Fine-grain Loop Scheduling for MIMD Machines

Technical Report 90-34

Carrie J. Brownhill, Ki-chang Kim and Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

October 2, 1990

Abstract

Previous algorithms for parallelizing loops on MIMD machines have been based on assigning one or more loop iterations to each processor, introducing synchronization as required. These methods exploit only iteration level parallelism, and ignore the parallelism that may exist at a lower level.

In order to exploit parallelism both within and across iterations, our algorithm analyzes and schedules the loop at the statement level. The loop schedule reflects the expected communication and synchronization costs of the target machine. We provide test results that show that this algorithm can produce good speedup of loops on an MIMD machine.

Keywords - MIMD, Static code transformations, parallelizing compilers, loop scheduling, communication costs.

1 Introduction

Loop scheduling is the process of mapping the operations inside a loop to the available processing units at compile time. When the loop executes, each processor will execute only the operations assigned to it, but may need to communicate or synchronize with the other processors, if they access the same data. The scheduling algorithm should minimize the amount of idle time caused by this communication and synchronization.

To schedule the work to the available processing units, a loop may be partitioned along operation boundaries, so that some or all executions of a particular operation are done by one processor. This mode of execution can be very efficient, as is the case for vector machines [4]. Unfortunately, many real-life loops contain dependencies and control flow that make them hard to vectorize.

Another way to divide the loop is along iteration boundaries. In this case, one or more iterations is assigned to each processor [8], with any necessary synchronization added to handle loop-carried dependencies, e.g., Doall, Dopipe, and Doacross loop scheduling [5,7]. These methods of loop scheduling have been extensively used on MIMD machines. They exploit coarse grain (iteration level) parallelism, but ignore any parallelism that may exist at a lower level of granularity.

The algorithm we discuss in this paper uses fine-grain (statement level) loop parallelization techniques in an attempt to fully exploit the parallelism available in a loop. While not all of the fine-grain parallelism exposed by these techniques can be utilized on machines with relatively high communication and synchronization costs, we have found that the extra parallelism is often useful in speeding up the execution time of loops by masking the latency of communication and synchronization delays.

In principle, the parallelization technique we use is able to expose *all* of the parallelism available in a loop, subject only to the data dependencies [1,2]. However, in order to map a loop effectively to an MIMD machine, the communication and synchronization costs of the machine need to be taken into account. These costs may vary dynamically even on the same architecture. Nevertheless, a good *estimate* of the typical cost can usually be obtained. Adapting a loop schedule according to expected communication and synchronization costs can improve the execution time of the loop, while minimizing run-time overhead.

2 Algorithm

Perfect Pipelining (PP) [1, 2] is a fine-grain parallelization technique that has been developed for use on VLIW (Very Large Instruction Word) machines, with synchronous processors and no communication costs. On this type of machine, with sufficient resources, PP produces an optimal loop schedule. The same schedule will be optimal for an MIMD machine with sufficient resources, if communication and synchronization costs are negligible. For MIMD machines with measurable communication and/or synchronization costs, the schedule must be adapted to allow for these costs.

This paper describes the technique for adapting the PP schedule for use on MIMD machines. The algorithm presented will 'trade-off' communication and synchronization costs for extra parallelism. It will only schedule a statement on a separate processor, if that assignment is expected to improve the execution time of the loop. Thus, the granularity of the parallelism is automatically adapted to the operating parameters of the machine.

2.1 Loop Model

To begin, we describe our model of a loop using a dependence graph representation. In the graph of a sequential loop, each node represents a statement, and each arc represents a data dependence. See Figure 1. For simplification of our examples, we will assume that the dependence distances in the graph are all either zero or one. That is, all dependencies are within the same iteration, or between adjacent iterations. Loops with larger dependence distances can be handled by unwinding the loop as required [6]. In addition, any branch tests present in the loop must be if-converted, as described in [3]. [1]

For our purposes, it is useful to partition the statements of the loop into three disjoint sets. The set of *flow-in* statements includes all statements which either have no predecessors in the data dependence graph, or which only have predecessors which are themselves in the flow-in set. Similarly, the set of *flow-out* statements includes all statements which are not flow-in and which have no successors, or which have only successors which are themselves in the flow-out set. The set of *cyclic* statements includes all statements which are neither flow-in nor flow-out. In Figure 1, statements 0, 1, 2 3, and 5 are flow-in. Nodes 7, 8, and 10 are flow-out. The rest are cyclic.

Separating the loop into these three sets is a key part of the algorithm. Essentially, the cyclic

[1] While PP can exploit fine-grain parallelism beyond conditional branches without if-conversions, the use of such parallelism does not seem efficient when communication costs are high. Therefore, for the purposes of this paper, we assume that conditionals are if-converted.

