Scheduling DAG's for Asynchronous Multiprocessor Execution

Brian A. Malloy, Errol L. Lloyd, and Mary Lou Soffa

Abstract—A new approach is given for scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors. The key idea in our approach is to exploit the *fine grained parallelism* present in the instruction stream. In this context, schedules are constructed by a careful balancing of execution and communication costs at the level of individual instructions, and their data dependencies. Three methods are used to evaluate our approach. First, several existing methods are extended to the fine grained situation considered here. Our approach is then compared to these methods using both static schedule length analyses, and simulated executions of the scheduled code. In each instance, our method is found to provide significantly shorter schedules. Second, by varying parameters such as the speed of the instruction set, and the speed/parallelism in the interconnection structure, simulation techniques are used to examine the effects of various architectural considerations on the executions of the schedules. These results show that our approach provides significant speedups in a wide-range of situations. Third, schedules produced by our approach are executed on a two-processor Data General shared memory multiprocessor system. These experiments show that there is a strong correlation between our simulation results (those parameterized to "model" the Data General system), and these actual executions, and thereby serve to validate the simulation studies. Together, our results establish that fine grained parallelism can be exploited in a substantial manner when scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors.

Index Terms— Concurrency, parallelism, multiprocessor, fine grained parallelism, schedule, asynchronous.

I. INTRODUCTION

OVER the past decade or so, changes in technology have provided the possibility for vast increases in computational speed and power through the exploitation of parallelism in program execution. Indeed, within certain computational domains, these technological changes have permitted solutions to *computation intensive* problems such as weather modeling, image processing, Monte Carlo simulations and sparse matrix problems. An important part of this technology has focused on two approaches to parallelizing a sequential instruction stream:

1) exploiting fine grained parallelism, such as single statements, for VLIW machines, [8] and

2) exploiting coarse grained parallelism, such as loops and procedures, on vectorizable machines and on asynchronous multiprocessors.

In the first approach, VLIW machines support the concurrent execution of multiple instruction streams and perform many operations per cycle. VLIW machines however, also employ a *single* control unit, thereby permitting only one branch to be executed per cycle. Furthermore, while the VLIW architectures perform well on programs dealing with scientific applications, their performance can degrade rapidly when faced with factors that decrease run-time predictability. [27] In particular, although general purpose programs typically have an abundance of fine grained parallelism, it is difficult to exploit that parallelism on a VLIW machine because general purpose programs are much less predictable than scientific applications. In the second approach, existing techniques for asynchronous multiprocessors produce schedules at the coarse grained level. Due to their multiple control units, asynchronous multiprocessors have greater flexibility than VLIW machines. Unfortunately, it is frequently the case that a program segment may be unable to support coarse grained parallelism because it does not contain any loops, or because the data dependencies in its loops preclude such concurrentization. Thus, asynchronous multiprocessors, currently present in many installations, are frequently underutilized due to the absence of techniques to exploit fine grained parallelism in an asynchronous manner.

In this paper we offer an alternative approach to the exploitation of parallelism in programs by combining the fine grained approach of the VLIW with the flexibility of the asynchronous machine. In so doing, we thereby provide a mechanism by which parallelism may be exploited in programs where factors are predictable (such as scientific applications), as well as in programs with unpredictable factors (such as general purpose applications).

Thus, we focus on exploiting fine grained parallelism to schedule a sequential instruction stream for execution on an asynchronous multiprocessor system. Recall the *processors* in an asynchronous multiprocessor execute independently and that communication is performed explicitly through asynchronous communication primitives. It follows that scheduling for such systems will necessarily involve packing together fine grained operations, including synchronization commands, for execution on the individual processors. The difficulty in such scheduling lies in balancing the desire to utilize all of the processors, with the desire to minimize the amount of synchronization that is introduced by utilizing different processors for operations having data dependencies.

We conclude this section by noting that although our work is directed toward the parallelization of entire programs, the focus of this paper is on the parallelization of straight line

code such as that found in a basic *block.*[1] Although early studies indicated that basic blocks of programs provide on average only two or three instructions that can be executed in parallel, [24] compiler techniques such as loop unrolling, [7, 26] in-line substitution, [15] code duplication, [12] and trace scheduling [9] are now being employed resulting in a significant increase in the size of basic blocks (currently, up to 1000 instructions). These techniques have, in turn, vastly increased the fine grained parallelism present in a basic block. Throughout the remainder of this paper we focus exclusively on scheduling the instructions of a single basic block for execution on asynchronous tightly coupled multiprocessors.

The remainder of this paper is organized as follows. In the next section, we provide some specifics on the computational/architectural model that is assumed in this work, along with a precise discussion of scheduling in this context. We investigate the complexity of computing a fine grained schedule under our model and conclude that the problem is NP-complete. We then discuss how several existing coarse grained methods can be extended to the fine grained situation considered here. In Section III, we present our approach, the *Preferred Path Selection* algorithm (PPS), for fine grained scheduling on asynchronous multiprocessors. The remainder of the paper is devoted to evaluating our approach. In Section IV, we study the performance of our approach in relation to the modified coarse grained methods described in Section II. Here, comparisons are made using both static schedule length analyses, and simulated executions of the scheduled code. In each instance, our method is found to produce significantly shorter schedules. In addition, these results show explicitly that the approach scales to at least 16 processors when the communication structure provides sufficient parallelism. In Section V, further simulation techniques are used to determine the performance of the PPS algorithm for varying communication speeds and interconnection structure bandwidths, including the modeling of the contention in the communication structure. We conclude that for fast or moderate communication speeds and bandwidths, the PPS algorithm can provide significant speedup for dags containing sufficient parallelism. Finally, in Section VI, schedules produced by our approach are executed on a two-processor Data General AViiON shared memory multiprocessor system. [2] These experiments show that there is a strong correlation between our simulation results (those parameterized to "model" the Data General AViiON system), and these actual executions, and thereby serve to validate the simulation studies.

Together, the simulations and actual executions establish that fine grained parallelism can indeed be exploited in a substantial manner when scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors.

II. MODELS, SCHEDULES AND RELATED WORK

In this section, we provide some specifics on the computational/architectural model that is assumed in this work, along with a precise discussion of scheduling in this context.

[1] A basic block is a sequence of instructions for which the only entrance is through the first statement of the block, and the only exit is through the last statement of the block.

A. The Computational/Architectural Model

In order for us to accurately evaluate the quality of the schedules that we produce, it is necessary that we be a bit more precise about certain aspects of the system that we utilize. In particular, we assume a multiprocessor system M that consists of p asynchronous identical processors, shared global memory modules, and a communication structure that allows processors to communicate with other processors or with the shared memory. We assume that the multiprocessor system includes the standard primitives *send* and *receive,* which are used for the synchronization of processors. Because of the kind of synchronization required here (i.e., based on data dependencies), we assume that the *send* operation does **not** require the invoker to wait until a corresponding receive is executed. [6]

In conjunction with the above system, we employ three parameters that, together, describe the "speed" of the architecture. The first is a function $F_e(I)$ that returns the number of cycles required to execute instruction I. The second is a function $F_c = F_a + F_w$, that indicates the number of cycles needed for communication of values through the interconnection structure. By an *interconnection structure* or *communication structure* we mean hardware support such as memory channels, [1] register channels [11] or an interconnection network [14] that provides support for communication of values. Here, the function F_a is the access time needed to traverse the communication structure and F_w is the number of cycles a processor waits (due to contention) before it can access a required value. The third parameter, BW, is the *bandwidth* of the communication structure or the number of processors that can simultaneously use the structure. Contention occurs when the number of processors vying to communicate during a given cycle, exceeds BW. The simulator used to obtain a variety of results described in Sections IV and V, takes the parameters $F_e, F_c,$ and BW as inputs.

