of the array A. A loop with parameter values $N = 6400$ and $W = 160\mu s$ is selected as the "base" loop. The H_{frac} and H_{size} parameters are used to simulate different dependence patterns. Each access has

```
    DO i=1,N,1
       DO j=1,r,1
C We assume alternating read and write accesses
          if (odd(j)) tmp1=A(INDEX(i*r+j))
          else A(INDEX(i*r+j))= tmp2
       ENDDO
       DO j=1,W,1
C A dummy loop simulating useful work
       ENDDO
    ENDDO
```

Figure 8: Loops used as experimental workload; *tmp1* and *tmp2* are dummy variables.

a probability H_{frac} to be a *hot* reference. If it is, then it is targeted to a randomly chosen location in the hot section of array A. The hot section is a portion H_{size} of array A. If the access is not hot, its target location is chosen from the entire A array. Loops with long dependence chains and therefore low parallelism have high H_{frac} and low H_{size}. In the following sections, a "mostly-serial" loop is one with $(H_{frac}, H_{size}) = (90\%, 10\%)$, while a "mostly-parallel" loop has $(H_{frac}, H_{size}) = (10\%, 90\%)$.

All runs are executed on Cedar. A sequential version of each of the loops without inspection overhead was executed to calculate the speedups. Because most results showed similar trends, we will show a few typical ones only.

4.2 Evaluation

In this section, we analyze our experimental results. First, we show the speedup of our scheme over serial execution. Then we compare our scheme to Zhu-Yew's.

4.3 Speedup

To determine the speedup of our algorithm, we consider two situations: one in which it performs both the inspector and the executor (Table 3(a)) and another in which it only performs the executor (Table 3(b)). The latter corresponds to the case where we reuse the *Ticket* table from a previous inspection of the loop. For each case, we consider both the mostly-serial loop and the mostly-parallel loop. We use 32 processors in both experiments, and vary the number of iterations, the iteration grain size, and the number of references per iteration. The latter affects the amount of communication required in the inspector and synchronization in the executor. We are not able to obtain measurement data for entries marked with "n/a" because of some limitations of the experimental system.

Overall, we obtain good speedups: up to 14 when inspector and executor are considered, and up to 27 when only the executor is considered. Obviously, the best results occur when the size of the loop body (W) is large and the

W	r	Mostly-Serial Loop					Mostly-Parallel Loop				
		N =1600	3200	6400	12800	25600	N =1600	3200	6400	12800	25600
40 μs	8	0.74	0.79	n/a	n/a	n/a	0.98	1.00	n/a	n/a	n/a
	4	0.90	0.89	0.89	n/a	n/a	1.28	1.30	1.30	n/a	n/a
	2	1.22	1.26	1.22	1.21	n/a	1.91	1.98	1.76	1.73	n/a
	1	1.64	1.83	2.00	1.93	1.87	1.80	2.50	2.71	2.46	2.35
160 μs	8	1.10	1.13	n/a	n/a	n/a	1.36	1.42	n/a	n/a	n/a
	4	1.50	1.56	1.56	n/a	n/a	2.26	2.26	2.25	n/a	n/a
	2	2.35	2.65	2.47	2.54	n/a	3.49	3.76	3.59	3.49	n/a
	1	3.66	4.36	4.76	4.53	4.45	4.67	6.18	5.48	5.80	5.58
640 μs	8	2.41	2.48	n/a	n/a	n/a	2.96	2.97	n/a	n/a	n/a
	4	3.93	3.86	3.90	n/a	n/a	5.67	5.40	5.39	n/a	n/a
	2	6.23	6.94	6.70	6.81	n/a	9.23	9.57	9.11	8.97	n/a
	1	9.55	11.95	11.39	11.36	11.41	11.65	13.60	14.19	13.60	13.52

(a)

