Parallelizing Non-Vectorizable Loops for MIMD machines

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

Technical Report #90-01
January 1990

Parallelizing Non-Vectorizable Loops for MIMD machines

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

Abstract

Parallelizing a loop for MIMD machines can be described as a process of partitioning it into a number of relatively independent subloops. Previous approaches to partitioning non-vectorizable loops were mainly based on iteration pipelining which partitioned a loop based on iteration number and exploited parallelism by overlapping the execution of iterations. However, the amount of parallelism exploited this way is limited because the parallelism inside iterations has been ignored. In this paper, we present a new loop partitioning technique which can exploit both forms of parallelism - inside and across iterations. While inspired by the VLIW approach, our method is designed for more general, *asynchronous*, MIMD machines. In particular, our schedule takes the cost of communication into account, and attempts to balance it with respect to parallelism. We show our method is correct, efficient, and produces better schedules than previous iteration level approaches.

1 Introduction

To utilize the power of multiple processors in asynchronous MIMD machines, we need to decompose a task into parallel subtasks. Parallelization of a task could be done by the human programmer, or by a parallelizing compiler. Our interest is in the latter. The major concern of this paper is loop parallelization — partitioning a loop into a number of relatively independent subtasks. Loop partitioning is different from graph partitioning, in that the former deals with a potentially infinite graph due to the number of iterations which in general is not known at compile time. Since we assume non-vectorizable loops (implying the presence of *loop-carried dependences*), the problem is how to partition, efficiently, a graph which contains a number of arbitrarily long paths that are entangled together.

A dominant technique for loop parallelization (for non-vectorizable loops) is iteration pipelining, e.g., Dopipe [Padua79] or DOACROSS [Cytron86]. It partitions a loop based on indices; the destination processor of any operation is determined solely by its iteration number. A typical way of partitioning is interleaving: the indices are partitioned into p groups, where the i_{th} partition contains those iterations whose indices satisfy $(x \bmod p) = i$, where x is the iteration number. The subloops formed in this way are distributed to processors and executed concurrently. Of course, since dependences may exist between iterations, all such potential dependences need to be identified at compile time, and skewing between the parallel iterations needs to be introduced. This skewing can be obtained on asynchronous multiprocessors by inserting synchronization code at appropriate points. This synchronization will have some cost, and for the iteration pipelining technique to work well this cost should be relatively small. Such synchronization can be effectively achieved in a variety of machines, hence the popularity of this technique. However, it does not, in itself, attempt to optimize execution by taking into account communication cost. Furthermore, since the unit of scheduling is an iteration, all parallelism that might have existed inside iterations is ignored, and only the parallelism across iterations is being exploited.

Another technique for dealing with non-vectorizable loops is Perfect Pipelining [AiNi88a] [AiNi88b]. This technique was targeted for statically scheduled, synchronous architectures (e.g. VLIW's)[FiDo84], and, thus, its purpose is to find as many parallel operations as possible, regardless of iteration boundaries, to fill the long instruction word. When we assume zero communication/synchronization delay, the loop parallelization problems in MIMD's and VLIW's, become similar. Given enough processors for MIMD machines, and sufficient functional units in VLIW machines, the optimal schedule based on (compile time) data dependences for both architectures can be obtained by scheduling each operation at the earliest time it can be executed.

Since the number of iterations in the loop is, in general, not known at compile time, scheduling every operation at the earliest time it can be executed seems impossible. However [AiNi88a] has found that when every operation is scheduled as early as possible, the resulting schedule shows a repeating pattern.[1] An example of a pattern is found in Figure 3(b). It is obtained by sorting the graph in Figure 3(a) topologically subject to data dependences, which corresponds to scheduling the operations in the figure as early as possible. We underlined a set of repeating operations (with a finite difference in index value, 1 in this case), which we call a pattern. The importance of this pattern is that we can reproduce the optimal schedule of the loop merely by repeating its pattern. Perfect Pipelining is based on this concept of pattern. It identifies the pattern and replaces the loop body with it, yielding an optimal schedule (given compile time data dependences) for a multiprocessor with zero communication time and enough processors.

In this paper we extend the concept of pattern to the case of non-zero communication time. We prove that a pattern emerges in the resulting schedule even when each operation in the loop is assigned to the first available processor, that is, the first processor that can execute the operation at the earliest time, considering the cost of communication. This assignment destroys the ideal pattern of Perfect Pipelining due to the introduction of communication delays but we show that the resulting schedule produces a new pattern of its own. Thus our scheduling algorithm trades off parallelism and inter-processor communication in an effort to optimize overall performance on MIMD machines with non-zero communication time.

To conform with the inability of general purpose MIMD machines to execute multi-way jumps of the kind supported by VLIW's, we will assume the input loop is either without conditional statements or is if-converted [AlKe83]. This will also make comparison with conventional iteration based methods for MIMD machines meaningful, as this technique does not deal with in-loop conditional jumps.

The rest of the paper is organized as follows: Section 2 explains the scheduling technique we have developed and proves its correctness; Section 3 gives several examples to highlight various points in the scheduling process; Section 4 reports the results of experimentations we have performed to test the performance and robustness of our algorithm; and Section 5 summarizes our conclusions.

[1]More accurately each operation shows a repeating pattern (i.e., repeats with a fixed frequency). The details of how an overall pattern can be detected and the proof of its existence are in [AiNi88a].

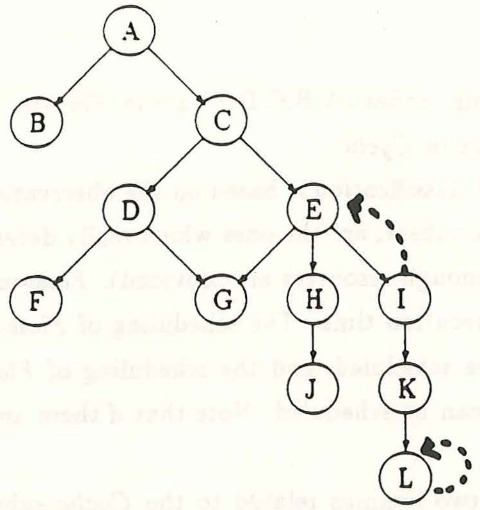


Figure 1: A classification example.

2 Scheduling in the presence of communication constraints.

2.1 Modeling the structure of a loop

Before we present our algorithm, we need to introduce our model of a loop. This model is useful in simplifying the following discussion.

We assume two things: the data dependence graph of the loop is a connected one, and all dependence distances are one or zero.[2] If the graph is not connected, we can simply separate the graph into several connected ones and apply our scheduling algorithm to each of them independently. Also if the dependence distances are greater than one, we can reduce them down to one or zero by unwinding the loop properly, as explained in [MuSi87].

A loop is viewed as a five-tuple, $<$ *V, E, Flow-in, Cyclic, Flow-out* $>$. *V* is a set of nodes, where a node represents a unit of computation — it could be a single operation or a whole procedure.[3] *E* is a set of two-tuples, $< v_1, v_2 >$, where each two-tuple represents a data dependence link from node v_1 to node v_2. Together, *V* and *E* define the data dependence graph for this loop. *Flow-in, Cyclic,* and *Flow-out* are disjoint subsets of *V* satisfying the following conditions: a node is in *Flow-in* if it has no predecessors or all of its predecessors are in *Flow-in*; a node is in *Flow-out* if it is not in *Flow-in*, and has no successors or all of its successors are in *Flow-out*; a node is in *Cyclic* if it is neither in *Flow-in* nor in *Flow-out.*[4]

[2]The precise definitions of data dependence graph and dependence distance used here conform to the standard ones as described in [Padua79].

