Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

THE VALIDATION OF A MULTIPROCESSOR SIMULATOR

Brian A. Malloy

Department of Computer Science
Clemson University
Clemson, South Carolina 29634-1906

ABSTRACT

In this paper, we present the design and implementation of a multiprocessor simulator written in the language *SimCal*. We use the simulator to test our scheme to partition a sequential program for parallel execution on a shared memory, asynchronous multiprocessor. The results of the simulations indicate that our partitioning scheme can provide significant speed-up by executing the program in parallel. We then execute the partitioned program on an actual multiprocessor and find a high degree of correlation between the simulations and the actual executions. This correlation serves to validate our simulator. We then use the multiprocessor simulator to hypothetically extended the actual multiprocessor and we show that adding more processors will not provide significant improvement in the parallel executions unless the communication structure is also improved to contain more parallelism.

1. 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. However, it has been difficult to test these new developments in parallelism on a target architecture since the architecture is often not available, or too expensive for the researcher to obtain. One approach to solving the problem of unavailability of the target architecture is to use a simulator to capture the behavior of the architectural system. A problem with the use of simulators is the possibility that the simulations do not adequately capture the system due to the omission of an important factor in the system or because not all of the factors in the target system are adequately known. In this event, the simulations may allow a researcher to arrive at an erroneous conclusion about the power of the parallel system under scrutiny. Also, despite the researchers care in capturing every detail of the target system, other researchers are sometimes sceptical about the reliability of the simulation approach.

We present the design of a multiprocessor simulator. We briefly discuss our techniques to partition a sequential program into threads for parallel execution on a shared memory, asynchronous multiprocessor. We then use the simulator to execute the threads for parameters that describe an actual multiprocessor system, the Data General AViiON[DataGeneral1990]. The correlation that we obtain between the simulations and the actual executions verify that the multiprocessor simulator captures the important factors of the multiprocessor system. Having validated the simulator, we use it to hypothetically extend the Data General AViiON to determine the degree of speed-up that might be obtained if the multiprocessor were configured differently, for example, by adding more processors to the AViiON.

The multiprocessor simulator that we present is coded in a simulation language, *SimCal*[Malloy1986], that is based on Simula. Simula is a powerful, process oriented simulation language that possesses a high degree of expressibility.

The paper is organized as follows. In section 2, we briefly describe *SimCal*, the simulation language that we use to implement the multiprocessor simulator. In section 3 we discuss the computational model that captures the important features of the multiprocessor system under study. We then briefly describe the parallel threads that are executed on the multiprocessor system followed by the design and construction of the multiprocessor simulator. In section 4, we describe our validation of the simulator through the comparison of the results of the simulations with the results obtained by executing the threads on an actual multiprocessor, the Data General AViiON. Also, in section 4, we hypothetically extend the AViiON through the use of the simulator. Finally, in section 5 we draw conclusions.

2. DESCRIPTION OF *SimCal*

SimCal is a process-driven simulation language that is based on standard Pascal. The *SimCal* language is extended to directly incorporate simulation primitives designed to have essentially the same syntax and semantics as those found in Simula. Therefore, a *SimCal* user, knowledgeable in Pascal, need only consult previous work[Malloy1990, Malloy1986] or a Simula reference text[Lamprecht1983] for information regarding the syntax and semantics of the simulation primitives. The simulation primitives are directly incorporated into Pascal, meaning that the user is not responsible for adding any calls to system procedures or declaring any extra data structures. This is all handled by a preprocessor for *SimCal* that takes a *SimCal* program and translates it into a standard Pascal program.

We chose Pascal as the base language because it is a widely used language. The simulation primitives are based on Simula because it a powerful, process-oriented simulation language. The *SimCal* language was designed as a preprocessor so that it can be used in any environment that has a Pascal compiler and therefore obviates any additional software cost. As a preprocessor, *SimCal* sits on top of the Pascal compiler and therefore requires no alterations to the compiler in any way.

2.1. Using *SimCal* Language Primitives for Simulation Modeling

We have previously described the design and implementation of *SimCal*[Malloy1990], and the reader interested in the preprocessor construction may consult this work. We do not discuss the *SimCal* design and implementation in this paper but rather we summarize the actions of the language primitives and demonstrate how they can be used to construct a simulation model. The discussion in this section will facilitate our discussion of the multiprocessor simulator described in the next section.

Because *SimCal* is a process-driven simulation language, there are language facilities to support the creation and manipulation of processes. A system clock and an event list ordered by time are included in the language. Since it is essential to express the relationships among processes in a simulation, the Simula list facility is also included.