W	r	Mostly-Serial Loop					Mostly-Parallel Loop				
		N =1600	3200	6400	12800	25600	N =1600	3200	6400	12800	25600
40 μs	8	4.30	6.21	n/a	n/a	n/a	9.98	9.09	n/a	n/a	n/a
	4	9.22	10.77	9.43	n/a	n/a	11.96	8.77	10.38	n/a	n/a
	2	14.08	13.11	10.68	9.76	n/a	13.07	15.71	11.02	10.49	n/a
	1	12.36	11.08	10.62	11.17	11.19	14.78	12.76	12.14	12.78	11.17
160 μs	8	6.79	8.02	n/a	n/a	n/a	12.14	12.07	n/a	n/a	n/a
	4	14.08	11.71	12.43	n/a	n/a	19.42	14.72	15.12	n/a	n/a
	2	18.23	18.43	14.84	15.94	n/a	21.32	19.07	17.02	15.25	n/a
	1	16.57	21.27	19.77	18.02	18.54	18.42	21.36	21.57	18.59	18.97
640 μs	8	13.52	15.11	n/a	n/a	n/a	17.99	16.92	n/a	n/a	n/a
	4	22.80	21.36	20.25	n/a	n/a	25.54	21.69	21.22	n/a	n/a
	2	21.30	23.58	22.64	23.10	n/a	26.18	24.33	24.05	23.75	n/a
	1	26.59	26.06	24.25	25.12	25.51	24.46	26.32	25.90	25.69	26.33

(b)

Table 3: Speedup of our algorithm including (a) inspector and executor; and (b) executor only. The experiments correspond to 32 processors.

number of accesses (r) is low. In this case, the loop has a low communication-to-computation ratio. Of course, when the opposite is true, speedups will be lower. Tables 3 also show that, as expected, the performance is better if the dependence chains are short; this is why the speedup of the mostly-parallel loops is higher than the speedup of the mostly-serial loops. In summary, the results show that run-time parallelization is a viable approach to speed up loops that cannot be handled by compile-time parallelization. This approach is especially promising if the compiler is sophisticated enough to reuse the results of the inspector analysis across loop invocations.

To see how the speedup varies with the number of processors, we also ran several experiments with mostly-serial and mostly-parallel loops using various numbers of processors. With up to 32 processors, we obtain almost linear speedups when measuring (1) both inspector and executor; and (2) executor only [2]. We expect the speedup to be scalable with even larger numbers of processors.

4.4 Performance Comparison with Zhu-Yew's Scheme

Since Zhu-Yew's scheme has the same applicability and requires the same system support as ours, we have compared its performance to that of our algorithm. The results can also be applied to another scheme proposed by Midkiff and Padua [7] because they use the same inspector algorithm and therefore have similar communication patterns. Table 4 shows the ratio between the execution time of Zhu-Yew's algorithm and ours for 32 processors.

The table shows that our scheme is nearly always faster than Zhu-Yew's. In some cases, it is up to 37 times faster. Furthermore, our algorithm is relatively faster in the mostly-serial loops, where it is harder to obtain speedup. This is because our scheme allows overlapped execution of dependent iterations and reduced communication via iteration blocking.

The table also shows that Zhu-Yew's scheme is sometimes slightly faster than our scheme. However, for those cases, the absolute speedup for our scheme is fairly good already (Table 3(a)). Consequently, it is clear that the performance of our scheme is more consistent than Zhu-Yew's scheme's. In addition, we do not consider the reuse of the *Ticket* table in the comparison. If the table is reused, the speedup for our scheme will be much higher because there will be no inspector overhead. To understand why our scheme performs better, we now examine the issues of communication overhead. and sensitivity to access patterns.

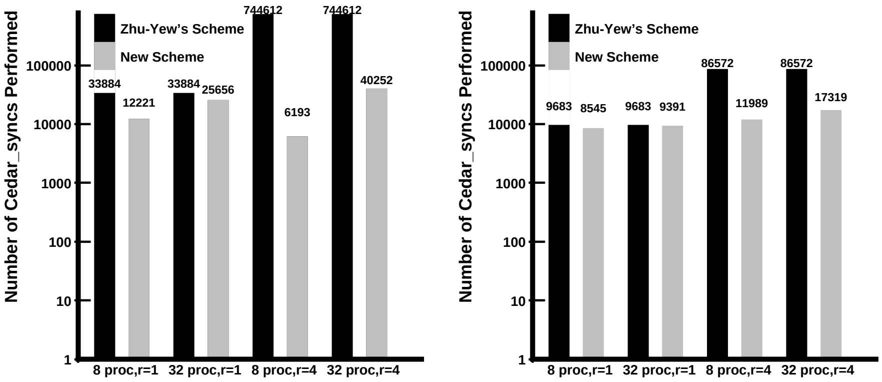
Communication Overhead

To compare the communication overhead in the two algorithms, we compare the number of synchronization operations. There are two types of synchronization operations, namely *compare&stores* and barriers. Again, we stress that while we implement the former with the *Cedar_sync* primitive on Cedar, it can be implemented with simple *lock* and *unlock*. In the following, we measure the number of *Cedar_sync*s and barrier operations.