[3]Granularity should be chosen depending on machines, to make the execution time of a node within the same order of magnitude as communication cost.

[4]So, to identify these subsets, the *Flow-in* subset should be identified first, then *Flow-out* subset, and then

In Figure 1, for example, nodes (A,B,C,D,F) are in *Flow-in*, nodes (G,H,J) are in *Flow-out*, and nodes (E,I,K,L) are in *Cyclic*.

The reason for this classification is based on the observation that the *Cyclic* nodes, nodes belonging to the *Cyclic* subset, are the ones which really determine the execution time of the given loop (assuming enough resources are provided). *Flow-in* and *Flow-out* nodes have little impact on the total execution time. The scheduling of *Flow-in* nodes is limited only by the latest time they can be scheduled, and the scheduling of *Flow-out* nodes is limited only by the earliest time they can be scheduled. Note that if there are no *Cyclic* nodes, the loop is a DOALL loop.

Below we present two lemmas related to the *Cyclic* subset which will be used later in Section 2.3.

Lemma 1. There is at least one strongly connected subgraph[5] in a *Cyclic* subset. (Examples of strongly connected subgraphs are (E,I) and (L) in Figure 1.)

Proof: If there is no strongly connected subgraph, there is no cycle in the *Cyclic* subset. This means all nodes in the *Cyclic* subset are *Flow-in* nodes by definition, because starting from the roots of the graph we can classify all nodes as *Flow-in* nodes. This is a contradiction since a *Cyclic* subset can not contain *Flow-in* nodes; therefore, a *Cyclic* subset contains at least one strongly connected component.

Lemma 2. For a loop which consists of a single *Cyclic* subset, unwinding it m times, there exists a path of length at least $m-1$.

Proof: Since there is at least one strongly connected subgraph in the original graph by Lemma 1, unwinding it m times, we should have a path of length at least $m-1$.

The algorithm *classification*, in Figure 2 is used to identify each subset. Its time complexity is $O(m)$, where m is the number of dependence links in the input data dependence graph, because each edge (i.e., dependence link) in the input graph can not be visited more than once. In terms of N, the number of nodes, it is $O(N^2)$ in the worst case.

2.2 Algorithm

The basic strategy of our algorithm is to extract the *Cyclic* nodes from the loop, which form the central part of the schedule, and schedule them utilizing the concept of pattern, and then

Cyclic subset. Also, since we don't deal with conditional jumps inside the loop in any special way, we ignore them in the scheduling process, and thus a data dependence graph alone is enough to represent the loop unambiguously.

[5]A strongly connected graph is one in which every node can be reached from every other node.

```
Algorithm. classification
Input.     Data Dependence Graph of a loop
Output.    Flow-in, Cyclic, Flow-out subsets of the loop
Method.
           0. Flow-in = Cyclic = Flow-out ={}
           1. buffer1 = {nodes which have no predecessors}.
           2. If buffer1 is empty, go to 5.
              Else add the nodes in buffer 1 to Flow-in.
           3. buffer 2 = {}.
              For each node x in buffer1
                for each successor of x
                    if  all predecessors of x are in Flow-in
                        include it in buffer 2.
           4. buffer 1 = buffer 2. go to 2.
           5. buffer 1 = {nodes which are not in Flow-in and have no successors}.
           6. If buffer 1 is empty, go to 9.
              else add the nodes in buffer 1 to Flow-out.
           7. buffer 2 = {}
              For each node x in buffer1
                for each predecessor of x
                     if all successors of x are in Flow-out
                        include it in buffer 2.
           8. buffer 1 = buffer 2. go to 6.
           9. Cyclic = {nodes which are not in Flow-in nor in Flow-out}.
```

Figure 2

include the schedule of non-*Cyclic* nodes. For now, suppose we have a loop which contains only *Cyclic* nodes (see Figure 3(a)).

The natural schedule that DOACROSS will produce for this loop is in the left two columns of Figure 3(c). However we can produce a better schedule as shown in the right two columns of the same figure.[6] There we are exploiting parallelism inside as well as across iterations, while in DOACROSS only the latter form of parallelism is exploited. The issue is can we exploit both forms of parallelism in the presence of a large or unknown loop bound, while factoring in communication cost?

As pointed out in the introduction section, our approach is based on a generalization of the concept of pattern first developed in [AiNi88a]. Our algorithm utilizes the concept of pattern in two ways. It first obtains the idealized pattern of Perfect Pipelining which does not take into account communication delays. Then, it schedules the nodes in the pattern one by one[7] to the processor which can execute it at the earliest time, when taking into account not only operational latencies, but also communication cost.[8] By doing this, we are distorting (skewing) the idealized pattern to accommodate communication cost. Since the skewing we introduce is based on consistent (fixed) communication cost estimates, we expect that another pattern will emerge from the resulting schedule.

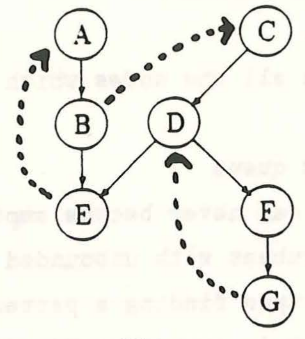
In the right two columns of Figure 3(c), we show one example of such a pattern emerging (enclosed with a box in the figure). In Section 2.3 we prove the existence of such patterns in general.

The algorithm for scheduling the *Cyclic* subset is in Figure 4. Its time complexity is $O(M * P * N^2 + M^3 * N^3)$, where M is the expected number of unrollings to find a pattern, N is the number of operations in the loop body, and P is the number of processors. We have a total of $M * N$ nodes to schedule. Most of the computing time is consumed in step 2. Its first sub-step is processor selection, the second pattern detection, and the third executable nodes collection. For each node, v, finding the destination processor takes $O(N * P)$, because in the worst case we have to compute P T(v,pj)'s, one for each processor, and computing a T(v,pj)

[6]In Figure 3(c) the subscripts show the iteration numbers. That is, A_0 implies an instance of A from iteration 0. In this example the execution time of each node and the cost of communication are both assumed to be one cycle.

[7]Since the (idealized) pattern shows only a partial ordering of nodes due to topological sorting, we need to enforce a fixed order for each set of parallel nodes in it to ensure the emergence of a new pattern. Any ordering (e.g., lexicographical ordering) is acceptable as long as it is consistent.

[8]In actual implementation, as can be seen later in algorithm *Cyclic-sched*, these two steps, finding an idealized pattern and scheduling it, are not separated. In algorithm *Cyclic-sched*, the data dependence graph of the loop is topologically traversed while at the same time each node visited is being scheduled.