In a portion of what follows, we use an idealized version of the above model to isolate the important issues involved in fine grained scheduling. In this UECC or *uniform execution and communication cost* model, the following conditions hold:

1) $F_e(I) = 1$ for every instruction I,
2) $F_a = 1$,
3) $F_w = 0$,
4) BW $= p$,
5) synchronization primitives Sd_i and Rv_i can execute in the same cycle.

The first condition provides for the execution of any operation in one cycle, and the second and third conditions allow communication through the interconnection structure in one cycle. The fourth condition allows p processors to communicate simultaneously without contention; such throughput might, for example, be provided by a crossbar interconnection topology. The fifth condition allows one cycle for each processor to execute a communication or synchronization primitive. The communication primitive Sd_i indicates that node i has completed execution and the primitive Rv_i requires the executing processor to wait until node i has completed execution.

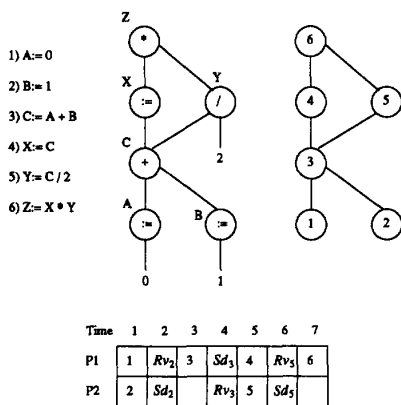Finally, as is standard practice, [3] we use a directed acyclic graph (dag) $G = (V, E)$, to represent the computation

Fig. 1. A program segmentis pictured in the upper left hand corner with acorresponding expression dag to the right of the program segment. To the right of the expression dag is the corresponding **task dag** consisting of nodescontaining numbers to represent the operations in the expressiondagand edges to indicate data dependencies between nodes. A scheduleforthe task dag is pictured at the bottom of the figure where nodes 1,3, 4 and 6 are assigned to list P1 and nodes 2 and 5 are assignedtolist P2.

performed in a basic block. In such a dag, the nodes correspond to computed values, and the acrs indicate data dependencies between values. Thus, the children of a node are the values used to compute the node. In the discussions that follows, we view the dag as shown in Fig. 1, with the root node(s) at the top of the figure. Levels in the dag are numbered from the bottom up, with the bottom (lowest) level numbered level 1.

B. Scheduling Dags

In this section we provide some general information and background on the scheduling of dags in the context of asynchronous machines.

We use the following general approach for scheduling dags on asynchronous machines. There are four phases. First, each node of the dag is assigned to a particular processor. Second, for each processor, a list is constructed of the nodes assigned to that processor. In these lists, nodes appear in *reverse topological order* (a node must appear in a list before any of it's parents). Furthermore, nodes are inserted into the lists in the same order that they appear in the program; thus, if statements 1, 2, and 5 are assigned to list $P1$, then they appear in the list in that order. Third, these lists are modified by incorporating the required communication primitives. Finally, these lists are used to produce a schedule. Of these four phases, phase one, the assignment of nodes to processors is the main focus of this paper. Phase 2 is straight-forward and is not discussed here. The remainder of this section is devoted to a discussion of phases 3 and 4.

In phase 3, note that if two operations A and B are connected with an arc in the dag (A being the parent of B), and they are assigned to different processors, then communication primitives must be inserted. In particular, a *send* is inserted immediately after B, and a *receive* is inserted immediately before A. For example, in Fig. 1, node 3 is the parent of node 2 and nodes 3 and 2 are assigned to different lists. Ultimately, when producing a schedule, a *send* is inserted immediately after node 2 in list $P2$, and a *receive* is inserted

Fig. 2. A possible run-time schedule for the compile-time in Fig. 1. All *receive* primitives required 2 time units to execute in the run-time schedule while *send* primitives required 1 time unit. Also, operation 4 required 2 time units to execute in the run-time schedule, possibly due to contention in the communication structure.

immediately before node 3 in list $P1$. For the example in Fig. 1, both the *send* and *receive* primitives are assigned to the same time slot, but this need not necessarily be the case. Since the communication primitives are asynchronous, even if the *send* operation occurs in a time slot prior to the *receive*, the processor that executes the *send* operation may continue execution. Of course, if the *receive* operation occurs in a time slot prior to the *send* operation, then the processor that issued the receive must wait until the send is issued. [6]

In phase 4, a *compile-time* schedule is produced from the lists of operations and communication primitives. This is, of course, a schedule which is constructed at compile time. We will also refer to a *run-time* schedule, which is what occurs in an actual execution of the compile-time schedule. These two kinds of schedules represent the distinction between what we can model/predict and what actually occurs in a real execution, respectively.

Both kinds of schedules are obtained in the obvious fashion: the operations in list i are executed on processor i, and the jth operation in a list executes only after the previous j-1 operations of the list have completed. Also, a *receive* operation may execute no earlier then its corresponding *send* operation (which is on another *processor*).[2] Clearly this means that some idle time may exist on the processor executing the *receive*. For example, processor $P2$ is idle during time slot 3 in the schedule shown in Fig. 1. In the compile-time schedule constructed under the UECC model, each operation requires one time unit to complete, and *send* and *receive* operations can occur in the same time unit. The *length* of schedule S is equal to the latest time slot during which a node of G executes. For example, in Fig. 1, the length of the schedule is 7. In a run-time schedule, the time to execute any particular operation may vary due to factors such as contention in the communication structure and variances in the actual processor speeds. For example, in Fig. 2, each of the *receive* operations required two time units while the *send* operations required one time unit, possibly due to the particular implementation of the synchronization operations by the multiprocessors.

Clearly the most desirable approach to the code scheduling problem is to produce an assignment that results in an optimal compile-time schedule. However, we establish that producing such a schedule for our UECC model that includes both execution and communication cost is NP-complete, even if there are only 2 processors. Recall the UECC model assumes a multiprocessor M with p identical processors that execute

---

[2] In an actual execution, this is not exactly what occurs. Rather, if the jth operation is a *receive*, then that *receive* executes immediately after the completion of the j-1st operation. Further executions on that processor are suspended until the corresponding *send* operation executes. This is equivalent with respect to time to the "no earlier than the *send*," requirement. We use that requirement to simplify explanations in later sections.

each instruction in one cycle and that a processor can communicate with another processor in one cycle. We assume that no processor has to wait to communicate with another processor and that the p processors can communicate simultaneously. Input to M is a dag $G = (V, E)$ where edges in the dag represent precedence constraints. Given nodes $\{u, v\} \in V$ and edge $(u, v) \in E$, the cost for scheduling u and v on different processors is one unit since communication in M is one cycle. We assume a cost of 0 if u and v are scheduled on the same processor. Formally, as is the usual practice, the problem is stated as a decision problem:

Asynchronous Processor Scheduling (APS):

Instance: A dag and a value L.

Question: Does there exist an assignment of the nodes of a dag to 2 processors such that the length of the synchronized schedule does not exceed L?