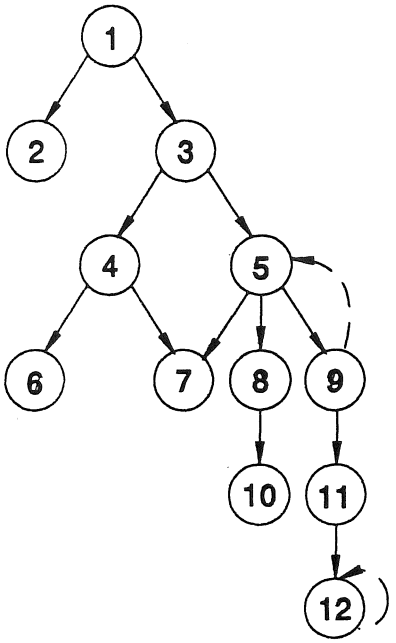

Figure 1: Example dependence graph

statements are those statements on the 'critical path' of the loop. They are involved in both producing and consuming values. By scheduling the cyclic statements first, the expected execution time of the loop can be estimated. The flow-in and flow-out statements can then allocated sufficient processors (if available) to insure that they finish at about the same time as the cyclic statements.

2.2 Partitioning the Statements

The first step in the algorithm is to divide the statements in the loop into the cyclic, flow-in and flow-out subsets. The algorithm for this partitioning is as follows.

1. Each statement is given a tag which is originally marked 'cyclic'.
2. All statements without predecessors are marked 'flow-in'.
3. The set of cyclic statements is searched for statements with only predecessors marked 'flow-in'. If any are found, then they are marked 'flow-in'. This step is repeated until one iteration is completed without finding any new flow-in statements.
4. All statements marked 'cyclic' which have no successors are marked 'flow-out'.

Time	Scheduled Operations (Iteration)			
0	1(1)			
1	2(1) 3(1)			
2	4(1)	1(2)		
3	6(1)	2(2) 3(2)		
4	5(1)	4(2)	1(3)	
5	7(1) 8(1) 9(1)	6(2)	2(3) 3(3)	
6	10(1) 11(1)	5(2)	4(3)	1(4)
7	12(1)	7(2) 8(2) 9(2)	6(3)	2(4) 3(4)
8		10(2) 11(2)	5(3)	4(4)
9		12(2)	7(3) 8(3) 9(3)	6(4)
10			10(3) 11(3)	5(4)
11			12(3)	7(4) 8(4) 9(4)
12				10(4) 11(4)
13				12(4)

Table 1: Schedule for four iterations of Perfect Pipelining example.

5. The set of cyclic statements is searched for statements with only successors marked 'flow-out'. If any are found, they are marked 'flow-out'. This step is repeated until one iteration is completed without finding any new flow-out statements.

2.3 Scheduling Cyclic Statements

Intuitively, Perfect Pipelining (PP) schedules on an 'as soon as possible (ASAP)' basis, as restricted by data and control flow dependencies. Each operation in the loop is scheduled to be executed as soon as the operations it is dependent on finish execution. The scheduling process continues across iteration boundaries, with the loop being incrementally unwound. On VLIW machines, the exact execution time of each operation is known, so that the loop schedule consists of sets of operations which will be executed in parallel at each cycle. As an example, the schedule of four iterations of the example loop represented in Figure 1 is shown in Table 1. In the table, the first number corresponds to the operation, and the number in parentheses is the iteration number.

Eventually, the sets of scheduled operations form a repeating pattern. Once this pattern is recognized, the loop does not need to be unwound any further. The pattern itself can be substituted

Time	Operation (Iteration)
0	1(4) 4(3) 5(2) 10(1) 11(1)
1	2(4) 3(4) 6(3) 7(2) 8(2) 9(2) 12(1)

Table 2: Pattern for Perfect Pipelining example.

for the body of the loop. [2] The pattern for the example loop is outlined in Table 1 and shown in Table 2.

In this algorithm, the PP technique is used to find the loop pattern for the cyclic, flow-in, and flow-out subsets of statements. The optimal pattern found, is then adjusted to take communication and synchronization costs into account.

2.4 Adapting the Ideal Pattern to Reflect Communication and Synchronization Costs

Once the cyclic statements are identified, they can be scheduled. The PP algorithm is used to find the loop pattern of these statements. This pattern is an optimal schedule for this loop subset, if there are no communication costs. To produce a schedule which reflects communication costs, the algorithm is as follows.

1. Allocate P_c processors, where P_c is equal to the maximum number of statements scheduled in parallel in the optimal pattern. (If there is a limited number of resources, the number of allocated processors can be decreased as described in the next section. However, this may increase execution time.)

2. Taking each statement in order according to the pattern, schedule it on the processor which allows it to begin execution soonest, taking into account any synchronization and communication costs. To do this, the following things are determined for each possible processor assignment.

[2] For the majority of loops encountered in practice, the above description is accurate. However, there are some loops for which a pattern does not emerge naturally. To provide for these loops, restrictions can be placed on the scheduling transformation. Namely, a limit is put on the distance that operations from the same iteration can stretch from each other in the schedule. This guarantees detection of the pattern, while leaving the optimality virtually unaffected. For proofs, see [2,1].

(a) Set S_p, the earliest possible start time, to initially be the value of the next free time slot on this processor.