(a)

$\underline{C_0A_0D_0B_0F_0E_0G_0}\,\underline{C_1A_1D_1B_1F_1E_1G_1}\,\underline{C_2A_2D_2B_2F_2E_2G_2}\ldots$

(b)

step	PE0	PE1	PE0	PE1
0	A_0		C_0	A_0
1	B_0		D_0	B_0
2	C_0		F_0	C_1
3	D_0		G_0	E_0
4	E_0		D_1	A_1
5	F_0		F_1	B_1
6	G_0	A_1	G_1	E_1
7		B_1	C_2	A_2
8		C_1	D_2	B_2
9		D_1	F_2	C_3
10		E_1	G_2	E_2
11		F_1	D_3	A_3
12	A_2	G_1	F_3	B_3
13	B_2		G_3	E_3
14	C_2		.	.
15	D_2		.	.
16	E_2		.	.
17	F_2			
18	G_2	A_3		
19		B_3		
20		C_3		
21		D_3		
22		E_3		
23		F_3		
23		G_3		

(c)

Figure 3: A scheduling example.

```
Algorithm. Cyclic-sched.
Input.     Data dependency graph of the Cyclic subset.
Output.    A schedule for the Cyclic subset.
Method.    1. Initialize the task queue with all the nodes which do not
              have predecessors.
           2. For  each node, v, in the task queue
                /* Note that the task queue can never become empty since we
                   are scheduling a Cyclic subset with unbounded unwinding.
                   So, this loop exits only upon finding a pattern */
                schedule  it to Pi, processor i, such that T(v,Pi) is the first
                   minimum in the list (T(v,P1), ... , T(v,Pp)), where p is the
                   total number of processors, and T(v,Pj) is the cycle in
                   the resulting schedule that v would be scheduled if it is
                   assigned to Pj.
                Check if a pattern has emerged.
                   If pattern found, exit.
                /* A pattern can be detected by checking if there is a
                   configuration repeating. The meaning of this configuration
                   and the proof that it correctly signals the emergence of a
                   pattern will be given in Section 2.3. */
                For  each successor w of v,
                   decrease the number of predecessors by one.
                     if number of predecessors for w = 0
                        add w to the task queue
                endfor
             endfor
```

Figure 4

```
Algorithm. Flow-in-sched.
Input.     Flow-in subset.
Output.    A schedule for Flow-in subset.
Method.
           1. Prepare p = Ceiling(L/H) free processors,
                  where
                      L is the size of the Flow-in subset, and H is the height
                      of the pattern obtained from algorithm Cyclic-sched. Call
                      them 0th, 1st,...,(p-1)th processor, each.
           2. For each iteration, i,
                     assign the Flow-in subset of iteration i to (i mod p)th processor.
             endfor.
```

Figure 5

takes $O(N)$ since we need to look at all the predecessors of node v, the number of which is bounded by N. Collecting executable nodes into the task queue also takes $O(N)$ since in the worst case, again, the node can have N successors. Therefore, the time complexity of the first and third sub-step is $O(M * P * N^2)$). However, in real loops, most nodes have only small numbers of successors/predecessors, which allows us to reduce the time complexity for these two sub-steps to $O(M * P * N)$ in realistic situation.

For each node scheduled, we check whether a pattern has been formed (the second sub-step). The number of nodes to inspect is $O(x^2)$, where x is the number of already scheduled nodes at the time of inspection. Since x could range from 0 to $M * N$, the time complexity for pattern detection is $0(M^3 * N^3)$. However, again, this is a worst case scenario. M is typically very small, less than 10 in all the examples we ran (see Section 3 and 4). Also, there is no need to check the pattern from the beginning of the scheduling process. By detecting the pattern after the schedule is stabilized, we can reduce the time complexity for pattern detection considerably. In fact, for all the examples in Section 3 and 4, the behaviour of the algorithm for pattern detection approached to $O(N)$.

The scheduling algorithm for *Flow-in* subset is in Figure 5, and, the final scheduling algorithm is in Figure 6, where algorithm *Flow-out-sched* is virtually the same as *Flow-in-sched.*

```
Algorithm.
Input       Data Dependence Graph of a loop.
Output      A schedule for it.
Method.
            1. Identify Flow-in, Cyclic, and Flow-out subsets (using algorithm
               classification).
            2. Schedule the Cyclic subset (using Cyclic-sched).
            3. Schedule the Flow-in subset (using Flow-in-sched).
            4. Schedule the Flow-out subset (using Flow-out-sched).
```

Figure 6

2.3 Proofs

Algorithm *Cyclic-sched* can terminate successfully only if a pattern is detected in the resulting schedule. We now prove the existence of that pattern. We assume that the number of processors, p, is sufficient to accommodate the resulting schedule, and the largest communication cost is k—each communication edge can have a different cost, but k is the upper bound of this cost.[9]. Note that since we are proving the correct termination of algorithm *Cyclic-sched*, we only need to look at the *Cyclic* nodes of a loop.

The proof can be visualized by imagining an infinite schedule resulting from full unwinding and a window drawn on it, with width p and height $k + 1$. We will refer to the portion of the schedule surrounded by it as a configuration. We slide the window down the schedule and watch the configuration in it changing. If we find a configuration that has been observed before, we stop the sliding, locate the position of the previous twin configuration, draw another window on it, and restart the sliding but this time with two windows at the same speed. If we see the two windows show the same sequence of configurations as they slide down, we know we have found a pattern. So, the proof consists of two things: first, we prove that there exist two distinct configurations which are identical, and then that once two configurations are identical, the following two configuration sequences after them should be

[9]This second assumption is only used in the process of scheduling. It does not in any way affect the correctness of the execution of the resulting schedule. In fact, as we will see in Section 3, the actual execution time can vary quite dramatically from that assumed in the scheduling process.

the same.

Definition 1. A shifted form of a set of nodes, $(n_0, n_1, \ldots, n_k)$, by d is the same set with the indices shifted by d, $(n_d, n_{1+d}, \ldots, n_{k+d})$.

Definition 2. Two configurations are identical if the set of nodes for one is a shifted form of the other, and the schedules for them are exactly the same.

Lemma 3. Any two nodes, v and w, if they are in the same configuration, should be within a finite number of iterations from each other. That is, if v is from iteration i, and w from iteration j, then $|i - j|$ is finite.

Proof: Suppose $d = |i - j|$ is arbitrarily large. We assume i is smaller than or equal to j, without loss of generality. From Lemma 2, we know there is a path of length at least d from iteration i to iteration j. Let the start node of this path be V, and the end node of it W. Also let the cycle of V in the resulting schedule be t_V, and that of W be t_W. Since d is arbitrarily large, the number of cycles between V and W, $t_W - t_V$, in the resulting schedule should be arbitrarily large, too. This means the number of cycles between v and w in the resulting schedule is also arbitrarily large because they are from the same iteration as V and W respectively, and thus should be scheduled within a finite number of cycles from V and W each.[10] This is a contradiction because since v and w are in the same configuration, the number of cycles between them in the schedule can not be greater than k.

Lemma 4. The number of non-identical configurations in the schedule is finite.

Proof: Again imagine a window sliding down the schedule. We will prove that the number of non-identical configurations that this window can show is finite. From Lemma 3, we know the number of consecutive iterations that any configuration can contain is bounded by some number, say M. Suppose at some point our window selects its nodes from iterations $(i_{1+d}, \ldots, i_{M+d})$, where d is an offset. We observe that the set of configurations that this window can possibly show from iterations $(i_{1+d}, \ldots, i_{M+d})$ is exactly the same as that it would from iterations $(i_1, \ldots, i_M)$, because every configuration from the former iterations has an identical matching configuration from the latter with a shifting distance d (see Definition 1 and 2). By generalizing, this means the

[10]Any two nodes with the longest path between them having a length of l, should be scheduled within $(k+1)l$ cycles from each other, assuming a sufficient number of processors. Obviously, the longest paths between v and V and between w and W both have finite lengths since we assume the original data dependence graph is a connected one.

kinds of configurations that this window can show is limited by the kinds that iterations $(i_1, \ldots, i_M)$ can supply. Since the number of nodes in this iteration range is finite, and the size of the configuration window is finite too, so is the number of configurations that this window can show; therefore, the total number of non-identical configurations in the schedule is finite.

Lemma 5. There exist two identical configurations separate in location in the resulting schedule.

Proof: Imagine a configuration window is sliding down this schedule. As the window slides down, the contents in it will change, but from Lemma 4 the number of possible configurations that it can show is finite; therefore, eventually it will repeat some configuration which has appeared before.

Lemma 6. If two configurations are identical, the two respective following configurations are identical, too.

