A Fine Grained Approach to Scheduling Asynchronous Multiprocessors

Brian Malloy  
Clemson University  
Clemson, SC 29634  
(803) 656-0808  
malloy@clemson.edu

Errol L. Lloyd  
University of Delaware  
Newark, DE 19716  
(302) 451-1958  
elloyd@dewey.udel.edu

Mary Lou Soffa†  
University of Pittsburgh  
Pittsburgh, PA 15260  
(412) 624-8425  
soffa@cs.pitt.edu

Abstract - A new approach is given for scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors. 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 and then compared to our approach using simulated executions. In each instance, our method is found to provide significantly shorter schedules. Second, the simulations are used to examine the effects of various architectural considerations on the execution of the schedules. Our approach provides significant speedup in a wide-range of situations. Third, schedules produced by our approach are executed on a two-processor Data General shared memory multiprocessor system, indicating a substantial manner when scheduling a sequential instruction for execution "in parallel" on asynchronous multiprocessors.

1. Introduction

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. We focus on exploiting fine grained parallelism to schedule a sequential instruction stream for execution on an asynchronous multiprocessor system. Asynchronous multiprocessors execute independently and communication is performed explicitly through asynchronous communication primitives. 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. 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.

In the next section, we provide specifics on the computational/architectural model assumed and a discussion of scheduling in this context. In section 3, we present our approach, the Preferred Path Selection algorithm (PPS), for fine grained scheduling on asynchronous multiprocessors. In section 4, we study the performance of our approach in relation to existing methods. In section 5, further simulation techniques are used to determine the performance of the PPS for varying communication speeds and interconnection structure bandwidths, including the modeling of the contention in the communication structure. Finally, in section 6, schedules produced by our approach are executed on a two-processor Data General AViiON shared memory multiprocessor system.

2. Preliminaries: Models and schedules

In this section, we provide specifics on the computational/architectural model, and discuss scheduling in this context.

2.1. The computational/architectural model

To accurately evaluate the quality of the schedules that we produce, we assume that the multiprocessor system consists of p asynchronous homogeneous processors, shared global memory modules, and a communication structure that allows processors to communicate with other processors or with the shared memory. We also assume the standard primitives send and receive, which are used for the synchronization of processors and that the send operation does not require the invoker to wait until a corresponding receive is executed.

In conjunction with the above system, we employ three parameters that describe the "speed" of the architecture. The first is a function $F_s(I)$ that returns the number of cycles required to execute instruction $I$. The second is a function $F_w = F_s + F_w$, 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, register channels or an interconnection network that provides support for communication of values. The function $F_s$ 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 4 and 5, takes the parameters $F_s$, $F_w$, and BW as inputs.

In a portion of what follows, we use an idealized version of the above in order to isolate the important issues involved in fine grained scheduling. In this UECC or uniform execution

† This work was partially supported by the National Science Foundation through grant CCR-910089 to the University of Pittsburgh.
and communication cost model, the following conditions hold: 1) $F_i = 1$ for every instruction $i$. 2) $F_i = 1, 3) F_i = 0$. 4) $GW = p$, and 5) Synchronization primitives $S_i$ and $S_0$ can execute in the same cycle. The first condition provides for the execution of any operation in one cycle, and the second condition allows communication through the interconnection structure in one cycle. The third condition allows $p$ processors to communicate simultaneously without contention; such throughout might, for example, be provided by a crossbar interconnection topology. The fourth condition allows one cycle for each processor to execute a synchronization primitive. We use a directed acyclic graph (dag) $G = (V, E)$ to represent the computation performed in a basic block. In each dag, the nodes correspond to computed values, and the arcs indicate data dependencies between values. We view the dag as shown in Figure 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.

\[\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10
\end{array}\]

Figure 1. Sample dag and schedule produced using PPS.

2.2. 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 four phase approach for scheduling dags on asynchronous machines. 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 its parents). Third, these lists are modified by incorporating the required communication primitives. Finally, these lists are used to produce a schedule. Phase one is the main focus of this paper. Phase 2 is straightforward and is not discussed here. The remainder of this section is devoted to a discussion of phases 2 and 4.

