Proceedings of the 1985 Winter Simulation Conference
D. Gantz, G. Blais, S. Solomon (eds.)

THE IMPLEMENTATION OF THE HIERARCHICAL ABSTRACT SIMULATOR ON THE HEP COMPUTER

Arturo I Concepcion
Department of Computer Science
Michigan State University
E. Lansing, MI 48824

A methodology is being developed to map the hierarchical abstract simulator onto distributed simulator architectures. The hierarchical abstract simulator is a multicomponent, multilevel discrete event models communicating via message passing. This paper reports on an alternative mapping realization of the hierarchical abstract simulator by using DENELCOR's FORTRAN 77, an extension of FORTRAN 77 for parallel programming, on the Heterogeneous Element Processor (HEP) computer. Several runs were made on the implementation and it was found out that there are three constraints that affect the performance (execution time) of the HEP implementation: number of processors available, degree of synchronization and intercommunication, and workload.

1. Introduction

A distributed simulation methodology based on Discrete Event Specification System, DEVS, [9] was introduced in Concepcion [2] in which multicomponent discrete event models may be simulated by employing multiprocessor architectures. The main thrust of that research is the mapping of the hierarchical multicomponent models onto distributed simulators so that correct and efficient simulation is obtained. The advantages of such distributed simulators over conventional sequential simulation are:

1. The mapping of a network of discrete event components onto the network of processors can better preserve its structure. In the best case, each processor might represent a single model component. This enhances comprehension of the simulator-model relationship, and therefore also, simulation experimentation and model exploration.
2. Advantage may be taken of intrinsic parallelism in the operation of model components by having concurrent execution by each processor of its component's state transitions.

The distributed simulation methodology consists of 5 layers and 4 steps. The lowest layer is the real system to be simulated. By means of the DEVS formalism, the real system is specified as a distributed model. This produces the second layer. From the specification of the distributed model, a transformation is applied to obtain the hierarchical abstract simulator. This third layer is the interpretation of the dynamics specified by the DEVS formalism. The fourth layer is reached by applying to the hierarchical abstract simulator a schema for synchronization and intercommunication among components. This fourth layer is called the distributed simulator. Finally by a mapping process, the distributed simulator is implemented on a hardware/software architecture.

In Concepcion [2], a design of a distributed simulator was proposed, the Hierarchical Multi-Bus Multiprocessor Architecture (HM^2A). This design can be readily implemented with off-the-shelf technology and directly reflects the abstract simulator specification. The architecture is designed around a primitive which is a cluster of processing elements communicating via a local bus and each cluster communicates via inter-cluster bus. Mapping the hierarchical abstract simulator onto the proposed architecture was shown to be a straight forward recursive manner [3].

This paper discusses the fourth step in the distributed simulation methodology, mapping the hierarchical abstract simulator onto a hardware/software architecture. This step serves as a convenient starting point in studying a variety of alternative physical simulator implementations. Also this paper presents an alternative realization of the hierarchical abstract simulator by using DENELCOR's FORTRAN 77, an extended FORTRAN for parallel programming, on the Heterogeneous Element Processor (HEP) computer. Section 2 reviews the dynamics of the hierarchical abstract simulator and its algorithms while in section 3, the translation of the algorithms to DENELCOR's FORTRAN 77 is discussed. Section 4 presents the performance (execution time) of the implemented hierarchical abstract simulator on the HEP computer. Finally, section 5 proposes future directions on this work.

2. Hierarchical Abstract Simulator

The hierarchical abstract simulator is an intermediate state in realizing the model on a physical implementation of the distributed simulator. The hierarchical abstract simulator consists of a network of coordinators where each controls a set of subordinates. If a subordinate is also a coordinator, then it too controls a set of subordinates, and so on. A subordinate which is not a coordinator is called a simulator. The algorithms for the hierarchical abstract simulator define the procedure in computing the state of the DEVS component, updating the simulation time and scheduling new events.