(b) Iterate over the set of statements in the loop on which this statement is dependent (in other words, the statement we are scheduling needs to wait for a value from each statement in this set).

i. Statements which are scheduled on the current processor can be ignored. There is no synchronization necessary. However, we need to count the statements that are scheduled on other processors, or not yet scheduled. (We include statements not yet scheduled to produce a conservative count.) This gives us R, the number of receives that will be necessary.

ii. In addition, for those statements that are already scheduled on another processor, we need to remember the latest start time of any send to this statement. This time, LS, is equal to the scheduled start time of the sending statement plus its estimated execution time.

(c) The earliest start time of the statement on this processor, S_p is therefore equal to the following:

$S_p = \max(S_p, LS + T_s + T_c) + (R * T_r)$

where:

LS = Latest expected send time

T_s = Execution time of one send

T_r = Execution time of one receive

T_c = Expected communication delay between processors

R = Number of receives required

3. Schedule the statement on the processor that has the earliest S_p, and update the next free cycle time for that processor.

4. Record the longest expected execution time (including communication time) of these processors. This number, E_c, is used when allocating processors to the flow-in and flow-out statements.

2.5 Allocating Processors for Cyclic Statements

The number of processors allocated to the cyclic statements is ideally equal to the maximum number of statements scheduled in parallel in the pattern. As the cyclic statement are scheduled, each one is assigned to the processor on which it has the earliest expected execution time.

Unless there is no communication and synchronization cost, there is an advantage to scheduling a statement on the same processor as any statements on which it is dependent, thereby making the necessary synchronization implicit. However, if there are intervening statements already scheduled on that processor, then the statement may execute sooner on another, available processor, rather than having to wait for the intervening statements to finish execution. Therefore, there is a trade-off between the parallelism, and the communication and synchronization costs. If these costs are high, the scheduler may actually assign all of the statements to some subset of the allocated processors, because of the reduced amount of synchronization. Processors which do not end up with any assigned statements can easily be reclaimed for use by the flow-in and flow-out statements.

If there are limited resources, the number of processors allocated to the cyclic statements can be reduced to fit the target machine. Because the scheduler only tries to assign statements to the allocated processors, it automatically adapts the schedule as necessary. If there are less than the ideal number of processors available, then the scheduler will end up assigning more statements to each processor. A good heuristic is to make the number of cyclic processors, P_c = min (maximum parallelism, expected execution time of the cyclic statements / expected execution time of (flow-in + flow-out statements).

2.6 Allocating Processors for Flow-in and Flow-out Statements

The scheduler attempts to allocate enough processors to the flow-in and flow-out statements to ensure that they finish at approximately the same time as the cyclic statements. To do this, it must calculate the expected execution time, E_f, of one iteration of the loop subset (flow-in/flow-out). The execution time of the cyclic statements, E_c, is equal to the longest expected execution time of all the cyclic processors. (This number was stored as described in Section 2.4.) The number of processors allocated is then equal to $\lceil E_c/E_f \rceil$.

Again, if there are a limited number of processors, then the number of processors allocated to the flow-in and flow-out statements can be reduced heuristically, at the cost of increasing the execution time of the loop. As in the case of the cyclic statements, the processors are allocated in the same ratio as the expected execution time of the loop subsets.

2.7 Scheduling Flow-in and Flow-out Statements

The flow-in and flow-out statement sets are scheduled separately, but the algorithm for both is the same, so it is described only once. First, the pattern for each subset is generated, and the following method followed.

1. P_f is defined as the number of processors allocated to this set of statements. Give each processor in P_f a unique number p, ranging from zero to $(P_f - 1)$.

2. For each statement in the set, schedule it on all processors. However, each processor, p, only executes iterations i, where $(i \bmod P_f) = p$.

The flow-in and flow-out statements are rearranged according to the pattern found using the PP technique to reduce the delay caused by loop carried dependencies. Given enough resources, the flow-in and flow-out processors should not dominate the execution time of the loop, because all of the dependencies in these subsets go forward.

2.8 Generating the Schedule

Once the schedule has been determined, the program must generate the final schedule for each process. Any loop prologue or epilogue necessary must be included in the final schedule. In addition, the program must generate all the necessary synchronizations.

Each synchronization point contains the following information:

1. The identity of the sending statement.

2. The identity of the processor that the sending statement is scheduled on. (This is necessary for absolute identification, because flow-in and flow-out statements may be scheduled on multiple processors.)

3. The identity of the receiving statement.

4. The identity of the processor that the receiving statement is scheduled on.

5. Whether this synchronization point is a send or a receive.

6. The iteration of the data being shared.

If a statement requires a synchronization with a flow-in or flow-out statement which is scheduled on multiple processors, the processor identity will depend on the iteration number. The correct processor can be calculated at run-time according to the iteration number in the synchronization. Otherwise, the loop can be unrolled by the scheduler, so that the synchronization can be explicit.

Each statement in the schedule is processed in the following way:

1. Iterate over the set of statements in the loop on which this statement is dependent. Generate a 'receive' synchronization for each of these which is scheduled on a different processor.
2. Generate the statement itself.
3. Iterate over the set of statements in the loop which depend on this statement. Generate a 'send' synchronization for each of these which is scheduled on a different processor.