Proof: Let the two configurations be C and D, and the schedule lines[11] right after each be l_1 and l_2 (see Figure 9(c) for an example). First we observe that any node in l_1 should have at least one of its direct predecessors in configuration C (the same thing can be said for the nodes in l_2 with respect to configuration D) . Otherwise its direct predecessors are all located before configuration C, and, therefore, it should have been scheduled within configuration C or before it.[12] This, in turn, means that the nodes that can come in l_1 and l_2 are among the direct successors of the nodes in configuration C and D respectively. Let the set of direct successors of the nodes in configuration C be S_C, and that of the nodes in configuration D be S_D. Since C and D are identical, S_C and S_D also should be identical. This means that the algorithm, right after the completion of configuration C, will look at the same sequence of nodes to schedule as it will after it has completed configuration D, as far as the schedule of line l_1 and l_2 is concerned. Then since the schedule in line l_1 and l_2 is completely determined by the configuration C and D respectively, the schedules of the two lines should be the same.

Since the schedules of l_1 and l_2 are identical, by moving the two surrounding windows for configuration C and D one cycle down, we can see two identical succeeding

[11] A schedule line is the schedule of all processors at some fixed cycle.

[12] A node can always be executed within $k+1$ cycles after its last direct predecessor is executed because we assume a sufficient number of processors. Note that k is the largest possible communication time. Since the height of a configuration is $k+1$, a node all of whose predecessors have been executed before the configuration should be executable at least at the bottom (schedule line) of the configuration.

configurations.

Lemma 7. If two configurations are identical, the sequences of configurations following them are same.

Proof: Let the two configurations be C_0 and D_0 and the following sequences C_i and D_i ($i >= 1$). The proof is by induction. If C_{i-1} and D_{i-1} are identical, we can say C_i and D_i are identical from Lemma 6. Since C_0 and D_0 are identical, through induction, we know C_i and D_i are identical for all i.

Theorem 1. *Cyclic-sched* produces a schedule which shows a pattern.

Proof: From Lemma 5, a configuration eventually repeats itself. Once it is repeated, it keeps appearing regularly by Lemma 7; so, we have a repeating pattern between the first and (not including) second configuration.[13]

3 Examples

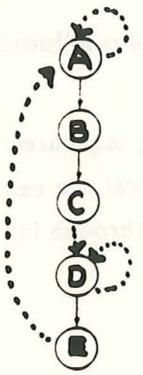
The first example (see Figure 7(a)-(e) and Figure 8(a) and 8(b)) shows the nontriviality of loop partitioning. The code is given in Figure 7(a), and its data dependence graph is in Figure 7(b). We note there is only one kind of node, *Cyclic*. The latency vector lv shows the estimated execution time of the nodes. Figure 7(c) shows the topological sorting of the nodes. By scheduling each node from this list one by one, with the communication time ($k = 2$, in this example) taken into consideration, we get Figure 7(d). Here, we can see that each processor is repeating some pattern of its own; and in effect, each iteration is completed every three cycles. Finally in Figure 7(e), we show the transformed loop where the original loop is partitioned into two subloops. DOACROSS will produce the schedule in Figure 8(a); it is the same as the schedule of a sequential execution (by collapsing the columns in the figure into PE0 column and removing all empty cycles) because no pipelining is possible due to the (E,A) dependence link. Even with an optimal reordering, as in Figure 8(b) which is obtained by an exhaustive search[14], DOACROSS would still yield no performance improvement, in this case, since no parallelism is achievable at the iteration level when synchronization cost is taken into account. The percentage parallelism obtained for this example, which we define as in [Cytron84] to be $s_p = (s - p/s) * 100$, where s and p are sequential and parallel execution time respectively, is 40 by our algorithm, while that by DOACROSS is 0.

[13] Note the difference between a configuration and a pattern. In Figure 9(c), for example, window C shows a configuration, while the pattern is enclosed by a box with height 6 below it.

[14] In general, optimal reordering of nodes is NP-hard [Cytron86][MuSi87].

```
FOR I = 1 TO N
A: A[I] = A[I-1] + E[I-1]
B: B[I] = A[I]
C: C[I] = B[I]
D: D[I] = D[I-1] + C[I-1]
E: E[I] = D[I]
ENDFOR
```

(a)



lv = (1,1,1,1,1) for nodes A,B,C,D,E in the same order.

(b)

$$\underline{A_1D_1B_1E_1C_1D_2A_2E_2B_2C_2}\,\underline{A_3D_3B_3E_3C_3D_4A_4E_4B_4C_4}A_5D_5B_5\ldots$$

(c)

step	PE0	PE1
0	A_1	D_1
1	B_1	E_1
2	C_1	
3	D_2	A_2
4	E_2	B_2
5		C_2
6	A_3	D_3
7	B_3	E_3
8	C_3	
9	D_4	A_4
10	E_4	B_4
11		C_4
12	A_5	D_5
13	B_5	E_5
14	.	
15	.	

(d)

```
PARBEGIN  (N IS ASSUMED TO BE AN EVEN NUMBER.)

PE0: A[1] = A[0] + E[0]
      (SEND A[1] TO PE1)
     B[1] = A[1]
     C[1] = B[1]
      (RECEIVE D[1] FROM PE1)
     D[2] = D[1] + C[1]
      (SEND D[2] TO PE1)
     E[2] = D[2]
     FOR I1 = 3 TO N-1 BY 2
         (RECEIVE A[I1-1] FROM PE1)
        A[I1] = A[I1-1] + E[I1-1]
         (SEND A[I1] TO PE1)
        B[I1] = A[I1]
        C[I1] = B[I1]
         (RECEIVE D[I1] FROM PE1)
        D[I1+1] = D[I1] + C[I1]
         (SEND D[I1+1] TO PE1)
        E[I1+1] = D[I1+1]
     ENDFOR

PE1: D[1] = D[0] + C[0]
      (SEND D[1] TO PE0)
     E[1] = D[1]
      (RECEIVE A[1] FROM PE0)
     A[2] = A[1] + E[1]
      (SEND A[2] TO PE0)
     B[2] = A[2]
     C[2] = B[2]
     FOR I2 = 3 TO N-1 BY 2
         (RECEIVE D[I2-1] FROM PE0)
        D[I2] = D[I2-1] + C[I2-1]
         (SEND D[I2] TO PE0)
        E[I2] = D[I2]
         (RECEIVE A[I2] FROM PE0)
        A[I2+1] = A[I2] + E[I2]
         (SEND A[I2+1] TO PE0)
        B[I2+1] = A[I2+1]
        C[I2+1] = B[I2+1]
     ENDFOR
 PAREND
```

(e)

Figure 7: A non-trivial scheduling example.

step	PE0	PE1	PE3	PE4
0	A_1			
1	B_1			
2	C_1			
3	D_1			
4	E_1			
5				
6		A_2		
7		B_2		
8		C_2		
9		D_2		
10		E_2		
11				
12			A_3	
13			B_3	
14			C_3	
15			D_3	
16			E_3	
17				.
18				.
19				.

(a)

step	PE0	PE1	PE3	PE4
0	A_1			
1	B_1			
2	D_1			
3	E_1			
4	C_1			
5				
6		A_2		
7		B_2		
8		D_2		
9		E_2		
10		C_2		
11				
12			A_3	
13			B_3	
14			D_3	
15			E_3	
16			C_3	
17				.
18				.
19				.

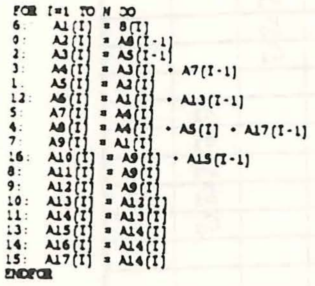
(b)

Figure 8: Schedules by DOACROSS for Figure 7(b). Compare it with Figure 7(d).