Six types of messages were identified in [2] as sufficient for current execution of DEVS models: (x,τ), $(*,\tau)$, (o,τ), (y,τ), *done* and t_N respectively, these carry external event information, internal event notices, output information, processor termination and next event information. In this paper, the (o,τ) message is not included in the implementation. The (o,τ) message is used to increase the degree of parallelism in the hierarchical abstract simulator when several simulators have output available from the last computation. These messages are exchanged among the coordinators in the interior and root of the hierarchical structure and the workhorse simulators at its leaves. The routing tables and code schemes for the coordinators and the process descriptions for the simulators were specified in terms of functional units to facilitate their realization at the implementation layer. The resulting logical structure was shown to be a correct implementation scheme, and to be free of interferences and deadlocks [2]. This contrasts with other approaches which attempt to maximize parallelism by loosening up on the strict timing requirements of simulation. These approaches must necessarily allow for rolling back the simulation when an out-of-sequence event is detected. In summary, our approach aims for simplicity and uniformity of design, with guaranteed deadlock prevention.

Procedurally, the algorithms that describe the dynamics of the hierarchical abstract simulator are given in Figures 1 and 2. Note that each algorithm is guarded by a lock/unlock operation to assure mutual exclusion.

The following is a list of variables used in the algorithms:

t_L	= time of last event.
τ	= global time.
t_N	= time to next event.
ta	= time advance function.
i^*	= immiment component (minimum t_N).
e	= elapsed time in this state.
s	= state of the model component.
δ_{ext}	= external transition function.
δ_{int}	= internal transition function.
y	= output from model component.
λ	= output function.
(x,τ)	= input external message x occuring at time τ.
$(*,\tau)$	= input internal message occuring at time τ.
(y,τ)	= output message occuring at time τ.
EXT_IF TABLE	= external interface table.
INT_IF TABLE	= internal interface table.
OUT_IF TABLE	= output interface table.
MINTN	= function that determines the minimum t_N.

There are two groups of algorithms, one for a coordinator and one for a simulator. Each group is divided into: *when receiving an* (x,τ) *message* and *when receiving an* $(*,\tau)$ *message*. The following gives a summary of the actions taken by the components of the hierarchical abstract simulator when receiving a message.

When a simulator receives an $(*,\tau)$ message: it checks first the simulation time t, then it sends its output as (y,τ) to its coordinator. Simultaneously, the simulator computes its new state which includes determining a new t_N which is sent to the coordinator. At termination of computation, the simulator sends its *done* signal.

```
1.  when receive an input (x,$\tau$):
2.     lock (bit)
3.     done := false
4.     if $t_L \le \tau \le t_N$ then
5.          [ e:= $\tau - t_L$
6.          s:= $\delta_{ext}$(s,e,x)
7.          $t_L$:= $\tau$
8.          $t_N$:= $t_L$+ ta(s)
            ]
9.     else error
10.    done := true
11.    unlock (bit)
12. end when receive
```

(a) Algorithm when receiving a (x,τ) message.

```
1.  when receive an input (*,$\tau$):
2.     lock (bit)
3.     done := false
4.     if $\tau = t_N$ then
5.          [ cobegin
6.               y:= $\lambda$ (s)
7.               send (y,$\tau$) to coordinator
8.               s:= $\delta_{int}$(s)
9.            coend
10.         $t_L$:= $\tau$
11.         $t_N$:= $t_L$+ ta(s)
            ]
12.    else error
13.    done := true
14.    unlock (bit)
15. end when receive
```

(b) Algorithm when receiving a $(*,\tau)$ message.

Figure 1: Algorithms for a Simulator

```
1.  when receive an input (x,$\tau$):
2.     lock (bit)
3.     done := false
4.     if $t_L \le \tau \le t_N$ then
5.          [ send input ($x_i$,$\tau$) to all the affected
              simulators i via a   table look-up of
              EXT_IF TABLE
6.          wait until all simulators i's done are true
7.          $t_L$:= $\tau$
8.          $t_N$:= MINTN(all subordinates under
                 coordinator)
            ]
9.     else error
10.    done := true
11.    unlock (bit)
12. end when receive
```