3 Examples

Our algorithm was implemented on a Sequent Symmetry with eight processors. The program takes as input the dependence graph of the loop and the expected execution time (latency) of each statement. The communication and synchronization costs are specified with three numbers: The estimated time for a send (or synchronization), the estimated time for a receive (or synchronization), and the communication delay. The number of processors available is also specified. The scheduler outputs the loop schedule for each processor.

The Sequent is a shared memory MIMD machine. Processors share direct access to memory, governed by synchronization inserted to preserve the original execution semantics. The cost of synchronization was timed as described in the following section, and used as input into the scheduler. The Sequent is not known to facilitate very fast synchronization. While machines with much faster synchronization exist, we chose the Sequent as our first target, both for reasons of convenience (it's available at our site at UCI) and because it provides a rather extreme environment. If we obtain positive results on it, we should do much better on machines with hardwired support for synchronization, such as the Alliant.

3.1 Synchronization cost

To implement the required synchronization, mailboxes (storage spots) are allocated. Each pair of communicating statements gets its own mailbox. When the 'sending' statement has finished

Operation	Average Time
Synchronization	1
A[1][i] = A[2][i]	1
A[1][i] = A[2][i] + A[3][i]	2

Table 3: Execution times on the Sequent

calculating the necessary data, it waits until the mailbox is available, and sets it. The 'receiving' statement waits until it sees the signal, and then resets the mailbox. Because only one processor does a 'send' and only one processor does a 'receive' to each mailbox, no race conditions exist, and therefore no locking is necessary. This makes the synchronization fairly efficient, even on the Sequent.

Actually, each send and receive pair gets allocated two mailboxes, which are used on alternate iterations. While most of the time, the extra mailbox is not necessary, it allows one process to lag an iteration behind the other without leaving the mailbox blocked. This may help 'cushion' variations in execution speed. In addition, with only one mailbox, a processor with a loop carried dependence may actually force another processor to be always one iteration 'behind', because it continually uses the value from the previous iteration. The other processor must wait for the mailbox to be cleared before it can continue. If two processors have loop carried dependencies on each other, they may both need to be one iteration ahead of the other and will therefore deadlock, waiting for the mailboxes to clear. Having two mailboxes allows both processors to be executing and writing the current iteration of values, without waiting for the previous iteration to clear.

The execution of the synchronization primitives was timed, so that this information could be used when scheduling loops. In addition, the execution times of operations on data of type double were also measured, in order to estimate the execution time of the loop statements. The results are in Table 3. The average execution times were calculated by taking the total execution time of 360 iterations, subtracting the loop overhead time, and dividing the result by 360.

The execution times were used to estimate the cost of synchronization and the execution times of loop statements on the Sequent. Although the actual execution times could be used, the variance on a multi-processing machine like the Sequent do not warrant such precision; we chose to approximate the synchronization cost as one unit. Each statement in the loop was estimated to take an execution time in units equal to the number of data loads on the right hand side of the equation.

3.2 The Effect of Communication and Synchronization Costs on Schedules

By using parallelism to mask synchronization latency, our performance improves over previous techniques, since more parallelism is available for this purpose. Of course, if communication cost is disproportionately large, relative to instruction execution time, the most efficient way of executing the code may well be sequential, and neither our approach, or any other, will be able to speedup execution.

The possible effect of the communication and synchronization costs on the loop schedule can be seen in an example taken from [5]. The sequential code for this loop is shown in Figure 2 and the dependence graph is shown in Figure 3. In this example, there are six cyclic statements (1, 2, 3, 4, 6, and 7) and eleven flow-in statements. There are no flow-out statements.

If the communication and synchronization costs are input as zero, the schedule produced for the cyclic statements is that shown in Figures 4 and 5. The schedule for the flow-in statements is shown in Figure 6. However, if the synchronization costs are specified as being one unit for a send and one unit for a receive (as was timed on the Sequent), the schedule for the cyclic statements changes to that shown in Figures 7 and 8. The schedule for the flow-in statements does not change. Notice that there is less synchronization required in the second schedule, but there are more statements scheduled on one processor than on the other, which would appear to be less efficient, if the communication delay is not taken into account.

These two schedules were run on the Sequent. As a comparison, the loop was also scheduled and run using a Doacross scheduling algorithm, with redundant synchronizations removed. The same synchronization method was used for all the loops. The total execution times were measured and are shown in Table 6. Speedup is also shown and was calculated using the standard formula, speedup = (sequential time - parallel time) / sequential time. These numbers illustrate that adapting the schedule to the expected communication costs can produce a better speedup.

```
for (i = 1; i <= N; i++)
{
    A[0][i] = B[i];
    A[1][i] = A[7][i - 1];
    A[2][i] = A[4][i - 1];
    A[3][i] = A[2][i] + A[6][i - 1];
    A[4][i] = A[1][i];
    A[5][i] = A[0][i] + A[12][i - 1];
    A[6][i] = A[3][i];
    A[7][i] = A[3][i] + A[4][i] + A[16][i - 1];
    A[8][i] = A[0][i];
    A[9][i] = A[8][i] + A[14][i - 1];
    A[10][i] = A[8][i];
    A[11][i] = A[8][i];
    A[12][i] = A[11][i];
    A[13][i] = A[12][i];
    A[14][i] = A[13][i];
    A[15][i] = A[13][i];
    A[16][i] = A[13][i];
}
```

Figure 2: Sequential loop example from [5].