The second example is from [Cytron86] (see Figure 9(a)-(c) and Figure 10). As in the first example, we show the code, data dependence graph, and the schedule. However, in step 1 of our algorithm, the *Flow-in* buffer will contain nodes (6,7,8,9,10,11,12,13,14,15,16). There are no *Flow-out* nodes. The rest of the nodes are all *Cyclic*, as determined by algorithm *classification*. Note that the latency of the operations is not unique. Using algorithm *Cyclic-sched*, we can generate Figure 9(c).[15] We can see processor 0 is repeating node 3 and 5, while processor 1 is repeating node 0,1,2, and 4. Again we assume the communication time is $k = 2$ arbitrarily. Figure 10 shows the final transformed loop after the *Flow-in* nodes are distributed into three processors, and synchronization code inserted. In algorithm *Flow-in-sched*, for this case, L, the size of the *Flow-in* subset is 11, and H, the height of the pattern from algorithm *Cyclic-sched* is 6. Therefore, p, the number of needed free processors, is 3. In the result, we have partitioned the original loop into five subloops. The *Flow-in* nodes are distributed to three processors so as not to delay the execution time of the *Cyclic* subset. For this case, the percentage parallelism obtained by our algorithm is 72.7%, and that by DOACROSS is 31.8%.

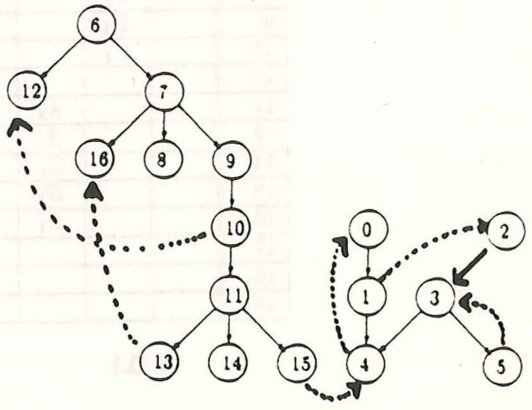
We give two more examples, one from the 18th Livermore Loop (Figure 11(a)-(d)), the other from a fifth order elliptic filter [PaKn89] (Figure 12(a) and Figure 12(b)). Figure 11(a) is the original data dependence graph for the first example. We extracted *Cyclic* nodes from

[15]In Figure 9(c), each node is represented by two things: its name and its iteration number. For example, (3,11) means the instance of node 3 from iteration 11.



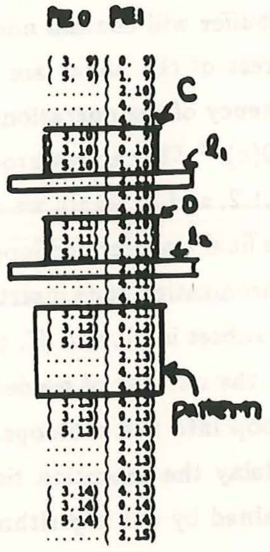
```
FOR I=1 TO N DO
6:    A1(I)  = B(I)
0:    A2(I)  = A8(I-1)
2:    A3(I)  = A5(I-1)
3:    A4(I)  = A3(I)  * A7(I-1)
1.    A5(I)  = A2(I)
12:   A6(I)  = A1(I)  * A13(I-1)
5:    A7(I)  = A4(I)
4:    A8(I)  = A4(I)  * A5(I)  * A17(I-1)
7:    A9(I)  = A1(I)
16:   A10(I) = A9(I)  * A15(I-1)
8:    A11(I) = A9(I)
9:    A12(I) = A9(I)
10:   A13(I) = A12(I)
11:   A14(I) = A13(I)
13:   A15(I) = A14(I)
14:   A16(I) = A14(I)
15:   A17(I) = A14(I)
ENDFOR
```

(a)



lv = (1,1,1,2,3,1,1,1,1,1,1,1,1,2,1,1,1,2) in the order of the labels.

(b)



```
( 3, 9)( 0, 9)
( 5, 9)( 1, 9)
.......  2,10)
.......  4, 9)
.......  4, 9)
( 3,10)( 4, 9)
( 3,10)( 0,10)
( 5,10)( 1,10)
.......  2,11)
.......  4,10)
.......  4,10)
( 3,11)( 4,10)
( 3,11)( 0,11)
( 5,11)( 1,11)
.......  2,12)
.......  4,11)
.......  4,11)
( 3,12)( 4,11)
( 3,12)( 0,12)
( 5,12)( 1,12)
.......  2,13)
.......  4,12)
.......  4,12)
( 3,13)( 4,12)
( 3,13)( 0,13)
( 5,13)( 1,13)
.......  2,14)
.......  4,13)
.......  4,13)
( 3,14)( 4,13)
( 3,14)( 0,14)
( 5,14)( 1,14)
.......  2,15)
```

(c)

Figure 9: An example from [Cytron86].

```
PARBEGIN    (N IS ASSUMED TO BE A MULTIPLE OF 3)
PE0  FOR I0 = 1 TO N
        (RECEIVE A3(I0) FROM PE1)
        A4(I0) = A3(I0) + A7(I0-1)
        (SEND A4(I0) TO PE1)
        A7(I0) = A4(I0)
     ENDFOR

PE1: A3(1) = A5(0)
     (SEND A3(1) TO PE0)
     A2(1) = A8(0)
     A5(1) = A2(1)
     FOR I1 = 2 TO N
        A3(I1) = A5(I1-1)
        (SEND A3(I1) TO PE0)
        (RECEIVE A4(I1-1) FROM PE0 AND A17(I1-2) FROM PE2,3,OR 4, EXCEPT A17(0))
        A8(I1-1) = A4(I1-1) + A5(I1-1) + A17(I1-2)
        A2(I1) = A8(I1-1)
        A5(I1) = A2(I1)
     ENDFOR
     (RECEIVE A4(N) FROM PE0 AND A17(N-1) FROM PE2, 3, OR 4)
     A8(N) = A4(N) + A5(N) + A17(N-1)

PE2: FOR I2 = 1 TO N-2 BY 3
        A1(I2) = B(I2)
        A9(I2) = A1(I2)
        A11(I2) = A9(I2)
        A12(I2) = A9(I2)
        A13(I2) = A12(I2)
        (SEND A13(I2) TO PE3)
        A14(I2) = A12(I2)
        (RECEIVE A13(I2-1) FROM PE4)
        A6(I2) = A1(I2) + A13(I2-1)
        A15(I2) = A14(I2)
        (SEND A15(I2) TO PE3)
        A16(I2) = A14(I2)
        A17(I2) = A14(I2)
        (SEND A17(I2) TO PE1)
        (RECEIVE A15(I2-1) FROM PE4)
        A10(I2) = A9(I2) + A15(I2-1)
     ENDFOR

PE3: FOR I3 = 2 TO N-1 BY 3
        A1(I3) = B(I3)
        A9(I3) = A1(I3)
        A11(I3) = A9(I3)
        A12(I3) = A9(I3)
        A13(I3) = A12(I3)
        (SEND A13(I3) TO PE4)
        A14(I3) = A12(I3)
        (RECEIVE A13(I3-1) FROM PE2)
        A6(I3) = A1(I3) + A13(I3-1)
        A15(I3) = A14(I3)
        (SEND A15(I3) TO PE4)
        A16(I3) = A14(I3)
        A17(I3) = A14(I3)
        (SEND A17(I3) TO PE1)
        (RECEIVE A15(I3-1) FROM PE2)
        A10(I3) = A9(I3) + A15(I2-1)
     ENDFOR

PE4: FOR I4 = 3 TO N BY 3
        A1(I4) = B(I4)
        A9(I4) = A1(I4)
        A11(I4) = A9(I4)
        A12(I4) = A9(I4)
        A13(I4) = A12(I4)
        (SEND A13(I4) TO PE2)
        A14(I4) = A12(I4)
        (RECEIVE A13(I4-1) FROM PE3)
        A6(I4) = A1(I4) + A13(I4-1)
        A15(I4) = A14(I4)
        (SEND A15(I4) TO PE2)
        A16(I4) = A14(I4)
        A17(I4) = A14(I4)
        (SEND A17(I4) TO PE1)
        (RECEIVE A15(I4-1) FROM PE3)
        A10(I4) = A9(I4) + A15(I4-1)
     ENDFOR

PAREND
```

Figure 10: The parallelized loop for Figure 9(a).