(a) Algorithm when receiving a (x,τ) message.

```
1.  when receive an input (*,$\tau$):
2.     lock (bit)
3.     done := false
4.     if $\tau = t_N$ then
5.          [ send the input (*,$\tau$) to $i^*$
8.            when receive an input Y sent by $i^*$
9.               :Y is used in the same enclosure:
10.                   send the message (x,$\tau$) to each $i^*$
                      influencees via a table look-up
                      of INT_IF TABLE
11.                   wait until $i^*$influencees' done
                      are true
12.              :Y is used outside of the enclosure:
13.                   send the message (y,$\tau$) to the
                      next level coordinator via a
                      table look-up of OUT_IF TABLE
14.              :wait until $i^*$done is true
15.           end when receive
23.         $t_L$:= $\tau$
24.         $t_N$:= MINTN(all subordinates under
                 coordinator)
            ]
25.    else error
26.    done := true
27.    unlock (bit)
28. end when receive
```

(b) Algorithm when receiving an input $(*,\tau)$ message.

Figure 2: Algorithms for a Coordinator

When a coordinator receives a $(*,\tau)$ message: it checks the simulation time, t, then it sends the $(*,\tau)$ message to the component with the minimum t_N. This component is called the imminent component. The coordinator then waits for *done* signal from the imminent component. Afterwhich the coordinator proceeds to determine the new imminent component.

When a simulator receives an (x,τ) message: it checks the simulation time first and then it computes its new state, s. A new t_N is determined which is sent to the coordinator. At termination, the simulator sends *done* signal to the coordinator.

When a coordinator receives an (x,τ) message: it performs a check on the simulation time and then it sends the reformatted (x,τ) message to the affected subordinate by a table look-up of EXT_IF TABLE. The coordinator waits for all affected components to send *done* signals. Afterwhich the coordinator proceeds to determine the new imminent component.

When a coordinator receives a (y,τ) message from its subordinate, it determines whether this message is used within its enclosure or not. If the message is used within, then the coordinator sends the (y,τ) message as an (x,τ) message to the affected subordinate by a table look-up of the INT_IF TABLE otherwise, by a table look-up of OUT_IF TABLE, the coordinator sends the message (y,τ) to the next higher level coordinator.

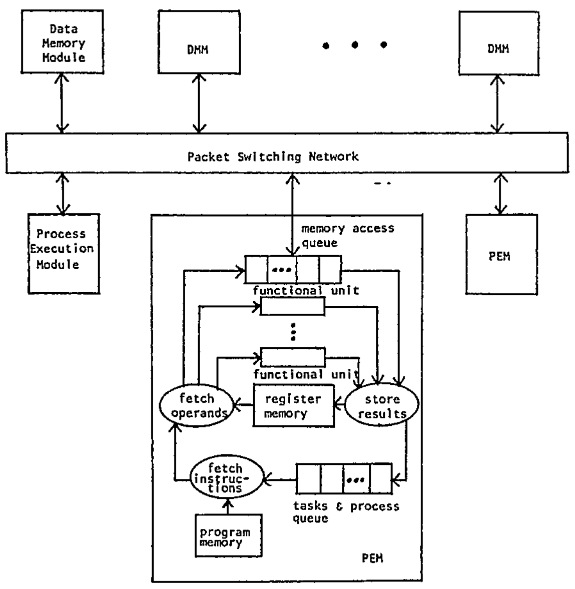

Figure 3: The HEP Functional Organization

3. Implementation on the HEP

The architecture of the Heterogeneous Element Processor (HEP) has been described in [6,7]. As shown in Figure 3, the main components are the Data Memory Module, the Packet Switch Network and the Process Execution Module. A program consists of one or more tasks while each task consists of one or more processes. Each process is composed of a sequence of instructions. Both the tasks and processes are executed in parallel in the HEP while the instructions of each process are executed in sequential pipeline fashion. Each PEM has a program memory where active tasks and processes instruction streams are selected for execution. Up to 50 instruction streams can be active at any given time. Notice that each PEM has a number of functional units which allow pipeline execution of multiple instruction streams for multiple data streams. This makes the HEP computer an MIMD machine.