Theorem: Asynchronous Processor Scheduling (APS) is NP-complete. (The proof is in the appendix.)

C. Adapting Existing Scheduling Methods

Since the Asynchronous Processor Scheduling problem is NP-complete, we focus on heuristics for finding "good" assignments/schedules, rather than optimal ones. Our heuristic, the Preferred Path Selection algorithm (PPS) is presented in Section III. Sections IV and V are devoted to evaluating the scheduling method that we describe in Section III. One aspect of that evaluation is to compare our method to earlier methods. Unfortunately, only the *Early-Scheduling Method* [20] is aimed at precisely the problem that we consider where communication cost is included as part of the problem. Nonetheless, it has been suggested that traditional task scheduling techniques might be extended in natural ways in order to exploit fine grained parallelism. Two promising techniques are:

    Critical Path, Most Immediate Successors First
    (CP/MISF) [13]

    Internalization Prepass Approach [21]

Since all three of the above methods are a variation of *list scheduling* we begin with a brief discussion of how list scheduling can be used to produce schedules in the situation that we study. We then describe each of the above three methods and how they may be adapted to the fine grained scheduling problem that we consider.

Traditionally, list scheduling has been used for scheduling task systems on synchronous mathines. The idea is as follows:

Given a *priority list* L of the nodes of G, the list schedule S that corresponds to L can be constructed using the following procedure:

1) Iteratively assign the elements of S to a processor, starting at time slot 1 such that during the ith step, L is scanned from left to right, and the first ready node not yet scheduled is chosen to be executed during available time at slot i.

2) If no ready node is found or there is no available time at time slot i, then continue at time slot $i + 1$.

In constructing list L, the first two phases of our method are accomplished, assignment of nodes to a processor and construction of a list of nodes to be executed by each processor. The versions of list scheduling algorithms can be distinguished

by the method in which L is obtained. In *critical path* scheduling, nodes at the lowest levels of the dag (farthest from a root node) are inserted into L first. Since there can be more than one node at a given level in the dag, a version of critical path scheduling called CP/MISF [13] (critical path/most immediate successors first) attempts to establish a hierarchy among nodes at the same level by assigning a higher priority to those with more immediate successors.

To adapt list scheduling in general, and CP/MISF in particular, to an asynchronous model, communication primitives must be inserted in an appropriate fashion to accomplish phase three of our method. We view the "schedule" produced by a list scheduling algorithm (such as CP/MISF) as merely an assignment of operations to processors in a particular order. Using these assignments, each node in S is examined to determine if its successor(s) in the dag is scheduled on the same processor. If a node in S has a successor assigned to a different processor, then communication primitives are inserted in the appropriate lists.

The *Early-Scheduling Method* [20] represents an attempt to include communication cost in the determination of the schedule. The algorithm maintains a list E containing unscheduled nodes that are ready for execution (eligible nodes), and sequences s_1 through s_p. Sequence s_i contains the nodes that are already assigned to processor P_i. The algorithm proceeds iteratively as follows:

1) For each node $x \in E$ and each processor $P_i \in P = \{P_1, \cdots, P_p\}$, calculate the finish time of x on P_i including insertion of communication primitives if needed.

2) Let f be the earliest finish time of a node x from 1). Create set A containing all possible assignments of eligible nodes to processors having finish time f.

3) Choose a node randomly from set A and assign it to sequences s_i.

After all of the nodes in the dag have been assigned to a sequence s_i, sequence s_i is mapped to processor P_i. As in the other list scheduling approaches, communication primitives are inserted into s_i to produce an actual schedule.

The third method that we consider is the *Internalization Prepass Approach*, [21], [23] which processes program graphs which represent computation as dataflow graphs. This approach was not designed for scheduling dags (graphs whose nodes represent operations) but rather for graphs whose nodes represent structures contained in a program written in a functional language. We modify the Internalization Prepass Approach so that the nodes of the graph are operations and include it as a comparison with the PPS approach. The Internalization Prepass Approach attempts to minimize communication cost by internalizing (executing on the same processor) nodes along the critical path. [21] The algorithm maintains a list of blocks that initially contains 1 node per block and a table DeltaCPL $[i, j]$ that represents the decrease in the critical path length obtained by merging blocks i and j. Blocks that will result in a decrease in the critical path length are merged until further mergers cannot reduce the critical path length. In computing the critical path length, all nodes in the same block are sequentialized since they will be assigned to the same processor. After the internalization prepass, the approach uses

Input: A dag G=(V,E) and the number p, of processors.
Output: An assignment of each node of G to a processor.

$i := 1$;
While there are unassigned nodes Do
   $k :=$ largest level with an unassigned node;
   If $\exists$ an unassigned node at level k with at least one parent assigned to P_i
      Then *BestNode* := such an unassigned node
      Else if $\exists$ an unassigned node at level k with no parent
         Then *BestNode* := such an unassigned node
         Else *BestNode* := any unassigned node at level k;

Done := False;
While NOT *Done* Do
   Assign *BestNode* to processor P_i;
   If all children of *BestNode* are assigned
      Then *Done*:= True
      Else If $\exists$ an unassigned child of *BestNode* with at least one
            other parent assigned to P_i
         Then *BestNode* := such an unassigned child of *BestNode*
         Else *BestNode* := any unassigned child of *BestNode*;

$i := i \bmod p + 1$;

Fig. 3. The Preferred Path Selection algorithm (PPS).

a modified priority list scheduling algorithm to assign nodes to processors with the modification that when a node is assigned to a processor, all other nodes in the same block are assigned to the same processor.

III. THE PREFERRED PATH SELECTION ALGORITHM—AN INTEGRATED APPROACH

In this section, we describe our algorithm for the scheduling of program dags on an asynchronous multiprocessor. Actually, based on the discussion in the previous section, we limit the discussion here to "phase 1"—that is, to the assignment of each node to some processor. Throughout this paper, we use the term PPS to refer both to the entire algorithm and more particularly, to this first step. Typically, the meaning will be clear from the context.

As noted earlier, the key idea in assigning nodes to processors, is to exploit the *fine grained parallelism* present in the instruction stream by a careful balancing of execution and communication costs at the level of individual instructions, and in consideration of their data dependencies. Thus the algorithm that we present incorporates the dag *structure,* as well as communication costs in its computation of a schedule. In particular, the algorithm attempts to *minimize communication costs* by locating a **path** L_i in the dag and assigning *all of the nodes on the path* to the same processor P. Such a path, by definition, represents a series of data dependencies, and by scheduling the entire path for execution on a single processor, the need for synchronization among the nodes on this path is eliminated. Further, we attempt to maximize these savings in communication costs, by insuring that in the construction of L_i for execution on processor P: 1) that nodes with a parent unassigned or assigned to P, are preferred over those with a parent assigned to a processor other than P; and 2) that L_i is maximal (i.e., it cannot be extended). The complete algorithm is given in Fig. 3; an input of a dag $G = (V, E)$ and a multiprocessor with **P** processors is assumed.