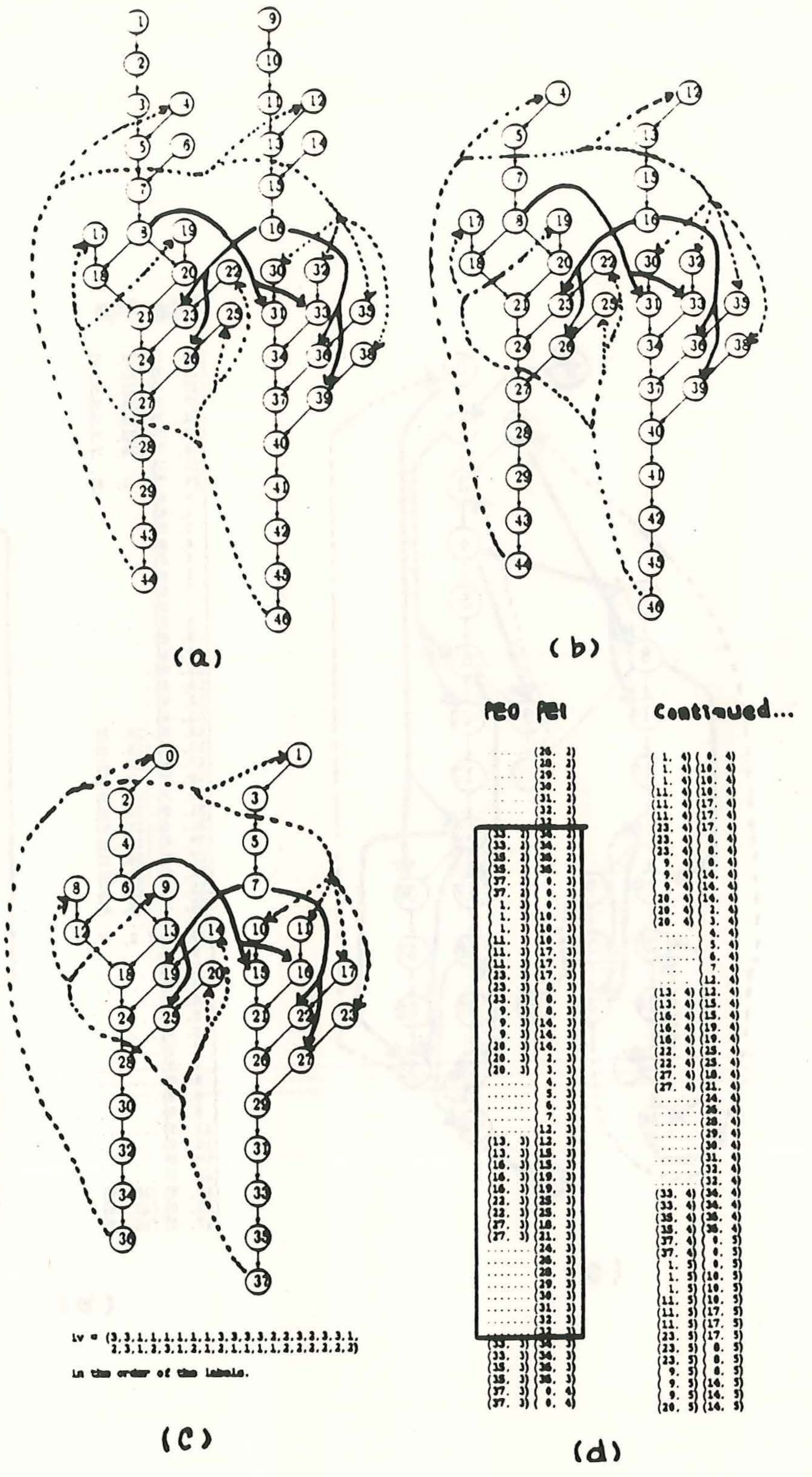
it resulting in Figure 11(b), and re-labeled the nodes as shown in Figure 11(c). The schedule is shown in Figure 11(d) with the pattern enclosed with a box. Figure 12(a) is the data dependence graph for the second example, and Figure 12(b) is its schedule for the *Cyclic* nodes. In both cases, most of the nodes are in *Cyclic*; for the first example, only 8 nodes, (1,2,3,6,9,10,11,14) in Figure 11(a), are *non-Cyclic* nodes (they are *Flow-in* nodes), while in the second, only node 34 is a *non-Cyclic* node (a *Flow-out* node). In such cases, scheduling *non-Cyclic* nodes separately may cause low processor utilization. One way of avoiding this waste is to schedule these *non-Cyclic* nodes into one of the relatively idle processors, processor 0 in the first example and processor 1 in the second one. For both cases, inclusion of *non-Cyclic* nodes can be achieved with only small amount of delay. The strategy is simple; after the schedule of *Cyclic* nodes is completed, if there is a relatively idle processor with idle time slots wide enough to accommodate the *non-Cyclic* nodes with little or no additional delay, combine the *non-Cyclic* nodes into the idle processor. This heuristic can be easily combined with our algorithm.

In both cases, the loops are partitioned into two relatively independent subloops (see Figure 11(d) and Figure 12(b)), and these partitionings are producing higher percentage parallelism than those of DOACROSS. The percentage parallelism achieved by our algorithm for each example are 49.4 and 30.9, while those by DOACROSS are 12.6 and 0. Again, we assumed $k = 2$, where k is the communication cost.

4 Experimentation

The above examples show superior results for our algorithm. However, we have assumed that the communication cost is fixed, which means there is no unpredictable fluctuation in communication time. Also, the dependence pattern in the examples may have favored our algorithm. To test the performance of our algorithm and its robustness under unstable communication traffic and complex dependence graphs thoroughly, we have generated 25 random loops and tested our algorithm under various traffic conditions.