Figure 9 depicts the number of *Cedar_sync*s performed in

W	r	Mostly-Serial Loop					Mostly-Parallel Loop				
		N =1600	3200	6400	12800	25600	N =1600	3200	6400	12800	25600
40 μs	8	30.68	37.86	n/a	n/a	n/a	7.40	7.93	n/a	n/a	n/a
	4	13.24	14.04	14.08	n/a	n/a	2.62	2.82	2.69	n/a	n/a
	2	5.01	5.22	5.11	4.98	n/a	1.20	1.16	1.03	0.96	n/a
	1	1.58	1.65	1.71	1.61	1.52	0.55	0.85	0.73	0.69	0.67
160 μs	8	31.70	37.55	n/a	n/a	n/a	7.22	7.92	n/a	n/a	n/a
	4	12.93	13.96	13.89	n/a	n/a	2.77	2.86	2.70	n/a	n/a
	2	4.57	5.05	4.78	4.84	n/a	1.04	1.12	1.03	0.94	n/a
	1	1.58	1.66	1.72	1.56	1.48	0.72	0.91	0.71	0.73	0.73
640 μs	8	31.69	37.35	n/a	n/a	n/a	7.25	7.61	n/a	n/a	n/a
	4	13.04	13.13	13.05	n/a	n/a	2.88	2.79	2.61	n/a	n/a
	2	4.49	4.76	4.49	4.39	n/a	1.27	1.27	1.15	1.03	n/a
	1	1.71	1.85	1.61	1.50	1.42	0.95	1.00	0.90	0.87	0.84

Table 4: Ratio between the execution time of Zhu-Yew's scheme and ours using 32 processors.



(a) Mostly-Serial Loop

(b) Mostly-Parallel Loop

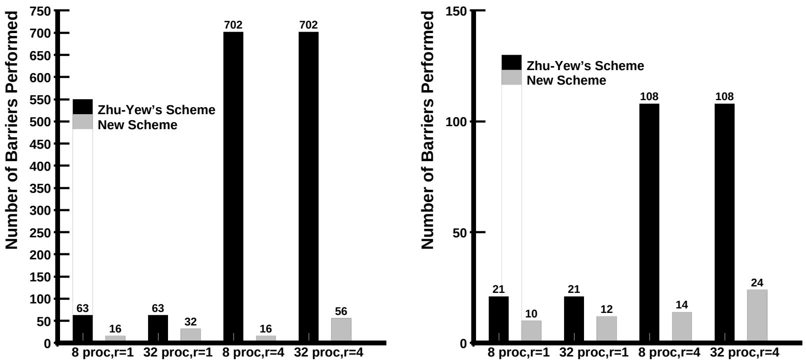
Figure 9: Comparing the number of *Cedar_sync*s in the two algorithms for the base loop with $r = 1$ or 4 and 8 or 32 processors.

each scheme using the base loop and $r = 1$ or 4 references per iteration. The experiments use 8 and 32 processors and run the mostly-serial and mostly-parallel loops. Because of the big difference in some of the results, we use logarithmic scale in the y-axis. From the figure, we see that when there are many dependences, such as in the mostly-serial loops (Figure 9(a)), and when the number of references per iteration is high, the new algorithm uses fewer *Cedar_sync*s. For example, for the mostly-serial loops with $r = 4$ and 8 processors, the new scheme has 1% of the *Cedar_sync* in Zhu-Yew's scheme. One reason for this is that the local inspector in the new scheme reduces the amount of communication required. A second reason is that in Zhu-Yew's scheme, iterations with dependences cannot overlap at all, and therefore more retries are necessary.