Time	Statement (Iteration)
0	A[2](2) A[7](1)
1	A[3](2)
2	
3	A[1](2) A[6](2)
4	A[4](2)

Table 4: Pattern for cyclic statements.

Time	Statement (Iteration)
0	A[0](1)
1	A[8](1)
2	A[11](1) A[10](1)
3	A[12](1)
4	A[5](1) A[13](1)
5	A[15](1) A[14](1) A[16](1)
6	A[9](1)
7	

Table 5: Pattern for flow-in statements.

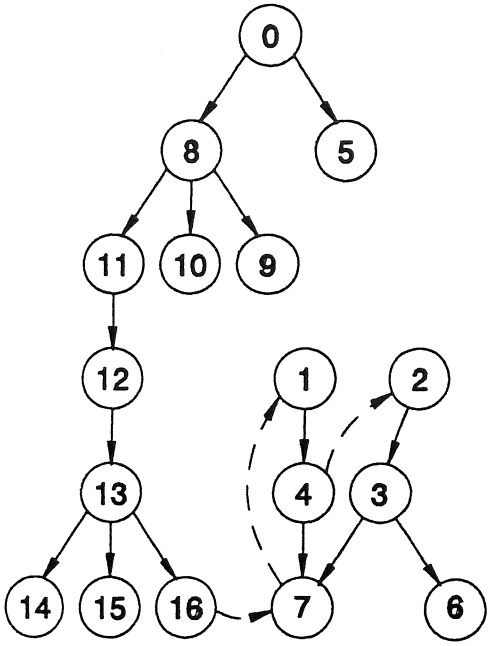

Figure 3: Dependence graph

```
/* Loop prologue */
A[2][i] = A[4][i - 1];
RECEIVE_FROM_S6;
A[3][i] = A[2][i] + A[6][i - 1];
SEND_TO_S6;
SEND_TO_S7;
RECEIVE_FROM_S7;
A[1][i] = A[7][i - 1];
A[4][i] = A[1][i];
SEND_TO_S7;

for (i = 1; i <= N - 1; i++)
{
    A[2][i + 1] = A[4][i];
    RECEIVE_FROM_S6;
    A[3][i + 1] = A[2][i + 1] + A[6][i];
    SEND_TO_S6;
    SEND_TO_S7;
    RECEIVE_FROM_S7;
    A[1][i + 1] = A[7][i];
    A[4][i + 1] = A[1][i + 1];
    SEND_TO_S;
}
```

Figure 4: Schedule for processor 0, no communication costs.

```
/* Loop prologue */
RECEIVE_FROM_S3;
A[6][i] = A[3][i];
SEND_TO_S3;

for (i = 1; i <= N - 1; i++)
{
    RECEIVE_FROM_S16;
    RECEIVE_FROM_S3;
    RECEIVE_FROM_S4;
    A[7][i] = A[3][i] + A[4][i] + A[16][i - 1];
    SEND_TO_S1;
    RECEIVE_FROM_S3;
    A[6][i + 1] = A[3][i + 1];
    SEND_TO_S3;
}

/* Loop epilogue */
RECEIVE_FROM_S16;
RECEIVE_FROM_S3;
RECEIVE_FROM_S4;
A[7][i] = A[3][i] + A[4][i] + A[16][i - 1];
SEND_TO_S1;
```

Figure 5: Schedule for processor 1, no communication costs.

```
for (i = proc_num; i <= N; i = i + 2)
{
    A[0][i] = B[i];
    A[8][i] = A[0][i];
    A[11][i] = A[8][i];
    A[10][i] = A[8][i];
    A[12][i] = A[11][i];
    SEND_TO_S5;
    RECEIVE_FROM_S12;
    A[5][i] = A[0][i] + A[12][i - 1];
    A[13][i] = A[12][i];
    A[15][i] = A[13][i];
    A[14][i] = A[13][i];
    SEND_TO_S9;
    A[16][i] = A[13][i];
    SEND_TO_S7;
    RECEIVE_FROM_S14;
    A[9][i] = A[8][i] + A[14][i - 1];
}
```

Figure 6: Schedule for flow-in statements.

```
/* Loop prologue */
A[2][i] = A[4][i - 1];
SEND_TO_S3;
A[1][i] = A[7][i - 1];
A[4][i] = A[1][i];

for (i = 1; i <= N - 1; i++)
{
    A[2][i + 1] = A[4][i];
    SEND_TO_S3;
    RECEIVE_FROM_S16;
    RECEIVE_FROM_S3;
    A[7][i] = A[3][i] + A[4][i] + A[16][i - 1];
    A[1][i + 1] = A[7][i];
    A[4][i + 1] = A[1][i + 1];
}

/* Loop epilogue */
RECEIVE_FROM_S16;
RECEIVE_FROM_S3;
A[7][i] = A[3][i] + A[4][i] + A[16][i - 1];
```

Figure 7: Schedule for processor 0, with communication costs.

```
/* Loop prologue */
RECEIVE_FROM_S2;
A[3][i] = A[2][i] + A[6][i - 1];
SEND_TO_S7;
A[6][i] = A[3][i];

for (i = 1; i <= N - 1; i++)
{
    RECEIVE_FROM_S2;
    A[3][i + 1] = A[2][i + 1] + A[6][i];
    SEND_TO_S7;
    A[6][i + 1] = A[3][i + 1];
}
```

Figure 8: Schedule for processor 1, with communication costs.