The way we generated a random loop is as follows. First, we fixed the number of nodes in the loop as 40, and the number of loop carried dependences (*lcd's*) and simple dependences (*sd's*) at 20 each. The execution time of each node is randomly chosen from 1 to 3 cycles using a random number generator. Then, again using the random number generator, we generated actual dependence links, 20 for *lcd's* and another 20 for *sd's*. After this was done, we extracted only *Cyclic* nodes from the graph. The effect is that we have generated a random loop, which contains only *Cyclic* nodes whose latencies vary from 1 to 3 cycles, with less than



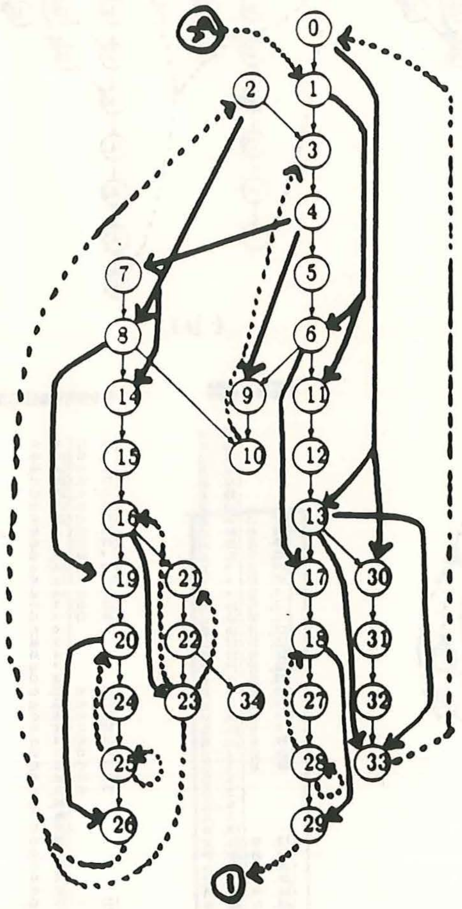
```
lv = (3,3,1,1,1,1,1,1,3,3,3,3,2,2,3,2,3,3,1,
      2,3,1,2,3,1,2,1,2,1,1,1,1,2,2,2,2,2,2)

in the order of the labels.
```

```
PE0      PE1             Continued...

........ (26.  2)        ( 1.  4) ( 0.  4)
........ (28.  2)        ( 1.  4) (10.  4)
........ (29.  2)        ( 1.  4) (10.  4)
........ (30.  2)        (11.  4) (10.  4)
........ (31.  2)        (11.  4) (17.  4)
........ (32.  2)        (11.  4) (17.  4)
........ (32.  2)        (23.  4) (17.  4)
(33.  2) (34.  2)        (23.  4) ( 8.  4)
(33.  2) (34.  2)        (23.  4) ( 8.  4)
(35.  2) (36.  2)        ( 9.  4) ( 8.  4)
(35.  2) (36.  2)        ( 9.  4) (14.  4)
(37.  2) ( 0.  3)        ( 9.  4) (14.  4)
(37.  2) ( 0.  3)        (20.  4) (14.  4)
( 1.  3) ( 0.  3)        (20.  4) ( 2.  4)
( 1.  3) (10.  3)        (20.  4) ( 3.  4)
( 1.  3) (10.  3)        ........ ( 4.  4)
(11.  3) (10.  3)        ........ ( 5.  4)
(11.  3) (17.  3)        ........ ( 6.  4)
(11.  3) (17.  3)        ........ ( 7.  4)
(23.  3) (17.  3)        ........ (12.  4)
(23.  3) ( 8.  3)        (13.  4) (12.  4)
(23.  3) ( 8.  3)        (13.  4) (15.  4)
( 9.  3) ( 8.  3)        (16.  4) (15.  4)
( 9.  3) (14.  3)        (16.  4) (19.  4)
( 9.  3) (14.  3)        (16.  4) (19.  4)
(20.  3) (14.  3)        (22.  4) (25.  4)
(20.  3) ( 2.  3)        (22.  4) (25.  4)
(20.  3) ( 3.  3)        (27.  4) (18.  4)
........ ( 4.  3)        (27.  4) (21.  4)
........ ( 5.  3)        ........ (24.  4)
........ ( 6.  3)        ........ (26.  4)
........ ( 7.  3)        ........ (28.  4)
........ (12.  3)        ........ (29.  4)
(13.  3) (12.  3)        ........ (30.  4)
(13.  3) (15.  3)        ........ (31.  4)
(16.  3) (15.  3)        ........ (32.  4)
(16.  3) (19.  3)        ........ (32.  4)
(16.  3) (19.  3)        (33.  4) (34.  4)
(22.  3) (25.  3)        (33.  4) (34.  4)
(22.  3) (25.  3)        (35.  4) (36.  4)
(27.  3) (18.  3)        (35.  4) (36.  4)
(27.  3) (21.  3)        (37.  4) ( 0.  5)
........ (24.  3)        (37.  4) ( 0.  5)
........ (26.  3)        ( 1.  5) ( 0.  5)
........ (28.  3)        ( 1.  5) (10.  5)
........ (29.  3)        ( 1.  5) (10.  5)
........ (30.  3)        (11.  5) (10.  5)
........ (31.  3)        (11.  5) (17.  5)
........ (32.  3)        (11.  5) (17.  5)
........ (32.  3)        (23.  5) (17.  5)
(33.  3) (34.  3)        (23.  5) ( 8.  5)
(33.  3) (34.  3)        (23.  5) ( 8.  5)
(35.  3) (36.  3)        ( 9.  5) ( 8.  5)
(35.  3) (36.  3)        ( 9.  5) (14.  5)
(37.  3) ( 0.  4)        ( 9.  5) (14.  5)
(37.  3) ( 0.  4)        (20.  5) (14.  5)
```

Figure 11: Scheduling the 18th Livermore Loop.



(a)

```
PE0 PE1                  continued...

(31, 7) (20, 7)          (30, 9) (19, 9)
(31, 7) (22, 7)          (18, 9) (21, 9)
(27, 7) (22, 7)          (31, 9) (20, 9)
(27, 7) (24, 7)          (31, 9) (22, 9)
(32, 7) (24, 7)          (27, 9) (22, 9)
(23, 7) (25, 7)          (27, 9) (24, 9)
(28, 7) (26, 7)          (32, 9) (24, 9)
(33, 7) .......          (23, 9) (25, 9)
(29, 7) ( 2, 8)          (28, 9) (26, 9)
( 0, 8) .......          (33, 9) .......
( 1, 8) .......          (29, 9) ( 2,10)
( 3, 8) .......          ( 0,10) .......
( 4, 8) .......          ( 1,10) .......
( 5, 8) .......          ( 3,10) .......
( 5, 8) .......          ( 4,10) .......
( 7, 8) .......          ( 5,10) .......
( 7, 8) .......          ( 5,10) .......
( 6, 8) .......          ( 7,10) .......
( 8, 8) .......          ( 7,10) .......
( 9, 8) .......          ( 6,10) .......
(11, 8) .......          ( 8,10) .......
(14, 8) .......          ( 9,10) .......
(10, 8) .......          (11,10) .......
(12, 8) .......          (14,10) .......
(12, 8) (15, 8)          (10,10) .......
(13, 8) (15, 8)          (12,10) .......
(17, 8) (16, 8)          (12,10) (15,10)
(30, 8) (19, 8)          (13,10) (15,10)
(18, 8) (21, 8)          (17,10) (16,10)
(31, 8) (20, 8)          (30,10) (19,10)
(31, 8) (22, 8)          (18,10) (21,10)
(27, 8) (22, 8)          (31,10) (20,10)
(27, 8) (24, 8)          (31,10) (22,10)
(32, 8) (24, 8)          (27,10) (22,10)
(23, 8) (25, 8)          (27,10) (24,10)
(28, 8) (26, 8)          (32,10) (24,10)
(33, 8) .......          (23,10) (25,10)
(29, 8) ( 2, 9)          (28,10) (26,10)
( 0, 9) .......          (33,10) .......
( 1, 9) .......          (29,10) ( 2,11)
( 3, 9) .......          ( 0,11) .......
( 4, 9) .......          ( 1,11) .......
( 5, 9) .......          ( 3,11) .......
( 5, 9) .......          ( 4,11) .......
( 7, 9) .......          ( 5,11) .......
( 7, 9) .......          ( 5,11) .......
( 6, 9) .......          ( 7,11) .......
( 8, 9) .......          ( 7,11) .......
( 9, 9) .......          ( 6,11) .......
(11, 9) .......          ( 8,11) .......
(14, 9) .......          ( 9,11) .......
(10, 9) .......          (11,11) .......
(12, 9) .......          (14,11) .......
(12, 9) (15, 9)          (10,11) .......
(13, 9) (15, 9)          (12,11) .......
(17, 9) (16, 9)          (12,11) (15,11)
```

(b)

Figure 12: Scheduling the fifth order elliptic filter.

presence of unpredictable communication cost.