With mostly-parallel loops and few references in each iteration, the differences in the total number of *Cedar_sync*s in the two schemes are small. This is shown in Figure 9(b). In this case, the new scheme does not have a clear advantage because, if the loop has few dependences, few dependent iterations are overlapped in the new scheme. In addition, the local inspector in the new scheme is relatively less effective and most of the dependence chains are identified in the global inspector using *Cedar_sync*.

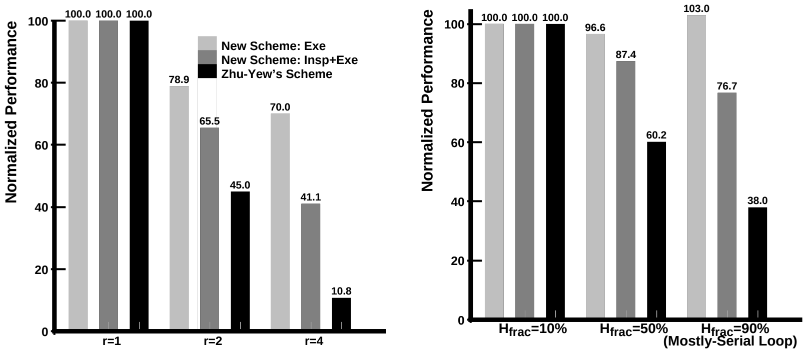
Another type of synchronization in these schemes is barriers between concurrent loops in the inspector. Barriers not only consume network bandwidth but also introduce processor idle time. Our scheme has fewer barriers. There are two reasons for this. First, in the new scheme, the dynamic number of barriers is proportional to the longest chain of dependent accesses, while in Zhu-Yew's scheme it is proportional to the longest chain of dependent iterations. Second, many barriers are eliminated in the new scheme if the sources and the sinks of the dependences are in iterations given to the same processor in the local inspector. The resulting number of barriers in the two schemes for the base loops is shown in Figure 10. The figure corresponds to $r = 1$ and $r = 4$ using 8 and 32 processors. From the figure, we see that the new scheme requires fewer barriers than Zhu-Yew's, in particular for $r = 4$ and mostly-serial loops (Figure 10(a)).

To summarize, our scheme requires fewer *Cedar_sync* operations and barriers, particularly when there are many dependences. This implies lower network traffic and less processor busy-waiting time. The reason is that our algorithm solves dependences between references instead of between iterations. This greatly reduces the length of the dependence chain. Furthermore, many *Cedar_sync* in Zhu-Yew's scheme are redundant and are eliminated in the new scheme. Finally, we note that for the new scheme, the plots show a correlation between the number of processors used and the communication required: more processors introduce more partition boundaries in the local inspector, which result in higher communication in the global inspector.



(a) Mostly-Serial Loop (b) Mostly-Parallel Loop

Figure 10: Number of barrier synchronizations in the two schemes for the base loops with $r = 1$ or 4 and 8 or 32 processors.



(a) Sensitivity to the number of references per iteration.

(b) Sensitivity to the fraction of accesses to hot locations for H_{size}=10%.

Figure 11: Performance sensitivity to access and dependence patterns. The number of processors is 32.

Sensitivity to Access and Dependence Patterns

Since the performance of run-time schemes varies when loop characteristics change, we measured the sensitivity to the number of references per iteration r (Figure 11(a)), and the fraction of accesses to the hot location H_{frac} (Figure 11(b)). In Figure 11(a), we plot the base mostly-parallel loop, with 1, 2, and 4 references per iteration running with 32 processors. The bars show the performance normalized to the loop with $r = 1$. For each set of three columns, the leftmost one corresponds to the executor phase of the new algorithm, the central one to the inspector and executor phases of the new algorithm, and the rightmost one to Zhu-Yew's scheme. From Figure 11(a), we see that the performance decreases as we increase the number of accesses per iteration. The reason is that the ratio of communication-to-computation has increased. However, our scheme is much less sensitive than Zhu-Yew's scheme, particularly if only the executor phase is considered.