To illustrate the manner in which the PPS algorithm assigns nodes to processors, we use it to schedule the dag shown in Fig. 4 on two processors. Here, the initial value of k is 3, since node 1 is at level 3 and is unassigned. BestNode is also node 1 since it has no parent. In the first iteration of the inner **While** loop, node 1 is assigned to P_1. In the next iteration of this inner **While** loop, a child of node 1, say node 2, is chosen as BestNode and is assigned to P_1. In the next iteration of

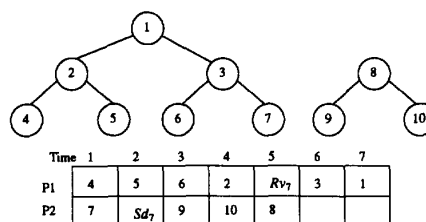| Time	1	2	3	4	5	6	7
P1	4	5	6	2	Rv_7	3	1
P2	7	Sd_7	9	10	8		

Fig. 4. A sample dag and the corresponding schedule produced using the PPS algorithm.

the inner loop, node 4 is assigned to P_1 and this inner loop terminates since all children of node 4 are assigned. Thus, path (1, 2, 4) in the dag is assigned to P_1. The PPS algorithm continues execution in the outer **While** loop by updating i to 2 indicating that we are now assigning nodes to P_2. The variable k is also updated to 2, since nodes 3 and 8 are unassigned. As execution continues at the top of the outer **While** loop, node 8 becomes BestNode since it has no parent and it is assigned to P_2. BestNode is then updated to 9, assigned to P_2 in the inner **While** loop and the inner loop terminates with path (8, 9) in the dag assigned to P_2. In the outer **While** loop i is updated to 1 and execution continues at the top of the outer loop where k remains 2 and BestNode becomes 3 since its parent, node 1, is also assigned to P_1. BestNode is assigned to P_1 and is updated to a child of node 3, say node 6, in the inner loop and the inner loop terminates with path (3, 6) in the dag assigned to P_1. The PPS algorithm continues until all nodes in the dag are assigned. The schedule resulting from this assignment is shown in Fig. 4.

IV. PERFORMANCE EVALUATION USING THE COMPUTATIONAL/ARCHITECTURAL MODEL

In this section we compare the performance of our approach to the modified coarse grained methods described in Section II. In the first portion of this section, we compare the lengths of compile-time schedules produced by each of the methods. These comparisons use the UECC model for numbers of processors ranging from 2 through 16. These results show that the PPS algorithm performs significantly better than any of the other methods. Further, these results show that in absolute terms, the performance of the PPS approach is quite good. That is, for every number of processors in this range the PPS algorithm is able to produce schedules which utilize a significant amount of the parallelism provided by those processors. In particular, these results show explicitly that the approach scales to at least 16 processors, provided that the communication structure provides sufficient parallelism. In the second portion of this section we compare the performance of the various methods using simulated executions of the compile-time schedule. Recall from Section II that we refer to these as the run-time schedules. Unlike the compile-time schedule results, these simulations are not restricted to the UECC model. Once again, the PPS algorithm is found to provide significantly shorter schedules than the other methods. In conjunction with these evaluations, we provide extensive evaluations of the PPS approach under a range of architectural assumptions including the modeling of the contention in the communication structure. All of these results verify the

TABLE I
PERFORMANCE EVALUATION (p = 2 PROCESSORS)

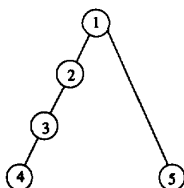| Program	Nodes in dag	Heuristics				
		CP/MISF	Early	Prepass	Random	PPS
Sample	10	11	9	12	13	7
Fibonacci	20	30	20	20	33	20
Pyramid	36	36	35	42	45	28
Mat Mult	120	126	97	97	139	77
Dual Dag	107	104	104	104	131	54
Whetstone	137	148	89	108	158	84
FFT	190	159	159	127	187	97
Livermore	203	170	102	102	194	102

Fig. 5. A dag with a chain.

compile-time schedule results—namely that the PPS algorithm is able to scale to 16 processors.

A. Compile-Time Schedule Comparisons

In this section, we compare the lengths of compile-time schedules produced by each of the methods: *CP/MISF, Early-Scheduling Method, Internalization Prepass* and *PPS* algorithms. In addition, a *Random* assignment algorithm is included to serve as a "control" for the comparison of the heuristics. This algorithm, assigns the nodes of a dag to processors in a random fashion. The details of the implementation are straight-forward and are left to the reader.

Finally, we note that in this section, all of the comparisons were done using the UECC model. Results were obtained for 2, 3, 4, 8, and 16 processors. In each instance, the results show that the PPS algorithm performs significantly better than any of the other methods.

The results of the evaluations on two processors are summarized in Table 1 (the results for 3, 4, 8, and 16 processors are similar and may be be found in [16]). For example, *Sample* is a program whose corresponding dag contains 10 nodes as shown in Fig. 4. Applying CP/MISF to Sample resulted in a compile-time schedule of length 11, while Early, Prepass and Random produce schedule lengths of 9, 12, and 13 respectively. Applying the PPS algorithm to Sample resulted in a schedule of length 7 as shown in Fig. 4.

To fully evaluate the heuristics, their performance was examined using a variety of dags as input, including dags having long or wide topologies, duplication of similar patterns, those having theoretical interest as well as those of practical application. The number of nodes in the dags ranges from 10 to 203. In addition to program *Sample* discussed above, Table I contains seven other test programs. The programs Fibonacci and Mat Mult were obtained by using loop unrolling to compute the first ten Fibonacci numbers and to multiply two 3 × 3 matrices. The program Pyramid is an example of a grid. [19] FFT is a program whose dag is a complete binary tree and Dual Dag is a program whose dag contains duplicate components. Finally, the whetstone program was obtained by unrolling loops in four of the Whetstone modules and Livermore is a program containing the first 20 iterations of the first kernel of the Livermore loops. [18]

From Table I, it is clear that in almost every instance, our PPS algorithm produces significantly shorter schedules than any of the other methods. We believe that this superior performance of the PPS algorithm can be attributed primarily to its focus on minimizing communication costs, while the earlier algorithms (all based on list scheduling) attempt to minimize processor idle time exclusively. To accomplish this, the earlier algorithms focus primarily on executing nodes at the lowest level first. Unfortunately, this strategy can schedule on different processors, nodes that are all connected to a single successor. Such a situation obviously requires a great deal of communication and therefore a longer schedule. A further advantage of the PPS algorithm is that it incorporates the structure of the dag in computing the *preferred path* and by assigning the entire path to a processor, the PPS approach maintains a *globalview* of the dag in its computation of a schedule. The earlier list scheduling algorithms utilize a much more *localview*, in examining primarily, nodes on a single level to decide which to schedule next. For example, the earlier algorithms may quite easily assign the nodes of Fig. 4 in the following manner: nodes 4, 6, 9, 2, and 8 to processor 1 and 5, 7, 10, 3, and 1 to processor 2. By assigning nodes 4 and 5, 6, and 7, 9, and 10, and 2 and 3, to different processors, communication between processors 1 and 2 is required, resulting in a schedule of length 11. For the PPS algorithm, nodes along the longest path are assigned to the same processor (for example nodes 1, 2 and 4) and communication is not required for any of these nodes.