For software support, HEP has the DENELCOR's FORTRAN 77 [5]. It provides the parallel programming environment for the HEP computer. It generates fully reentrant (sharable) code and provides synchronization among these codes. As shown in Figure 4, CREATE commands initiate processes A, B and C which execute in parallel with the MAIN. Synchronization among these processes is done via F/E (full/empty) bit that is tagged on special shared variables called asynchronous variables. These variables are prefixed with a "$" character. The following are some of the functions of the asynchronous variables:

J = $I, wait for full and set empty (integer).

X = $A, wait for full and set empty (real).

$Y = B, wait for empty and set full.

A = WAITF($B), wait for full, but do not set empty (real).

L = EMPTY($Q), test for empty access state

A = VALUE($Q), read regardless of state and leave unchanged (returns logical result).

For process initiation,

CREATE MYSUB(X,Y,Z), causes referenced subroutine MYSUB to execute in parallel with the creating routine with parameters X, Y and Z.

RETURN, terminates the parallel process executing a subroutine that was CREATEd.

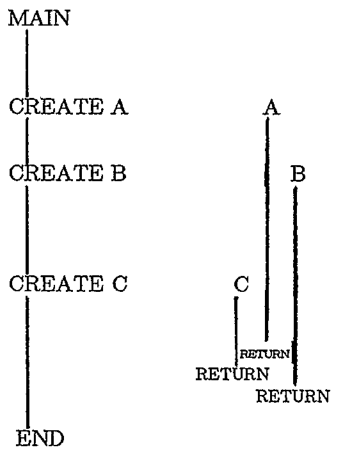

Figure 4: Process Initiation and Termination

Shown below is an example of parallel program in DENELCOR's FORTRAN 77 which creates four parallerocesses and performs all four executions of the subroutine S concurrently.

```
C     MAIN PROGRAM
      COMMON $NP
      PURGE $NP
      $NP = 4
      CREATE PS(1)
      CREATE PS(2)
      CREATE PS(3)
      CALL PS(4)
   20 IF (VALUE($NP).NE.0) GOTO 20
      END
      SUBROUTINE PS(N)
      COMMON $NP
      CALL S(N)
      $NP = $NP - 1
      RETURN
      END
```

The subroutine S is reentrant and the $NP is an asynchronous variable which is used to record the number of processes still executing. When S is finished, $NP is decremented. The MAIN PROGRAM waits until the value of $NP is zero. This means that all processes have finished executing S.

The implementation of the hierarchical abstract simulator on the HEP computer consists of translating the algorithms for a coordinator and for a simulator (see Figures 1 and 2) into the DENELCOR's FORTRAN 77. As seen from these algorithms, there are five types of messages, excluding (o,τ) message, being transmitted: (x,τ), $(*,\tau)$, (y,τ), t_N and *done* messages. The implementation is currently restricted to a binary structure with a maximum of 3 levels. The implementation can be easily expanded to more than 3 levels and applicable to general tree structure. At 3 levels, the hierarchical abstract simulator consists of three coordinators, see Figure 5:

- C_0, C_1 and C_2

and four simulators:

- $S_{1.1}$, $S_{1.2}$, $S_{2.1}$ and $S_{2.2}$

Also there is a process called GEN which generates the messages (x,τ) and $(*,\tau)$.

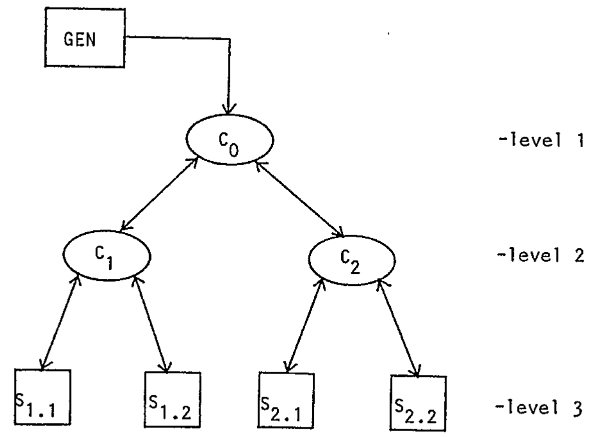

Figure 5: The Hierarchical Abstract Simulator Implemented on the HEP Computer

The main program does the following functions:

1. Obtain from the user the desired assignment of processors and other initialization inputs.
2. CREATE or CALL the processes for the appropriate coordinators and simulators.
3. Initialize the state and control variables of each process CREATEd or CALLed.
4. Perform the function of GEN and test for the termination of the execution.