A process in *SimCal* is represented by a special "procedure like" block of code called a PROCESS. A process may be acted upon by the simulation primitives ACTIVATE, ACTIVATE AT, PASSIVATE and HOLD. These primitives insert processes into or remove processes from the event list. Processes also may be inserted or removed from user defined lists. *SimCal* provides primitives INTO, OUT, FIRST, EMPTY and CARDINAL that may be used to examine or manipulate the user defined lists.

The use of the simulation and list primitives in the example in Figure 1 illustrate how control can be passed among processes both explicitly and implicitly. Control is passed explicitly through the use of the ACTIVATE primitive and implicitly through the use of PASSIVATE or HOLD. All simulation primitives pass control by using the event list, where the process at the head of the event list is the currently active process. For example, the ACTIVATE B primitive inserts process B at the head of the event list and passes control to an event manager which always activates the process at the head of the event list. Only the HOLD primitive can increment system time.

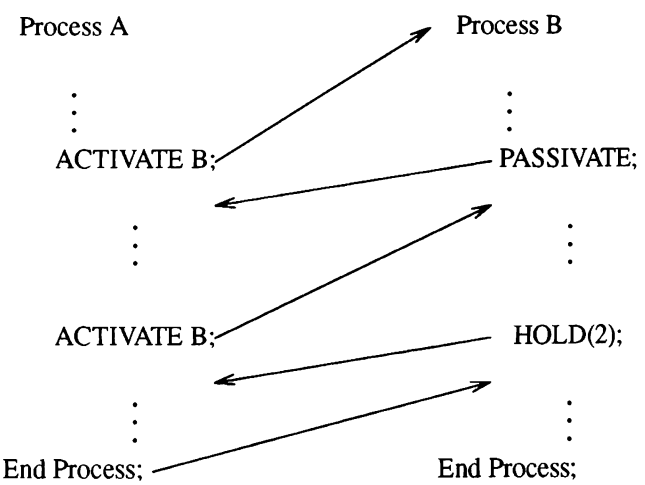

Figure 1: Process synchronization using primitives

3. DESIGN OF THE MULTIPROCESSOR MODEL

In this section we begin by presenting the computational model that forms the basis for our target architecture. We then give a brief explanation of the technique used to partition a sequential program into threads for parallel execution on a shared memory, asynchronous multiprocessor. Finally, we present the parameterized multiprocessor simulator that we construct from the computational model. This parameterized multiprocessor simulator is used to simulate execution of our parallel threads, constructed by partitioning a sequential program.

3.1. The Computational Model

In order for us to accurately evaluate the quality of the schedules that we produce, it is necessary that we be precise about certain aspects of the asynchronous processor system that we utilize. In particular, we assume that such a system 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. An example of such a communication structure is a **bus** that typically allows a single processor to communicate values to memory. We assume that the 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[Dinning1989].

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[Kowalik1985], register channels[Gupta1990] or an interconnection network[Lang1976] 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 multiprocessor simulator discussed in this paper takes the parameters p, F_e, F_c, and BW as inputs. We will discuss the computational model in more detail in section 4.

3.2. Threads

We have developed techniques to partition sequential code for parallel multiprocessor execution [Malloy1992a, Malloy1994, Malloy1992b]. We now present key ideas of the technique for partitioning straight line code, such as the code found in basic blocks, into threads for parallel execution[Malloy1994, Malloy1992b]. The interested reader may consult our previous work[Malloy1992a] for a discussion of partitioning entire programs for asynchronous multiprocessor execution.

Time	1	2	3	4	5	6	7
P1	1	Rv_2	3	Sd_3	4	Rv_5	6
P2	2	Sd_2		Rv_3	5	Sd_5	

Figure 2. A schedule where nodes 1, 3, 4, and 6 are assigned to list P1 and nodes 2 and 5 are assigned to list P2.

A schedule is obtained in the following manner: 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). 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 Figure 2. In the schedule presented in Figure 2, we use a *unit model* where 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 an operation executes. For example, in Figure 2, the length of the schedule is 7. In a simulated schedule, the time to execute any particular operation may vary due to factors such as contention in the communication structure. For example, in Figure 3, 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 multiprocessor.

Time	1	2	3	4	5	6	7	8	9	10
P1	1	Rv_2	Rv_2	3	Sd_3	4	4	Rv_5	Rv_5	6
P2	2	Sd_2			Rv_3	Rv_3	5	Sd_5		

Figure 3. A possible run-time schedule for the compile-time schedule in Figure 2. 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.