The Internalization Prepass Approach produces excellent results when applied to graphs that result from functional programs, [21] since they typically produce long chains of computations. However, the results in Table I indicate that the Internalization Prepass Approach does not perform as well as the PPS algorithm when applied to expression dags. This is primarily due to the fact that the Prepass algorithm is only able to internalize or merge a low percentage of the nodes that occur in expression dags, in particular, those that lie along a chain such as nodes 1, 2, 3, and 4 in Fig. 5. To demonstrate the merging of nodes, recall that the algorithm utilizes a table, DeltaCPL $[i,j]$, that represents the decrease in critical path length that will result when nodes i and j are merged. [23] DeltaCPL can be initialized with the loop, DeltaCPL $[i,j] :=$ origCPL–newCPL, for all $i \neq j$; the algorithm then merges pairs of nodes with a positive DeltaCPL entry until all entries are negative. Since one unit is required for node execution and one unit for communication, the critical path in Fig. 5 is 1, 2, 3, 4 with length 7. If nodes 1, 2, and 3 are merged, the critical path length reduces to 5 since the path (1, 2, 3, 4) has length 5 and the path (1, 2, 3, 5) also has length 5, where nodes 2 and 3 must be executed on the same processor as 1. No further merging is possible. For the dags in Table I, a low percentage of nodes were merged and thus the Internalization Prepass Approach gave results nearly identical to the other local-view algorithms. For example, the Prepass merged none of the nodes in Whetstone.

We conclude this section by noting that the PPS algorithm is able to provide *speedup*, not only for two processors (Table

TABLE II
SCALABILITY OF THE PPS ALGORITHM

Program	Nodes in dag	Avg Indegree	No. of Components	Speedup p=8	Speedup p=16
Fibonacci	20	1.30	1	1.00	1.00
Pyramid	36	1.25	1	1.89	1.89
Mat Mult	120	1.23	9	3.53	5.45
Dual Dag	107	1.43	2	1.98	1.98
Whetstone	137	1.42	13	2.74	3.26
FFT	127	0.99	1	6.13	8.64
Livermore	203	1.30	20	6.34	9.67

I), but also for numbers of processors up to 16 (Table II). Here, speedup is the ratio of the number of nodes in the dag to the length of the compile-time schedule. When the speedup is equal to the number of assigned processors, then we say that the speedup is *linear*. In the case of a dag containing 10 nodes, a schedule must have length 5 to provide linear speedup for 2 processors. In our evaluations, the PPS algorithm provided virtually linear speedup for FFT and Livermore on 8 processors, and was able to provide significant speedup for programs containing sufficient parallelism. In particular, the Fibonacci and Pyramid programs did not experience significant speedup because their corresponding dags contain a large number of data dependencies. The speedup provided by the PPS algorithm for the test cases is summarized in Table II for 8 and 16 processors.

The results in Table II also indicate the factors that contribute to the degree of parallelism contained in a dag. These factors are:

1) the number of nodes in the dag;
2) the *average indegree* for each node, and
3) the number of connected components in the dag.

The number of nodes in the dag is important since, in the extreme, it is impossible for the dag to support linear speedup when the number of assigned processors is greater than the number of nodes in the dag.

The *average indegree* of a dag is the ratio of the total number of edges entering nodes to the total number of nodes,

$$\text{Avg Indegree} = \frac{\text{Number of edges entering a node}}{\text{Total number of nodes}}.$$

For example, in Fig. 4 nodes 2, 3, 8, and 1 have indegree 2 while nodes 4, 5, 6, 7, 9, and 10 have indegree 0. The average indegree of a dag indicates the number of uses of computed values and therefore indicates possible synchronization points that can increase the length of the computed schedule. Table II correlates the average indegree with the ability of the PPS algorithm to provide good speedup. As the average indegree increases from 1 to 2, the speedup for 8 and 16 processors decreases so that for Whetstone and Dual Dags the PPS algorithm did not scale for 8 processors and performance did not improve when the number of processors was doubled to 16. When the average indegree approaches 1, the PPS algorithm provided significant speedup for 16 processors except for the Fibonacci and Pyramid dags which do not contain enough nodes to support 16 processors.

The number of connected components in the dag can also affect speedup. For example, a single connected component assigned to n processors must include $n - 1$ communication pairs in the corresponding schedule. Moreover, n connected components assigned to n processors poses the possibility of

linear speedup if each component contains the same number of nodes and is assigned to a different processor. The PPS algorithm was able to provide good speedup on 2 processors for programs that contain 2 or more connected components, as can be seen from Table I for Mat Mult, Dual Dag, Whetstone and Livermore. When the dag contains a single connected component and has an average indegree larger than 1.25, the PPS algorithm was not able to provide significant speedup such as with the Fibonacci and pyramid dags.

B. Simulation Results For Run-time Schedules

In the previous section we evaluated the various methods by comparing the lengths of the compile-time schedules they produced. While we believe that these comparisons provide a very good indication of the *relative* quality of the corresponding run-time schedules, it is true that the compile-time schedules provide only a lower bound on the lengths of the run-time schedules for the given assignment of nodes to processors. Further, there is no reason to believe that among the heuristics that we consider, one would be any more or less affected than another by runtime factors such as contention in the communication structure or the speed of the structure.

Nonetheless, it seems appropriate to test these observations by comparing run-time schedules. Thus, in this section we simulate the executions of schedules produced by the various methods, on architectures differing in communication speed and bandwidth. As noted in section 2, this is achieved by supplying the three parameters, $F_e(I)$, F_c, and BW, to a simulator that we constructed using the process oriented simulation language Simcal. [17]

In the simulations of this section, the values established by Sarkar [22] are used to describe the execution times for simple operations $(F_e(I))$ and the time needed to communicate a value (F_c). In particular, a table of cost values is utilized to define the value of the function $F_e(I_j)$ for each instruction I_j. To describe the access time via the communication structure, we let $F_a = 2^* k^* s$. We consider three situations, depending on values for k of 0.0, 0.125 and 1 which correspond to *fast, medium* and *slow* access times respectively. Examples of such communication structures are *channels* for providing a fast communication structure, a *crossbar* or *omega* network providing a medium speed structure and a *unibus* providing a slow structure. The parameter s describes the size of the data value being transferred and for fine grained scheduling is assumed to be 4 bytes.

As in previous work, [5], [25] we use various bandwidths (BW) to model the contention in the communication structure. A value of 1 for BW describes a worst case communication structure that allows only one request to be accepted per cycle; a value of $\sqrt{p}$ describes a multistage network such as that proposed by Lang [14]; and finally, a value of p describes the best case bandwidth where p requests can be accepted per cycle.

The results of these simulation studies again show that in comparison with the other methods, the PPS approach produces significantly better schedules. [16] We omit these results since they are similar in nature to those of the previous section and present the simulation results for the PPS algorithm in Table III and IV. These tables illustrate the speedup obtained by executing the run-time schedules for *Mat Mult* and *FFT* on

TABLE III
SPEEDUP FOR PPS ALGORITHM—MAT MULT

F_a	BW	processors				
		p=2	p=3	p=4	p=8	p=16
fast	1	1.69	2.50	2.71	4.09	7.42
med	1	1.58	1.92	2.04	2.40	2.18
	$\sqrt{p}$	1.58	1.92	2.60	4.10	6.46
	p	1.68	2.42	2.78	4.10	7.00
slow	1	1.31	1.37	1.37	1.39	1.33
	$\sqrt{p}$	1.32	1.43	2.57	3.94	5.03
	p	1.80	2.63	3.26	4.70	7.93