The following are the inputs to the hierarchical abstract simulator at initialization:

- Specify whether the trace for debugging will be turned off or on.
- Specify the assignment of processors to the 3 leveled hierarchical abstract simulator. This is done by an input string of 7 bits. A 1 in this string means a processor is assigned, a 0 means no processor is assigned. The positions of the bit string corresponds to the list $C_0C_1C_2S_{1.1}S_{1.2}S_{2.1}S_{2.2}$.
- Enter the desired percentage of $(*,\tau)$ messages of the total messages generated by GEN, e.g., entering a 40 means that an average of 40% of the generated messages will be of $(*,\tau)$ type.
- Enter the total number of messages to be generated by GEN. The execution terminates when there are no more messages to be processed.

All the CREATEd processes at initialization are passive except the process assigned to GEN. When GEN produces the first message, the execution of the distributed simulation begins.

The following are the variables used for message passing and synchronization:

1. $MESS(*process id*), this is an array of asynchronous variables indexed by the process id. Each element in this array contains either an (x,τ) or $(*,\tau)$ message. The receipt of this message signals the process (either coordinator or simulator) to begin executing the appropriate subrotine. For each process, the following statement

   MYMESS = $MESS(*process id*)

   will force the process to wait if the right hand side is *empty* or to continue executing if the right hand side is *full.*
2. $DONE(*process id*), this is an array of asynchronous variables indexed by the *process id.* An element in this array contains the signal to a coordinator that a subordinate with the index *process id* has completed its execution. If process α is busy computing then $DONE($\alpha$) is full, otherwise it is empty.
3. TL(*process id*), this is an array that contains each process' time of last event, t_L.
4. TN(*process id*), this is an array that contains each process' time to the next event, t_N.

For a coordinator, the statement

MYMESS = $MESS(*process id*)

is used to determine whether a message was sent either by another coordinator or a subordinate.

The statement

$MESS(*process id*) = MYMESS

is used to send a message to a process (a coordinator or a simulator).

The statement

$DONE(*subordinate process id*) = 1

is used to flag the subordinate process to be in busy state. This means that the subordinate is busy computing by setting the asynchronous variable $DONE full.

Then the statement

```
20 IF(EMPTY($DONE(subordinate process id)).EQ.FALSE)GOTO 20
```

causes the coordinator to wait until the subordinate process is finished computing.

For a simulator, the statements

MYMESS = $MESS(*process id*)
$MESS(*coordinator process id*) = MYMESS

are used to receive and send messages respectively.

The statement

FINISH = $DONE(*process id*)

sets the asynchronous variable empty, thus signaling the appropriate coordinator that a subordinate has finished computing.

The computations of the following functions are simulated by holding the process for a randomly selected duration of time:

- δ_{int}, internal transition function.
- δ_{ext}, external transition function.
- ta , time advance function.
- λ , output function.

Thus we have a system of concurrent processes where there is no assumption made on the order of processes finshing their computations of state variables.

4. Experimental Runs and Results

As mentioned in section 3, the implementation of the hierarchical abstract simulator consists of 3 levels with 3 coordinators and 4 simulators. The advantage offered by the hierarchical abstract simulator is the exploitation of the parallelism inherent in the model, i.e., the external events sent by a model component to its influencees can all be processed concurrently. The parallelism is facilitated by the hierarchical model decomposition and such parallelism may thus grow exponentially with the number of levels of a hierarchical DEVS model.

Unfortunately, such gains from parallelism cannot be fully realized. Experimental runs were made and three factors were found to affect the execution time of the implementation on the HEP computer:

(a) Constraints of the hardware architecture (number of processors).
(b) Frequency of synchronization and intercommunication.
(c) Workload (number of messages to be processed).