3.3. Construction of the Multiprocessor Simulator

As discussed in the previous section, we construct a schedule consisting of tasks where each task represents a node in a dag or an operation such as an intermediate code operation. We now present the multiprocessor simulator which simulates the parallel execution of the schedule. Figure 4 summarizes the simulator.

For the simulator of Figure 4, execution begins in the main program by supplying the parameters p, F_e, F_c and BW, as discussed in the previous section. Since the simulator must actually execute the statements in the schedule, the second statement in the main program in Figure 4 initializes an *Interpreter* that actually executes the instructions in the schedule. After initializing the *Interpreter*, the simulator then reads the parallel schedule or *threads*, one thread for each processor. In executing the loop in the main program of the simulator, the CREATE primitive is used to instantiate p processors (*cpu*) and the ACTIVATE primitive is used to begin execution of thread$_i$ in the respective processor cpu$_i$. In *SimCal*, as in Simula, the main program is itself a process and must not be allowed to terminate before any child processes terminate, since that would cause the entire program to terminate prematurely. Thus, the final simulation primitive executed in the main program is HOLD(50000), which inserts the main program at the end of the event list, allowing each of the cpu$_i$ an opportunity to execute the respective threads. Execution will resume in main after all of the cpu$_i$'s have terminated. At that time, any statistics that may have been gathered during the simulation, such as the total time spent waiting to access the bus, may be output.

In addition to the main program of the simulator, a summary of the actions of each processor (cpu$_i$) is also illustrated. Each cpu$_i$ is itself a PROCESS that, through the use of the event list, can execute in "parallel". The important part of cpu$_i$ is a **while** loop that contains a **case** statement that chooses the actions to be performed by the simulator. The important operations listed are: *send*, *receive*, store, load and *operation*. By *operation*, we mean an intermediate or assembly code operation.

To illustrate the actions of the multiprocessor simulator, we will now discuss the *send* primitive listed in the **case** statement in PROCESS *cpu* shown in Figure 4. The first action of the *send* primitive is to "wait to access the bus" as described above. Having gained access, the next action of the *send* primitive is to increment busCount to update the number of processors currently using the bus. Then, the synchronization bit corresponding to the data value being communicated is set to indicate to the receiving process that the value is "ready". Having "sent" the data, the next action in implementing the *send* is to HOLD for the number of cycles that are required in the *send* operation; this execution of the HOLD will update system time appropriately. In the early stages of the multiprocessor simulations, we executed the HOLD primitive for the *send* operation for the number of cycles that we felt were reasonable. Later, as we will discuss in the next section, we conducted experiments on an actual multiprocessor to provide greater accuracy for our simulations/predictions. The final action of the *send* primitive is to decrement the busCount to indicate that this processor is now relinquishing the bus. The actions of the other operations listed in PROCESS *cpu* are similar to the *send* primitive.

4. VALIDATION OF THE MULTIPROCESSOR SIMULATOR

In the previous section we presented the design of the parameterized multiprocessor simulator. In this section, we validate the simulator by using it to simulate the executions of schedules produced by our algorithm to partition sequential code into threads for parallel execution. This is achieved by supplying appropriate values for the parameters p, $F_e(I)$, F_c, and BW, to the simulator that we constructed using *SimCal*[Malloy1990].

4.1. Performance of the Partitioning Scheme on a Data General Multiprocessor

In order to determine the performance of our partitioning scheme on a "real" multiprocessor, we executed our parallel threads on a Data General AViiON shared memory multiprocessor system[DataGeneral1990] equipped with a unibus communication structure and two identical processors. As we will show, we obtained an excellent correlation between these "actual executions' and the simulations, thereby validating our multiprocessor simulator. The *send* and *receive* primitives were implemented on the AViiON using spin-lock operations on *unix shared variables*[Bach1986]. In order to obtain the parameters for our simulator, 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_e for the simulation studies described below.

```
program multiprocessor;
const maxProcessors = 16;
ref (cpu) array processor [1..maxProcessors].

PROCESS cpu( i : integer );
var pc :integer;
begin
  while there are more statements to execute in thread_i do
  begin
      case this statement in thread_i of
            send        : wait to access the bus;
                          increment the busCount;
                          set synchronization bit for this data value;
                          HOLD(cycles required for send);
                          decrement the busCount;
            receive     : (* we are examining a synchronization bit stored in local memory *)
                          while synchronization bit for this data value is not set do
                                HOLD(1);
                          end (* while *)
                          HOLD(cycles required for receive);
                          reset synchronization bit for this data value;
            store       : wait to access the bus;
                          increment the busCount;
                          pass this operation to Interpreter
                          HOLD(cycles required for store);
                          decrement the busCount;
            load        : (* similar to store *)
            operation   : pass this operation to Interpreter, and
                          HOLD for the number of cycles specified by F_e(operation);
      end; (* case *)
      increment pc to indicate appropriate statement in thread_i;
  end; (* while *)
  writeln('processor ', i, ' terminates execution at ', time);
end; (* cpu *)

begin (* main *)
      input parameters to specify F_e(I), F_c, BW and p;
      initialize Interpreter;
      input threads;
      for i:= 1 to p do
            processor[i]:- CREATE cpu(i);
            ACTIVATE processor[i];
      end;
      HOLD(50000);
      output statistics;
end.
```