TABLE IV
SPEEDUP FOR PPS ALGORITHM—FFT

F_a	BW	processors				
		p=2	p=3	p=4	p=8	p=16
fast	1	1.95	2.46	3.61	5.92	8.06
med	1	1.58	1.74	1.83	1.83	1.78
	$\sqrt{p}$	1.58	1.74	3.16	4.48	5.44
	p	1.98	2.49	3.67	6.17	8.72
slow	1	1.22	1.19	1.21	1.21	1.18
	$\sqrt{p}$	1.22	1.61	2.10	2.94	4.41
	p	1.97	2.66	3.78	6.71	10.33

TABLE V
COMPARISON OF SIMULATION WITH ACTUAL EXECUTION

Test Program	Experimentation					
	Simulations using Parameterized Cost Model			Actual execution on Data General Multiprocessor		
	Time (p=1)	Time (p=2)	SpUp	Time (p=1)	Time (p=2)	SpUp
Fibonacci	54	60	0.90	0.23	0.25	0.88
Pyramid	102	113	0.90	0.43	0.67	0.65
Mat Mult	336	277	1.21	1.14	1.13	1.01
Dual Dag	311	160	1.94	2.49	1.27	1.96
Whetstone	411	300	1.37	0.90	0.67	1.34
FFT	506	325	1.55	2.05	1.37	1.49
Livermore	643	381	1.69	1.71	0.95	1.80

2, 3, 4, 8 and 16 processors using a fast, medium and slow communication structure with a bandwidth of 1, $\sqrt{p}$ and p.

In analyzing the results shown in Tables III and IV, recall from Table II, that the dag for Mat Mult contains nine connected components, and that the dag for FFT contains a small average indegree and therefore few data dependencies. The results in Tables III and IV demonstrate that a good speedup can be achieved for these two programs using a fast communication structure. Using a medium speed structure, good speedup is also achieved if the bandwidth is $\sqrt{p}$ or p. However, if the bandwidth is the *worst case* value of 1, representative of a unibus structure, the performance can degrade with increasing number of processors due to contention in the communication structure. For the Mat Mult program executed on a multiprocessor with a medium speed unibus structure, the results in Table III show that speedup increases form 1.58 on 2 processors to 1.92 on 3 processors, to 2.04 on 4 processors and to 2.40 on 8 processors. Speed up on 16 processors decreases from that achieved on 8 processors, from 2.40 to 2.18. This phenomenon whereby speedup "levels off" or decreases as the number of processors is increased from 8 to 16 can be observed in Tables III and IV for all cases where the bandwidth is 1. Thus, for a unibus communication structure, increasing the number of processors can produce more contention and a longer run-time schedule.

V. PERFORMANCE OF THE PPS ALGORITHM ON A DATA GENERAL MULTIPROCESSOR

As noted earlier, the PPS algorithm was implemented on a Data General AViiON shared memory multiprocessor system [2] equipped with a unibus communication structure and two identical processors. The *send* and *receive* primitives were implemented using spin-lock operations on *unix shared variables* [4]. In order to compare the results of these actual executions, with corresponding simulation results, we first conducted a series of experiments to determine the average cost of the *send* and *receive* primitives and the cost of using the unibus communication structure. These experiments revealed that a *send* primitive requires approximately the same time to execute as a floating point multiplication, and that a *receive* primitive requires approximately twice as long as a floating point multiplication (provided, of course, that the *receive* does not have to wait). These values were utilized in setting the parameter F_c for the simulation studies described below.

The result summarized in Table V indicate a strong correlation between the simulation results and the actual executions on the Data General multiprocessor. In Table V, the first column lists the programs used in the experiments, the next three columns report the results of the simulations and the last three columns report the results of the actual executions. For the simulations, the second and third columns express the number of cycles required to execute the test program on 1 and 2 processors respectively. For the actual executions, the fifth and sixth columns express the number of seconds required to execute the test program 10,000 times; these experiments were conducted 1000 times and the results reported are the averages. As a particular instance, note that the simulation indicates that 54 cycles are required to execute the sequential code, and that 60 cycles are required to execute the schedule for 2 processors with a resulting speedup of 0.90 over the sequential *execution.* A speedup of less than one indicates that the parallel execution took longer than the sequential execution assuming machines with the same architectural configuration. For the actual execution of the Fibonacci program on the Data General multiprocessor, an average of 0.23 seconds were required for 10 000 iterations using 1 processor and 0.25 seconds were required for 10 000 iterations using 2 processors producing a speedup of 0.88 over the sequential execution.

The similarities in speedup between the simulation and actual execution results are established by comparing columns 4 and 7. with the exception of the Pyramid and Livermore programs, the difference between these speedups is never more than 0.25. This is a remarkably small difference, and certainly validates the use of the simulation approach in most instances.

In addition to supporting the correlation between the simulation results and the actual executions on a Data General Multiprocessor, Table V also supports the conclusion that the PPS algorithm is able to provide very good speedup for programs containing sufficient parallelism. Sufficient parallelism implies that the resulting dag does not contain a large number of data dependencies (as expressed by the average indegree for the edges), and has enough nodes to support all or most of the processors.

TABLE VI
SIMULATIONS FOR 2, 3, 4, 8, AND 16 PROCESSORS USING
PARAMETERS THAT DESCRIBE THE DATA GENERAL AViiON

Program	Processors				
	p=2	p=3	p=4	p=8	p=16
Fibonacci	0.90	1.35	1.35	1.35	1.35
Pyramid	0.90	1.32	1.30	1.30	1.30
Mat Mult	1.21	1.76	1.80	1.77	1.62
Dual Dag	1.94	1.65	1.45	1.45	1.45
Whetstone	1.37	1.40	1.67	1.47	1.47
FFT	1.55	1.29	1.31	1.30	1.28
Livermore	1.69	2.56	2.52	2.55	2.55

TABLE VII
SIMULATIONS FOR 2, 3, 4, 8, AND 16 PROCESSORS USING
PARAMETERS THAT DESCRIBE A DATA GENERAL MACHINE
EQUIPPED WITH AN OMEGA TYPE COMMUNICATIONS STRUCTURE

Program	Processors				
	p=2	p=3	p=4	p=8	p=16
Fibonacci	0.98	1.00	1.00	1.00	1.00
Pyramid	1.28	1.55	1.85	2.32	2.32
Mat Mult	1.58	2.35	2.60	4.10	6.46
Dual Dag	1.94	1.97	2.14	2.53	2.59
Whetstone	1.81	2.08	2.52	2.96	3.51
FFT	1.58	2.34	2.99	4.48	5.44
Livermore	1.69	2.60	3.38	5.79	10.54

Since the Data General AViiON multiprocessor at our installation is equipped with only two processors, we are not able to evaluate the performance of the PPS algorithm for actual executions of schedules using more than two processors. However, simulations using parameters appropriate to the Data General machine, produce the results shown in Table VI for executions on 2, 3, 4, 8, and 16 processors. These results suggest that if the AViiON were to maintain its current configuration except for the addition of more processors, no significant speedup would be achieved by using these additional processors. The main bottleneck in the system is the unibus communication structure. In fact, an examination of Table VI reveals the same "leveling off" effect that was observed in Tables III and IV for the case where a unibus communication structure is employed. The lack of parallelism in the unibus communication structure produces a great deal of contention when accessing memory for load/stores and for synchronization with unix shared variables.

On the other hand, if the Data General were equipped with both a larger number of processors, and an omega type communication structure that permitted $\sqrt{p}$ processors to communicate simultaneously, then the speedups shown in Table VII could be achieved. These results show that the addition of the omega network produces significant speedup using 4 processors for the Mat Mult, Dual Dag, Whetstone, FFT, and Livermore programs. Of course, increasing the speed of the communication structure and providing architectural support for the synchronization primitives [1], [11] would produce even more dramatic results for increased numbers of processors.