This section presents the effects of the above factors on the execution time of the implemented hierarchical abstract simulator. With regards to the constraints of the hardware (number of processors), we run the following assignments of processors:

- 3 processors, with each coordinator being assigned a processor and no simulators being assigned a processor.
- 4 processors, each coordinator is assigned a processor and one of the simulators is assigned to a processor.
- 5 processors, each coordinator is assigned a processor and two of the simulators are assigned each with a processor.
- 6 processors, each coordinator is assigned a processor and three of the simulators are assigned each with a processor.
- 7 processors, the full assignment.

When there are not enough processors assigned, the processes share processors which forces them to execute in a sequential manner. Only the last configuration has a one-to-one assignment.

The frequency of synchronization and intercommunication is simulated by varying the percentage of $(*,\tau)$ to (x,τ) messages that is generated by GEN. Also runs were made for processing 500 messages compared to 1000 messages.

Several runs were made of the hierarchical abstract simulator implementation and the results are summarized in Figures 6 and 7. Shown in Figure 6 is the effect on execution time by varying the percentage of $(*,\tau)$ messages. The more $(*,\tau)$ messages in the system, the more intercommunication occurs. This is due to the generation of (y,τ) messages by the simulator when it receives a $(*,\tau)$ message, see Figure 1(b). The (y,τ) message is routed to its destination by the coordinator either within or outside of its enclosure, see Figure 2(b). An increase in the number of $(*,\tau)$ message processed by the hierarchical abstract simulator, the longer is the execution time. A decrease in execution time is noted when there are

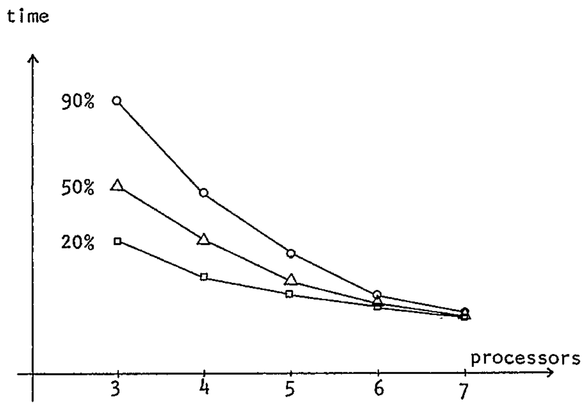

Figure 6: Runs Made for Changing Percent of $(*,\tau)$

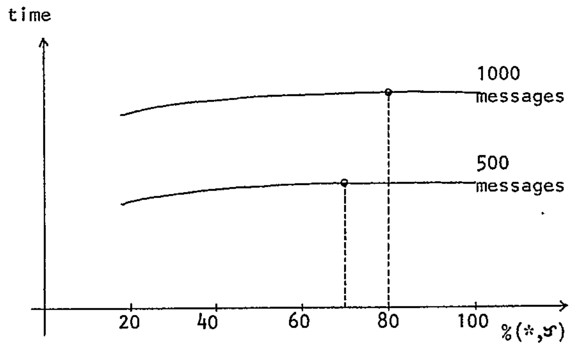

Figure 7: Runs Made for Changing the Number of Messages (using 7 processors)

more processors assigned to the hierarchical abstract simulator but this decrease is not so significant from 6 to 7 processors. Because of the under utilization of some of the processors, the gain by using one more processor (from 6 to 7) is not fully realized.

To get some insight on the effect of increasing the number of messages to be processed, runs were made and the results are shown in Figure 7. This result was obtained by using the full assignment of processors, which is 7. As expected, there is an increase of execution time when the hierarchical abstract simulator is processing more messages. But it was also observed that at 500 messages, the execution time did not increase beyond 80% (*,t) and at 1000 messages, the peak is reached at around 70% (*,t). This shows a saturation point where the increase of overhead due to intercommunication did not affect the execution time. This is due to the parallelism inherent in the hierarchical abstract simulator, the increase of intercommunication is absorbed by the concurrent execution of the processors.

Runs were also made to determine the effect of the following routing characteristics of messages:

- Having more (x,τ) messages routed to both subordinates of a coordinator.
- Having more (y,τ) messages routed to a subordinate and to the next higher level coordinator.