	Execution Time (μs)	Speedup (%)
Sequential	34067	
Assuming No Comm. Cost	25772	24
Assuming Comm. Cost	22571	38
Doacross	68037	-100

Table 6: Loop execution times and speedup on the Sequent

3.3 Test Results

To test the scheduling algorithm, a random loop generator was implemented. The loop generator takes four inputs: the number of statements in the loop, the total number of in-loop dependencies, the total number of loop carried dependencies, and the maximum number of dependencies for each statement. The loop generator uses a standard random number generator with a time-stamp seed and outputs the dependencies of the loop.

A code generator was used to translate the dependencies into actual statements. The statements were created so that the dependencies specified by the loop generator actually existed in the statements. This was done by putting the appropriate data values on the right hand side of an assignment statement. For example, if statement 5 was specified as having a loop-carried dependence on statement 7, and an in-loop dependence on statement 3, then the code generator would output statement 5 as, A[5][i] = A[7][i - 1] + A[3][i]. The execution time of this statement is estimated to be equal to the number of dependencies, in this case 2. Statements which were allocated no dependencies by the loop generator were assumed to read constants. For example, statement 3 might have the form, A[3][i] = B[i], with no dependencies. In this case, the execution time is assumed to be 1. The code generator was used in conjunction with the scheduler to automatically produce, schedule, and run various test loops.

By varying the parameters to the loop generator, loops with a variety of characteristics can be obtained. The question is, how to create loops which are similar to loops found in real programs. According to [9], loops in real programs can be characterized by the number of operands on the right hand side of assignment statements. The percentage of statements of each size found in loops in real programs is shown in Table 7. We chose our test parameters to create loops with similar characteristics. These are also shown in Table 7. The test loop parameters were permutations of the following: 20 and 25 statements, 10 and 15 in-loop dependencies, and 5 and 10 loop carried dependencies, with a maximum of 4 dependencies for each statement. Each combination of parameters was used to generate 4 different loops, for a total of 32 loops. Of the 32 loops, 10 had cyclic statements. Therefore, at least 22 of the loops were vectorizable.

For each loop, the sequential loop, the schedule produced by our algorithm, and one produced by a Doacross schedule were run for 360 iterations and timed. The total execution time and percent speedup are shown in Table 8 and graphed in Figure 9.

Although negative speedups were recorded for some of the loops scheduled using Doacross, in practice, expected speedup can be checked at compile time. Those loops which would have a

RHS Operands	Real Programs	Test Loops
1	80	77
2	15	16
3	3	5
4	2	2

Table 7: % of loop statements of various sizes

Loop #	Statements	In-Loop	Lcds	Sequential	PP	Doacross	PP	Doacross
loop0	20	10	5	39,674	14,117	62,420	64%	-57%
loop1	20	10	5	39,244	14,019	60,458	64%	-54%
loop2	20	10	5	38,622	14,071	36,716	64%	5%
loop3	20	10	5	37,536	12,422	24,900	67%	34%
loop4	20	10	10	41,634	16,630	59,599	60%	-43%
loop5	20	10	10	42,760	16,081	57,887	62%	-35%
loop6	20	10	10	44,420	17,641	51,978	60%	-17%
loop7	20	10	10	41,674	16,820	37,521	60%	10%
loop8	20	15	5	41,591	13,282	53,635	68%	-29%
loop9	20	15	5	46,381	14,883	51,584	68%	-11%
loop10	20	15	5	41,492	14,116	13,268	66%	68%
loop11	20	15	5	38,531	13,344	17,418	65%	55%
loop12	20	15	10	40,437	16,709	69,969	59%	-73%
loop13	20	15	10	42,626	47,630	72,954	-12%	-71%
loop14	20	15	10	50,530	31,266	76,702	38%	-52%
loop15	20	15	10	46,192	30,720	67,412	33%	-46%
loop16	25	10	5	39,635	14,998	50,319	62%	-27%
loop17	25	10	5	41,754	16,649	60,277	60%	-44%
loop18	25	10	5	42,759	19,397	41,664	55%	3%
loop19	25	10	5	45,579	15,699	52,873	66%	-16%
loop20	25	10	10	44,969	23,759	73,034	47%	-62%
loop21	25	10	10	46,354	19,168	56,848	59%	-23%
loop22	25	10	10	47,932	19,472	73,137	59%	-53%
loop23	25	10	10	46,790	18,205	27,232	61%	42%
loop24	25	15	5	47,393	15,914	16,187	66%	66%
loop25	25	15	5	45,819	27,918	58,697	39%	-28%
loop26	25	15	5	46,458	15,476	53,670	67%	-16%
loop27	25	15	5	48,915	16,269	37,620	67%	23%
loop28	25	15	10	53,843	29,842	81,972	45%	-52%
loop29	25	15	10	50,557	22,483	41,506	56%	18%
loop30	25	15	10	56,750	24,258	62,066	57%	-9%
loop31	25	15	10	56,697	20,972	76,251	63%	-34%