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 than its corresponding send operation (which is on another processor). Clearly this means that some idle time may exist on the processor executing the receive. The length of a schedule is the latest time slot during which a node of $G$ executes. For example, in Figure 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.

3. The Preferred Path Selection Algorithm

In this section we describe our algorithm for the scheduling of program dags on asynchronous multiprocessors. We limit the discussion here to "phase 1", the assignment of each node to some processor. 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. The algorithm that we present 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 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 ensuring that in the construction of $L_i$, for execution on processor $P$: 1) that nodes with a parent unassigned or assigned to $P$ are assigned 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 Figure 2; an input of a dag $G = (V, E)$ and a multiprocessor with $p$ processors is assumed.

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

```
BEGIN

For each level of $G$:

1. Find BestNode for level
2. Assign BestNode
3. Find new BestNode
4. Assign new BestNode

END

```

To illustrate the manner in which the PPS assigns nodes to processors, we use it to schedule the dag shown in Figure 1 on two processors. 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 loop, node 1 is assigned to $P_1$. In the next iteration, a child of node 1, say node 2, is chosen as BestNode and is assigned to $P_1$. In the next iteration of the inner loop, node 4 is assigned to $P_1$ and this loop terminates since all children of node 4 are assigned. Thus, path 1, 2, 4 is the dag is assigned to $P_1$. The PPS algorithm continues execution in the outer loop by updating $i$ to 2 indicating that we are now assigning nodes to processor two. $k$ is also updated to 2, since nodes 3 and 8 are unassigned. At the top of the outer 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 loop and the inner loop terminates.
with path 8, 9 in the dag assigned to P8. In the outer 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 P1. BestNode is assigned to P1 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 P1. The PPS algorithm continues until all nodes in the dag are assigned. The schedule resulting from this assignment is shown in Figure 1.

4. Performance using the Model

In this section we study the performance of our approach in relation to other methods. We first compare the schedules produced by each of the methods using the UECF model. These results show that the PPS algorithm performs better than the other methods and that the approach scales to 16 processors, provided that the communication structure contains sufficient parallelism. In the second portion of this section we provide evaluations of the PPS approach under a range of architectural assumptions including the modeling of the contention in the communication structure.

4.1. Simulation results using UECF model

In this section, we compare the lengths of schedules produced by each of the methods: CP, Early-Scheduling Method, Internatization Prepass and PPS algorithms. In addition, a Random assignment algorithm is included to serve as a "control" for the comparison of the heuristics. All of the comparisons in this section use the UECF model. 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 found in 4). For example, Sample is a program whose corresponding dag contains 10 nodes as shown in Figure 1. Applying CP to Sample resulted in a 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 (Figure 1).

From Table 1, it is clear that our PPS algorithm produces shorter schedules than the other methods. We believe that the 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 PPS 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 global-view of the dag in its computation of a schedule. The earlier list scheduling algorithms utilize a much more local-view, in examining primarily, nodes on a single level to decide which to schedule next.

The PPS algorithm is able to provide good speedup, not only for two processors (Table 1), but also for 8 and 16 processors (Table 2). In our evaluations, the PPS provided good 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.

Table 2. Scalability of the PPS Algorithm

<table>
<thead>
<tr>
<th>Program</th>
<th>No. in dag</th>
<th>Avg Indegree</th>
<th>No. of Comp</th>
<th>SpUp p=8</th>
<th>SpUp p=16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>20</td>
<td>1.30</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Pyramid</td>
<td>36</td>
<td>1.25</td>
<td>1</td>
<td>1.89</td>
<td>1.89</td>
</tr>
<tr>
<td>Mat Mult</td>
<td>129</td>
<td>1.23</td>
<td>9</td>
<td>3.53</td>
<td>5.45</td>
</tr>
<tr>
<td>Dual Dag</td>
<td>107</td>
<td>1.43</td>
<td>2</td>
<td>3.98</td>
<td>1.98</td>
</tr>
<tr>
<td>Whetstone</td>
<td>137</td>
<td>1.42</td>
<td>13</td>
<td>2.74</td>
<td>3.26</td>
</tr>
<tr>
<td>FFT</td>
<td>127</td>
<td>0.99</td>
<td>1</td>
<td>6.13</td>
<td>8.64</td>
</tr>
<tr>
<td>Livermore</td>
<td>203</td>
<td>1.30</td>
<td>20</td>
<td>6.34</td>
<td>9.67</td>
</tr>
</tbody>
</table>