VI. CONCLUSION

We have provided a new approach for scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors. The key idea in our approach is to exploit the *fine grained parallelism* present in the instruction stream. In this context, schedules are constructed by a

careful balancing of execution and communication costs at the level of individual instructions, and their data dependencies. Our approach was compared using both compile-time and run-time schedules to methods adapted from existing (primarily, coarse grained) methods. These comparisons show that our method provides superior schedules to each of the alternative methods. In addition, our results support the conclusion that if the multiprocessor system incorporates a communication structure that allows $\sqrt{p}$ or more processors to communicate simultaneously, then a large degree of speedup is achieved on 2 to 16 processors by using the PPS algorithm.

In addition to the compile-time and simulation studies, the PPS algorithm was implemented on the Data General AViiON shared memory multiprocessor system. Here, actual executions of PPS algorithm, generated schedules produce speedups that closely correspond to those produced in our simulation studies (those parameterized to "model" the Data General system). These results are encouraging for the development of compile time techniques for scheduling fine-grained operations.

APPENDIX
A PROOF THAT APS IS NP-COMPLETE

In this appendix we provide the proof of Theorem 1. Namely, we show that asynchronous processor scheduling (APS) is NP-complete, even when there are but two processors. We begin by recalling the definition:

Asynchronous Processor Scheduling (APS):
  Instance: A dag and a value L.
  Question: Does there exist an assignment of the nodes of the dag to 2 processors such that the length of the synchronized schedule does not exceed L?

Throughout this appendix, we use the term **schedule** to refer both to an assignment and to its corresponding schedule. The meaning of the term will be clear from the context.

To show that APS is NP-complete, we note that it is easy to show that APS $\in$ NP, and proceed directly to establishing that the following NP-complete problem is polynomially reducible to APS.

3-partition problem [10] **(3-PART):**
Input: Multiset A containing $3n$ integers and an integer bound $B >= 2$, where $B/4 < a_i < B/2$ for all $a_i \in A$ and $\sum_{i=1}^{3n} a_i = Bn$.
Question: Is there a partition of A into n triples of three elements each such that the sum of the integers in each triple equals B?[3]

Given an instance of 3-PART, we construct an instance of APS that consists of the following:

- For each a_i in the instance of 3-PART, there is a chain C_i of $2a_i$ nodes, (i.e., each node except for the end nodes has a unique parent and a unique child). The first a_i nodes in C_i are *red* nodes and the second a_i nodes are *black* nodes. All of the nodes in C_i are *partition* nodes.
- There is a chain of $2(B + 3)n$ nodes. The first $B + 3$ nodes are black, the second $B + 3$ nodes are red, the third

---
[3] Because the 3-partition problem is strongly NP-complete, a reduction that is polynomial in the value of the numbers in the 3-partition problem instance is sufficient for a proof of NP-completeness.
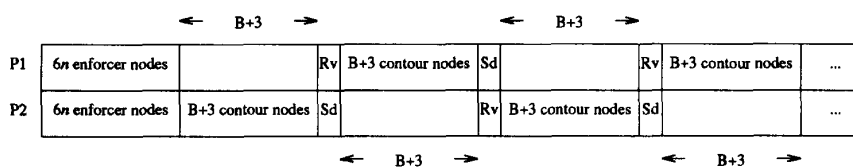
Fig. 6. Partial Schedule Construction. All of the nodes on $P1$ are red nodes and all of the nodes on $P2$ are black nodes. Rv indicates a *receive* and Sd indicates a *send*.

$B + 3$ nodes are black, and so on, alternating colors in blocks of $B + 3$ nodes. All of the nodes in this chain are *contour* nodes.

- There is a set of $6n$ additional red nodes and a set of $6n$ additional black nodes. These are *enforcer* nodes, and there is an edge from each red enforcer node to each red partition or contour node. There is an edge from each black enforcer node to each black partition or contour node. Intuitively, the enforcer nodes will force all of the red nodes to execute on one processor and all of the black nodes to execute on the other processor.
- $L = 6n + 2(B + 3)n + 2n - 1 = 2Bn + 14n - 1$.

Now suppose that there is a solution to the instance of 3-PART. A solution to APS is as follows: Completely fill the first $6n$ time units of the schedule by placing all of the red enforcer nodes on one processor, say p_1, and all of the black enforcer nodes on the other processor, p_2. Next, schedule all of the red contour nodes on p_1, and all of the black contour nodes on p_2. Note that these contour nodes appear in groups of $B+3$ nodes, with the groups alternating between p_1 and p_2. Thus, between successive groups of contour nodes, we insert a *send/receive* pair to synchronize between the last red(black) node in a group and the first black(red) node in the next group. The partial schedule constructed to this point is shown in Figure 6. Clearly, the partition nodes must be scheduled in the portions where no tasks are currently scheduled. Note that these unscheduled portions of the schedule occur in blocks of size $B + 3$ and alternate between the two processors. Thus, we schedule the nodes in the C_i chains as follows: Suppose that in the solution to the instance of 3-PART, that a_i, a_j and a_k form the hth element of that partition. Thus, $a_i + a_j + a_k = B$. Then, in the hth unscheduled block on p_1, we schedule the red nodes in C_i, C_j and C_k, followed by three sends (one from the last red node in C_i to the first black node in C_i, etc.). And, in the hth unscheduled block on P_2, we schedule the three corresponding receives, followed by the black nodes in C_i, C_j and C_k. Since each unscheduled block is of length $B+3$, and we schedule exactly B nodes and 3 synchronizations per block, we have a valid schedule.

Conversely, suppose that there is a solution to the constructed instance of APS. We need to show that there also exists a solution to the instance of 3-PART.

We begin by claiming that the APS schedule must be such that all of the red nodes are scheduled on one processor and that all of the black nodes are scheduled on the other processor. To see that this is the case, assume by way of contradiction that red nodes are scheduled on both processors. We consider two cases.

1) Assume that each processor executes at least one red contour or partition node. Then, each processor will contain at least $6n$ *sends* and $6n$ *receives* to account for synchronization between the red enforcer nodes and the red contour and partition nodes. Since there are $4Bn + 18n$ nodes altogether, this implies that the schedule length is at least $2Bn + 15n > L$, hence, a contradiction. Thus, all of the red contour and partition nodes are scheduled on one processor, and, similarly, all of the black contour and partition nodes are scheduled on the other processor.

2) Assume that each processor executes at least one red enforcer node. Since from case 1, we know that all of the red contour and partition nodes are scheduled on one processor, this means that there are at least $2(B + 3)n$ *sends* and $2(B + 3)n$ *receives* between red enforcer nodes and red contour and partition nodes. Since there are $4Bn + 18n$ nodes altogether, this implies that the schedule length is at least $4Bn + 15n > L$, hence, a contradiction. Thus, all of the red nodes (enforcer, contour and partition) are scheduled on one processor, and all of the black nodes are scheduled on the other processor.