Table 8: Test loop parameters, execution times in μs and % speedup

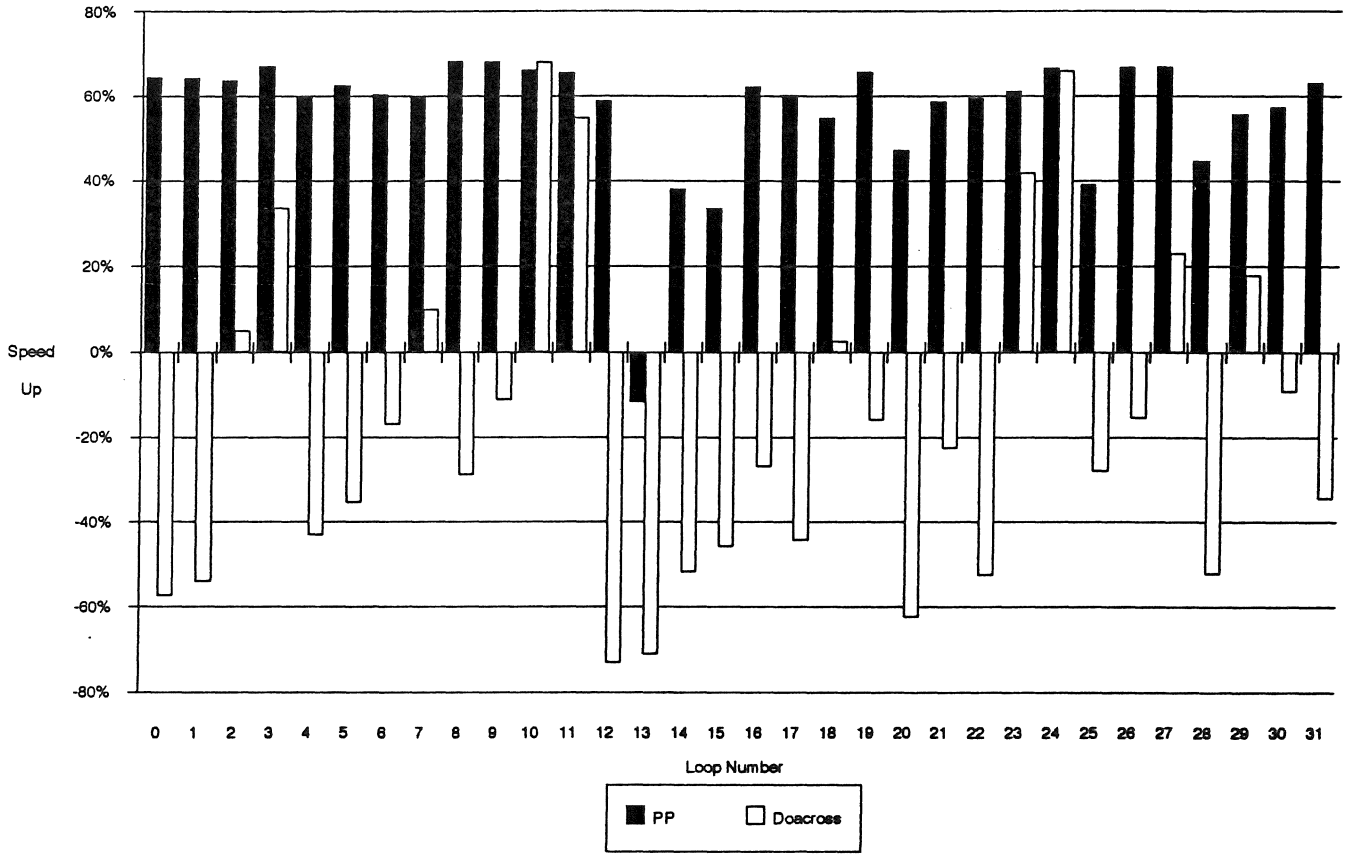

Figure 9: Test loop speedup in %, PP versus Doacross

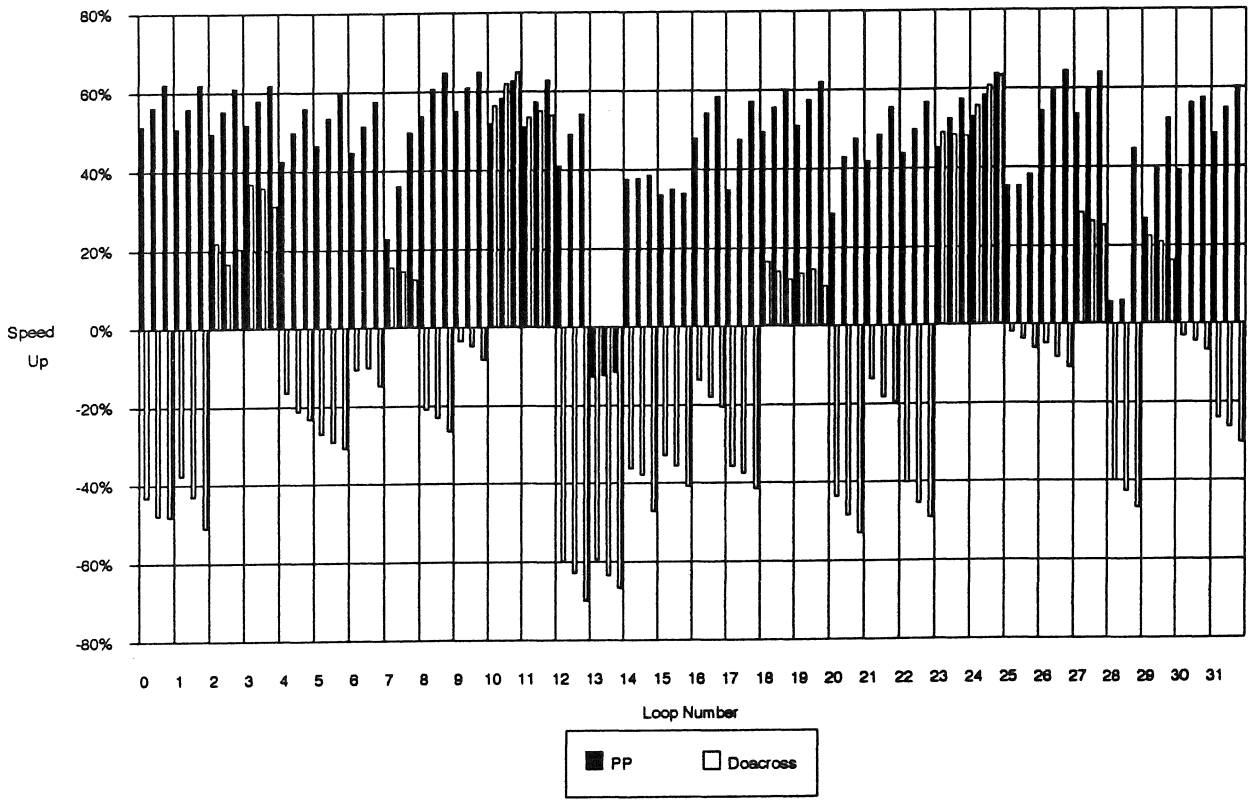

Figure 10: Test loop speedup in %, versus Doacross with 5, 6, and 7 processors

negative speedup would instead be run sequentially. By averaging the speedup over all of the test loops (weighted according to number of statements), we can estimate what the overall speedup would be for a real program. Even assuming that loops with negative speedup are instead run sequentially, the overall speedup for a program containing these 32 loops would be 57.2% with our technique, as opposed to 10.1% with Doacross.

In order to see the effect of limited resources on the loop execution time, we ran the same 32 loops on 5, 6, and 7 processors. The results are graphed in Figure 10. Most loops showed a steady incremental decrease in execution time as processors were increased. The exceptions were those loops whose ideal schedule already used less than 8 processors, which were therefore not affected by the processor limit.

4 Conclusions

In the past, loop scheduling on MIMD machines has consists primarily of assigned one or more loop iterations to each available processor, with synchronization added if necessary. This technique exploits any coarse grain (iteration level) parallelism available, but ignores any parallelism that may exist at a lower level. The scheduling algorithm presented in this paper uses fine-grain (statement level) parallelism techniques to try to exploit all of the parallelism present in the loop.

In addition, the algorithm adjusts the loop schedule according to the expected communication and synchronization costs of the target machine. Extra parallelism is exploited only when it is expected to improve the loop execution time.

A heuristic approach to rearranging statements *within* iterations to maximize Doacross overlap was presented in [6]. While interesting, this approach does not take into account communication costs, and is therefore not directly comparable with our technique. In the absence of communication costs, Perfect Pipelining is provably superior in exposing parallelism, and is therefore a better starting point when scheduling for MIMD machines.

We tested the algorithm on a synthetic benchmark of 32 loops. The loops were run on a Sequent, a shared memory MIMD machine, with 8 processors. The overall speedup of the loops was 57.2%. In addition, we ran the loops with resources restricted to 5, 6, and 7 processors, and found that the loop speedup tended to increase incrementally as the number of processors increased.

References

[1] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[2] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: a new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[3] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 1983 Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[4] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[5] R. G. Cytron. Doacross: beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[6] A.A. Munshi and B. Simons. Doacross: beyond vectorization for multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, June 1987.

[7] D.A. Padua. *Multiprocessors: Discussion of some theoretical and practical problems.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[8] C.D. Polychronopoulos and D.J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[9] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 3 edition, 1990.