In Figure 11(b), we use the base loop with $r = 1$ and 32 processors, and set H_{frac} to 10%, 50%, and to 90%. In all cases, hot locations account for 10% of the shared locations ($H_{size} = 10\%$), therefore the rightmost set of three bars corresponds to mostly-serial loops. The columns are normalized to the $H_{frac} = 10\%$ case. From Figure 11(b), we see that the overall performance of our scheme changes less than 25% (even if the inspector is considered) compared to the more than 60% drop in performance for the Zhu-Yew's scheme. For the executor-alone column, the small increase in performance when $H_{frac} = 90\%$ is likely to be the result of the changed dependence pattern among iterations. Overall, these results indicate that the performance of our scheme is more consistent than that of Zhu-Yew's scheme.

5 Conclusions

The automatic parallelization of loops is usually based on compile-time analysis of data dependences. In some loops, however, the data dependences cannot be determined at compile time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, it is necessary to use a more expensive analysis that is performed at run time. In this paper, we presented and evaluated a new algorithm to parallelize these loops at run-time. Our scheme handles any type of data dependence pattern without requiring any special architectural support. Furthermore, compared to an older scheme with the same generality, it speeds up execution by allowing the reuse of the inspector phase across loop invocations, allowing partial overlap of dependent iterations, and optimizing the inspector for high locality and low communication.

We have evaluated our algorithm with a set of parameterized loops running on the 32-processor Cedar shared-memory multiprocessor. We used loops with varying parameters, such as number of iterations and references. The results show that our algorithm gives speedups that reach 14 if the inspector is not reused and 27 if it is. Furthermore, our algorithm outperforms Zhu-Yew's scheme [14] in nearly all cases, reaching a 37-fold speedup when the loop has many dependences.

There are a few issues in run-time parallelization that we are currently working on. The first one is to use the dependence information gathered by the inspector to rearrange the order of execution of iterations in the executor to minimize busy-waiting time. This strategy would imply spreading the iterations that participate in a dependence chain as much as possible so that the processors that execute them do not have to wait for each other. This is not currently done and, therefore, processors spend time busy-waiting for the previous access in the dependence chain to finish.

A second optimization is to eliminate the serialization of reads with input dependences. Currently, our scheme serializes all accesses to the same location, including concurrent reads. This is not necessary. Finally, we are developing compiler algorithms to determine if the inspector results can be reused across loop invocations. This approach requires very aggressive interprocedural analysis to assure that the index arrays are not changed.

Acknowledgement

The authors would like to thank Alain Raynaud for his help in the experiments, Merle Levy for her careful reading of an earlier version of this manuscript, and the anonymous referees for their helpful comments.

References

[1] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[2] D.-K. Chen, J. Torrellas, and P.-C. Yew. Run-time parallelization for DOACROSS loops. CSRD Report No. 1345, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, April 1994.

[3] D.-K. Chen and P.-C. Yew. A scheme for effective execution of irregular DOACROSS loops. In *Int'l. Conf. on Parallel Processing*, pages 285–292, August 1992. Also available as CSRD tech report No. 1192.

[4] V. Krothapalli and P. Sadayappan. An approach to synchronization of parallel computing. In *ACM Int'l. Conf. on Supercomputing*, pages 573–581, June 1988.

[5] D. Kuck et al. The Cedar system and an initial performance study. In *20th Int'l Symp. on Computer Architecture*, May 1993.

[6] S.-T. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

[7] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. on Computers*, C-36(12), December 1987.

[8] D. A. Padua. *Multiprocessors: Discussion of Some Theoretical and Practical Problems.* PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1979.

[9] C. Polychronopoulos. Advanced loop optimizations for parallel computers. In E. Houstis, T. Papatheodorou, and C. Polychronopoulos, editors, *Lecture Notes in Computer Science No. 297: Proc. First Int'l. Conf. on Supercomputing, Athens, Greece*, pages 255–277, New York, June 1987. Springer-Verlag.

[10] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. on Computers*, 40(5):603–611, May 1991.

[11] T. H. Tzen and L. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Trans. on Parallel and Distributed Systems*, 4(5), May 1993.

[12] M. J. Wolfe. *Optimizing Compilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[13] M. J. Wolfe. Loop skewing: The wavefront method revisited. *Int. J. of Parallel Programming*, 4(15), 1986.

[14] C.-Q. Zhu and P.-C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. on Software Enginering*, pages 726–739, June 1987.