Since all of the red nodes are scheduled on one processor, say p_1, and all of black nodes on the other processor (p_2), it follows from the precedence constraints that, when considering only enforcer and contour nodes, the schedule must have the form shown in Fig. 6. That is, the enforcer nodes are scheduled in the first $6n$ time units. In time units $6n+1$ to L, the contour nodes alternate on the two processors in blocks of $B+3$ nodes, with a single *send/receive* pair being scheduled between each block of $B + 3$ nodes. This means that the partition nodes (and associated synchronizations) must be scheduled in the unused portions of the schedule shown in Fig. 6. Note that these unused portions can accommodate exactly $2(B + 3)n$ nodes and/or synchronization operations. Since there are $2Bn$ partition nodes and since, for each C_i, one send/receive pair is required between the last red node in C_i and the first black node in C_i (for a total of $3n$ sends and $3n$ receives), it follows that there is no idle time in the schedule, nor can any other synchronization be introduced.

To complete the proof, we consider the first unused block H_2 on p_2 and consider which partition nodes could be scheduled in that block. Note that since in the instance of 3-PART, each $a_i < B/2$, there must exist partition nodes scheduled in H_2 from three chains, say C_i, C_j and C_k. Could there be nodes from a fourth chain, say C_h? By way of contradiction, assume so. Then, since these partition nodes are black, it follows that all of the red nodes of C_i, C_j, C_k and C_h must be scheduled in H_1, the first unused block on p_1. Further, 4 sends must also

be scheduled in H_1. But, since each $a_i > B/4$, it follows that the total number of nodes and sends scheduled in H_1 exceeds $B + 4$. Since H_1 is of length $B + 3$, this is a contradiction. Thus we have the following.

1) H_1, contains all of the red nodes of C_i, C_j and C_k, along with three sends. It follows that $a_i + a_j + a_k + 3 \leq B + 3$, hence, $a_i + a_j + a_k \leq B$.

2) H_2 contains black nodes of C_i, C_j, and C_k, along with three receives, and nothing else. Since the schedule is known to contain no idle time, it follows that $a_i + a_j + a_k + 3 \geq B + 3$, hence $a_i + a_j + a_k \geq B$.

From these, we have that $a_i + a_j + a_k = B$. Thus, $\{a_i, a_j, a_k\}$ is one element of the desired 3-partitions. A complete solution to 3-PART follows in an inductive fashion.

ACKNOWLEDGMENT

REFERENCES

[1] "Parallel MIMD Computation: HEP Supercomputer & Its Applications," *Scientific Computation Series.* Cambridge, MA: MIT Press, 1985.
[2] *Installing and Managing the DG/UX System,* Data General Corporation, 1990.
[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques and Tools .* Reading, MA: Addison-Wesley, 1986.
[4] M. J. Bach, *The Design of the Unix Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986.
[5] Z. Cvetanovic, "The effects of problem partitioning, allocation, and granularity on the performance of multi-processor system," *IEEE Trans. Comput.,* vol. C-36, no. 4, Apr. 1987.
[6] A. Dinning, "A survey of synchronization methods for parallel computers," *Comput.,* pp. 66–76, July 1989.
[7] J. J. Dongarra and A. R. Jinds, "Unrolling loops in Fortran," *Software Practice and Experience,* pp. 219–226, Mar. 1979.
[8] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1986.
[9] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.,* vol. C-30, no. 7, July 1981.
[10] M. R. Garey and D. S. Johnson, *Computers and Intractability, A guide to the Theory of NP-Completness.* San Francisco: Freeman, 1979.
[11] R. Gupta, "Employing register channels for the exploitation of instruction level parallelism," presented at the *Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* Seattle Washington, Mar. 1990.
[12] W.-C. Hsu, C. N. Fischer, and J. R. Goodman, "On the minimization of loads/stores in local register allocation," *IEEE Trans. Software Eng.,* vol. 15, pp. 1252–1260, Oct. 1989.
[13] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.,* vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.
[14] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Trans. Comput.,* vol. C-25, no. 5, May 1976.
[15] M. D. MacLaren, "Inline routines in VAXELN Pascal," in *Proc. ACM SIGPLAN Symp. Compiler Construction,* vol. 19, no. 6, June 1984.
[16] B. Malloy, E. L. Lloyd and M. L. Soffa, "Fine grained scheduling of asynchronous multiprocessors in NP-complete," Tech. Rep. # 89–23, Dec. 1989.
[17] B. Malloy and M. L. Soffa, "Conversion of simulation processes to Pascal constructs," *Software-Practice and Experience,* vol. 20, no. 2, pp. 191–207, Feb. 1990.
[18] F. H. McMohan, "FORTRAN CPU performance analysis," Lawrence Livermore Laboratories, 1972.

[19] C. H. Papadimitriou and J. D. Ullman, "A communication-time trade-off," *Siam J. Computing,* vol. 16, no. 4, Aug. 1987.
[20] T. L. Rodeheffer, "Compiling ordinary programs for executing on an asynchronous multiprocessor," Tech. Rep. No. CMU-CS-85–155, Carnegie Mellon Univ., 1985.
[21] V. Sarkar and J. Hennessy, "Compile time partitioning and scheduling of parallel programs," in *Sigplan Symp. on Compiler Construction,* 1986, pp. 17–26.
[22] V. Sarkar, "Partitioning and scheduling parallel programs for execution on multiprocessors," Tech. Rep. no. CSL-TR-87–328, Standford Univ., Apr. 1987.
[23] ———, Private Communication, Dec. 8, 1989.
[24] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.,* vol. 19, no. 10, pp. 889–895, Oct. 1970.
[25] D. Vrsalovic, D. Seiwiorek, Z. Segall, and E. Gehringer, "Performance prediction and calibration for a class of multiprocessors," *IEEE Trans. Comput.,* vol. 37, no. 11, Nov. 1988.
[26] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," in *Second Int. Conf. Architectural Support for Programming Languages and Operating Syst.,* Oct. 1987.
[27] A. Wolfe and J. Shen, "A variable instruction stream extension to the VLIW architecture," in *Forth Int. Conf. Architectural Support for Programming Languages and Operating Syst.,* Apr. 1991, pp. 2–14.

B. A. Malloy received the B.S. degree in mathematics from LaSalle University in Philadelphia, M. Ed. in counselor education, and M.S. and Ph.D. in computer science from the University of Pittsburgh.

He is currently an Assistant Professor at Clemson University. His research interests include programming language design and implementation, implementation of parallelism, simulation modeling and software maintenance.

E. L. Lloyd received B.S. degrees in both computer science and mathematics from the Pennsylvania State University in 1975, and the S.M. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology in 1977 and 1980, respectively.

He is presently an Associate Professor in the Computer and Information Science Department of the University of Delaware. Earlier, he had been an Associate Professor at the University of Pittsburgh (1980–1988), and served as Program Director for the Computer and Computation Theory Program at the National Science Foundation (1988–1989). His research interests are in the design and analysis of algorithms, with a particular emphasis on approximation algorithms for NP-hard problems.

Dr. Lloyd has published numerous journal and conference papers. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

M. L. Soffa received the Ph.D. degree in computer science from the University of Pittsburgh in 1977.

Since that time, she has been a faculty member at the University of Pittsburgh and is currently a Professor in the Computer Science Department. Since 1991, she has been also serving as the Dean of Graduate Studies in Arts and Sciences at Pitt. Her research interests include language implementation, parallelizing compilers, program analysis, and software tools.

Dr. Soffa currently serves on the editorial boards of *ACM Transactions on Programming Languages and Systems,* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, and *International Journal of Parallel Programming, Computer Languages.*