5 Conclusion

In this paper we have presented a new technique to schedule non-vectorizable loops for MIMD machines which can produce higher percentage parallelism than conventional iteration-based pipelining techniques. We have proved our algorithm is correct and compared its performance against a conventional iteration-based pipelining technique. The results show that our approach can achieve higher performance, even when the estimation of communication cost is far off the mark, and the actual cost of communication is relatively high (7 times the basic node execution time). Thus our approach shows a great deal of robustness under adverse circumstances.

or equal to 40 nodes and less than or equal to 20 *lcd's* and *sd's*. We have repeated the same process with different seeds (1 to 25), producing 25 different loops.

For each loop generated, we have extracted only *Cyclic* nodes[16], and scheduled them using our algorithm and DOACROSS. The resulting schedules were executed on a simulated multiprocessor. We assumed fully overlapped communication, and the estimated communication time for our algorithm was $k = 3$ cycles. To model the fluctuation in the actual communication time and asynchrony by the processors, we used a varying factor mm. With this varying factor, the run time cost of each communication link varied between k and $k + mm - 1$. We compared our algorithm with DOACROSS under three different mm's: $mm = 1$ (no fluctuation), $mm = 3$ (maximum 67% of delay in communication time), and $mm = 5$ (maximum 130% of delay in communication time). Thus the schedule our algorithm produces is based on the estimated k, while at run time *all* communication takes $k + mm - 1$ cycles, clearly a worst case scenario.

The result of performance comparison is in Table 1(a). For each loop, we ran the simulated multiprocessor and measured the parallel execution time. By subtracting it from the sequential execution time and dividing the result by the sequential execution time, we calculated the percentage parallelism. The entry in Table 1(a) shows the percentage parallelism, obtained this way, for each loop. When $mm = 1$, our algorithm produced better schedules than DOACROSS in all loops. The average percentage parallelism of our algorithm is about a factor of 2.9 higher than that of DOACROSS. (See Table 1(b).) When $mm = 3$, in only one out of the 25 loops our algorithm produced a worse schedule than DOACROSS; when $mm = 5$, only two such loops out of the 25 loops. But in both cases, the average percentage parallelism of our algorithm are about a factor of 3.0 ($mm = 3$ case) and 3.2 ($mm = 5$ case) higher than those of DOACROSS as shown in Table 1(b). One thing to note is that $mm = 5$ implies the communication cost was underestimated by a factor of 2.3, which will happen only in a very unstable asynchronous traffic. Even under this unpredictable situation, our algorithm exploits more parallelism than DOACROSS on average. In fact, despite our expectation that our algorithm performance would worsen under such adverse conditions, Table 1(b) shows that in the presence of unstable communication cost, our relative performance versus DOACROSS actually improves (see *the factor of speedup over DOACROSS* in the table). This suggests that careful scheduling can be both robust and profitable in the

[16] Non-Cyclic nodes wouldn't increase parallel execution time in our case since the critical path in the schedule is formed only by the *Cyclic* nodes. The execution time in DOACROSS would not be delayed considerably by them either, if we properly separate them from *Cyclic* nodes through reordering of operations. Thus, we can put aside *non-Cyclic* nodes for the purpose of comparison between our algorithm and DOACROSS.

	$mm = 1$		$mm = 3$		$mm = 5$	
loop	x	doacross	x	doacross	x	doacross
0	57.8	26.8	51.3	23.7	45.2	18.6
1	36.1	0.0	26.0	0.0	15.2	0.0
2	55.8	38.7	50.9	33.0	45.8	27.9
3	41.2	19.0	34.2	13.8	26.6	8.0
4	68.5	11.4	62.1	9.2	55.7	7.2
5	39.8	10.5	28.5	6.8	18.2	0.8
6	48.6	16.9	40.9	12.4	33.2	8.3
7	42.0	14.2	30.8	9.2	15.2	6.2
8	65.7	40.7	60.5	37.7	56.7	33.1
9	21.2	15.3	6.0	11.3	0.0	7.1
10	48.5	15.7	44.1	13.4	39.4	8.6
11	56.0	31.1	52.3	27.5	47.8	24.2
12	66.0	20.0	61.4	16.2	57.1	11.1
13	55.6	10.5	47.7	7.7	36.4	4.6
14	36.6	31.1	32.3	28.3	23.3	26.9
15	44.3	13.1	31.6	10.4	22.0	5.9
16	34.1	0.0	22.5	0.0	12.8	0.0
17	36.1	11.5	25.0	8.5	13.8	5.5
18	56.7	11.7	43.5	7.5	30.0	2.9
19	36.4	25.3	30.3	21.2	18.7	17.3
20	47.3	0.0	39.2	0.0	29.8	0.0
21	42.9	18.8	30.6	14.1	16.7	8.7
22	34.4	3.7	29.2	1.2	17.7	0.0
23	49.3	9.6	41.9	5.8	34.5	1.1
24	61.3	11.1	52.7	6.5	44.1	2.2

(a)

	$mm = 1$	**$mm = 3$**	**$mm = 5$**
x	**47.4046**	**39.0674**	**30.2776**
DOACROSS	**16.3135**	**13.0623**	**9.4823**
Factor of speed-up over DOACROSS	**2.9**	**3.0**	**3.3**

(b)

Table 1: **Comparison of performance between our algorithm (denoted by x) and DOACROSS.**

References

[AiNi88a] Aiken, A. and Nicolau, A. 1988. Optimal loop parallelization. In Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, June.

[AiNi88b] Aiken, A. and Nicolau, A. 1988. Perfect Pipelining: A new loop parallelization technique. In Proceedings of the 1988 European Symposim on Programming. Springer Verlag Lecture Notes in Computer Science no. 300, March.

[AlKe83] Allen,J.R., Kennedy K., Porterfield, C. and Warren, J. 1983. Conversion of control dependence to data Dependence. In Proceedings of the 1983 Symposium on Principles of Programming Languages, pp. 177-189, January.

[Cytron84] Cytron, R.G. 1984 Compile-time Scheduling and Optimization for Asynchronous machines. PhD Thesis, University of Illinois at Urbana-Champagne.

[Cytron86] Cytron, R.G. 1986 Doacross: Beyond Vectorization for Multiprocessors. In Proceedings of the 1986 International Conference on Parallel Processing, St. Charles, IL, pp. 836-844, August.

[FiDo84] Fisher, J.A. and O'Donnell, J.J. 1984. VLIW machines: Multiprocessors we can actually program. In Proceedings of CompCon Spring 84, pp. 299-305. IEEE Computer Society, February.

[MuSi87] Munshi, A.A. and Simons, B. 1987. Scheduling Sequential Loops on Parallel Processors. In Proceedings of the 1987 International Conference on Parallel Processing, St. Charles, Illinois, August.

[Padua79] Padua, D.A. Multiprocessors: discussion of some theoretical and practical problems. PhD Thesis, University of Illinois at Urbana-Champagne.

[PaKn89] Paulin, P.G. and Knight, J.P. 1989. Force-directed scheduling for the Behavioral Synthesis of ASIC's. In IEEE transactions on Computer-Aided Design, Vol.8, No.6, June.