4.2. Simulation results for parameterized model

In this section, the values established by Sarkar et al. are used to describe the execution times for simple operations \( F_0(I) \) and the time needed to communicate a value \( F_1(I) \). A table of cost values is used to define the value of the function \( F_1(I) \) for each instruction \( I \). To describe the access time, we use values for \( F_2 \) corresponding to fast, medium and slow access times respectively. In constructing the simulator, we used the process oriented language Simcal. We use various bandwidths 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 and finally, a value of \( p \) describes the best case bandwidth where \( p \) requests can be accepted per cycle.

Table 3. Speedup for PPS -- Mat Mult

<table>
<thead>
<tr>
<th>F</th>
<th>BW</th>
<th>processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=3</td>
</tr>
<tr>
<td>fast</td>
<td>1.69</td>
<td>2.50</td>
</tr>
<tr>
<td>med</td>
<td>( \sqrt{p} )</td>
<td>1.58</td>
</tr>
<tr>
<td>slow</td>
<td>( \sqrt{p} )</td>
<td>1.31</td>
</tr>
</tbody>
</table>

The results of these simulation studies again show that in comparison with the other methods, the PPS approach produces better schedules. Since these results are very similar in nature to those of the previous section, we omit them. We do however present, in Tables 3 and 4, simulation results for the
PPS itself. These tables present the speedup obtained by executing the run-time schedules for Mat Mult and FFT on 2, 3, 4, 8 and 16 processors using a fast, medium and slow communication structure with a bandwidth of 1, √3 and p.

5. Performance of PPS on a Data General Multiprocessor

The PPS algorithm was implemented on a Data General AViiON shared memory multiprocessor system equipped with a unibus communication structure and two homogeneous processors. The send and receive primitives were implemented using spin-lock operations on Unix shared variables. In order to compare the results of these actual executions, 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_i.

The results summarized in Table 5 indicate a strong correlation between the simulations and the actual executions on the Data General machine. The first column of Table 5 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. As a particular instance, note that 54 cycles are required to simulate the sequential code of the Fibonacci program, and that 60 cycles are required to simulate the schedule for 2 processors with a resulting speedup of 0.90 over the sequential execution. 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 0.92 over the sequential execution. The similarities in speedup between the simulation and actual execution results can be seen in columns 4 and 7.

Table 5. Comparison of simulation with actual execution.

<table>
<thead>
<tr>
<th>Test</th>
<th>Simulations using Parameterized Cost Model</th>
<th>Actual execution on Data General Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Time (p=1)</td>
<td>Time (p=2)</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>Pyramid</td>
<td>102</td>
<td>113</td>
</tr>
<tr>
<td>Mat Mult</td>
<td>336</td>
<td>277</td>
</tr>
<tr>
<td>Dual Dog</td>
<td>311</td>
<td>160</td>
</tr>
<tr>
<td>Whetstone</td>
<td>411</td>
<td>300</td>
</tr>
<tr>
<td>FFT</td>
<td>506</td>
<td>325</td>
</tr>
<tr>
<td>Livermore</td>
<td>643</td>
<td>381</td>
</tr>
</tbody>
</table>

6. Conclusions

We have provided a new approach for scheduling a sequential instruction stream for execution "in parallel" on asynchronous multiprocessors. Our approach was compared to adapted existing methods showing that our method provides superior schedules to each of the alternative methods. In addition to the simulation studies, the PPS algorithm was implemented on the Data General AViiON shared memory multiprocessor where executions of PPS generated schedules produce speedups that 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.

References