Figure 4. Summary of the Multiprocessor Simulator.

Table 1.
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

The results summarized in Table 1 indicate a very strong correlation between the simulation results and the actual executions on the Data General multiprocessor. In Table 1, 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 speed-up of 0.90 over the sequential execution. Note that a speed-up 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 speed-up of 0.88 over the sequential execution.

The similarities in speed-up 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 speed-ups is never more than 0.06. 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 1 also supports the conclusion that the our partitioning scheme is able to provide good speed-up for programs containing sufficient parallelism. Sufficient parallelism implies that the sequence of code being scheduled does not contain a large number of data dependencies and has enough parallelism to support all or most of the processors.

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 partitioning scheme 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 2 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 speed-up 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 2 reveals a "leveling off" effect in the ability of the scheduler to provide speed-up 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.

Table 2.
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

5. CONCLUSIONS

We have reported our experiences in developing a multiprocessor simulator using the process oriented language *SimCal*. We used the simulator to test our scheme to partition a sequential program for parallel execution on a shared memory, asynchronous multiprocesor. The results of the simulations indicate that our partitioning scheme can provide significant speed-up in the parallel execution of a program. We also executed the parallelized program on a Data General multiprocessor where the speed-ups on the actual machine correlated very closely with the simulations. This correlation served as a validation for our simulations. We then used the simulator to hypothetically extend the Data General machine by adding more processors and a communication structure that provided more parallelism. We concluded that the parallel execution of the program would not achieve any significant speed-up simply by adding processors.

ACKNOWLEDGEMENTS

The author wishes to thank Mary Lou Soffa, whose guidance and inspiration has made an invaluable contribution to this work. Also, special thanks to Fred Harris for his work in implementing the scheduling technique on the Data General AViiON multiprocessor and for gathering the statistics in Table 1.

References

Maurice J. Bach (1986), *The Design of the Unix Operating System*, Prentice-Hall Inc.

Anne Dinning (1989), "A Survey of Synchronization Methods for Parallel Computers," *Computer*, pp. 66-76.

Data General (1990), *Installing and Managing the DG/UX System*, Data General Corporation.

Rajiv Gupta (1990), "Employing Register Channels for the Exploitation of Instruction level Parallelism," *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle Washington.

Janusz S. Kowalik (1985), "Parallel MIMD Computation: HEP Supercomputer & Its Applications," *Scientific Computation Series*, MIT Press.

G. Lamprecht (1983), *Introduction to Simula 67*, Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden.

Tomas Lang (1976), "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," *IEEE Transactions on Computers*, vol. C-25, No 5.

B. A. Malloy and M.L. Soffa (1986), "Simcal: The Merger of Simula and Pascal," *Proceedings Winter Simulation Conference*, pp. 397-403, Washington D. C.

B. A. Malloy and M. L. Soffa (1990), "Conversion of Simulation Processes to Pascal Constructs," *Software - Practice and Experience*, vol. 20(2), pp. 191-207.

B. A. Malloy, E.L. Lloyd, and M.L. Soffa (1992b), "A Fine Grained Approach to Scheduling Asynchronous Multiprocessors," *4th International Conference on Computing and Information*,, pp. 131-135.

B. A. Malloy, R. Gupta, and M. L. Soffa (1992a), "A Shape Matching Approach for Scheduling Fine-Grained Parallelism," *MICRO-25, The 25th Annual International Symposium on Microarchitecture*, pp. 131-135.

B. A. Malloy, E. L. Lloyd, and M. L. Soffa (1994), "Scheduling Dags for Asynchronous Multiprocessor Execution," *IEEE Transactions on Parallel and Distributed Computing.*