The first characteristic simulates the occurrence of having more simulators affected by an external message, (x,τ). This results in more concurrency in the execution of simulation. The second characteristic simulates the sending of output messages to the most remote simulator. This occurs when the (y,τ) message has to be sent by a coordinator to the next higher level coordinator. The results did not show any significant difference of execution times for both characteristics. This is due to the fact that the coordinator has no delays in doing the following activities:

- table look-up, to determine the destination of the message via the interface tables.
- determining the minimum t_N, the function MINTN was performed in 0 processing time.

But significant difference in execution times were observed when using different assignments of processors. The full assignment sometimes shows half the execution time compared to the execution time for 3 processors.

5. Conclusion

This paper has shown an alternative implementation of the mapping of the hierarchical abstract simulator to a hardware/software architecture. The HEP computer with its MIMD architecture and the support of a high level parallel language, DENELCOR's FORTRAN 77, the combination produces a very viable implementation for distributed simulation. Although there is a great need for more disgnostics and debugging tools to trace and debug parallel programs.

Performance in terms of execution times were measured on different runs of the implementation. It was observed that the number of processors, frequency of synchronization and intercommunication, and number of messages affect the execution time.

The following gives a summary of the results obtained:

- that an assignment of processors close to the full assignment gives almost the same execution time as a full assignment.
- that the execution time increases when there are more $(*,\tau)$ messages than (x,τ) messages to be processed.
- that for an assignment of processors, there is a saturation point where increasing the $(*,\tau)$ messages did not increase the execution time.
- that the execution time increases when there are more messages, $(*,\tau)$ and (x,τ), to be processed.

Some research have been done on performance of distributed simulation. Livny [8], discusses a measurement called the Optimal Execution Time which gives a relationship between the inherent parallelism and the number of concurrent simulators. Davidson and Reynolds [4] found out in their experiments of using 3 microcomputers for distributed simulation that the degree of communication degrades the performance of the simulators. The processes communicate with each other at the end of a certain time interval. If this time interval is less than 10 units of time, then degradation of performance was observed. In Baik and Zeigler [1], a methodology is presented for the performance evaluation of hierarchical distributed simulators. Their methodology measures the minimum average run time per task and the maximum throughput per unit of hardware complexity.

The difference of the above research from this work is that, an implementation of the distributed simulator is done on an actual multiprocessor architecture and that actual CPU real-time are measured. The results of this work also shows a saturation point for the hierarchical abstract simulator and that the full assignment of processors does not always produce the optimal performance.

Future work on the hierarchical abstract simulator implementation on the HEP computer will consists of the following:

(a) Inclusion of the (o,τ) message type and introducing delays in each coordinator for table look-up and MINTN activities.

(b) Expanding the current implementation to a general tree structure.

(c) Running a real-time simulation of a distributed computer system.

Acknowledgement

The author would like to thank Ann Hayes, Computing Research and Application Group, Los Alamos National Laboratory, for allowing us computing access to the HEP computer on site of the Los Alamos National Laboratory. The work would not have been possible also without the support of DENELCOR's consultants on site, Olaf Lubeck and Dale Carstensen. They have been very patient in answering our questions.

References

[1] Baik, D-K and Zeigler, B.P., "Performance Evaluation of Hierarchical Simulators", In Proc. of 1985 Winter Simulation Conference, San Francisco, CA, Dec 1985.

[2] Concepcion, A.I., "Distributed Simulation on Multiprocessors: Specification, Design and Architecture", Ph. D. Dissertation, Tech. Rep. CSC85-001, Dept. of Computer Science, Wayne State University, Jan 1985.

[3] Concepcion, A.I., "Mapping Distributed Simulators Onto the Hierarchical Multi-Bus Multiprocessor Architecture", In Proc. of the 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 8-13.

[4] Davidson, D.L. and Reynolds, P.F., "Performance Analysis of a Distributed Simulation Algorithm Based on Active Logical Processes", In Proc. of 1983 Winter Simulation Conference, Arlington, VA, Dec 1983, pp. 266-268.

[5] DENELCOR, "FORTRAN 77 Reference Manual, Release 1.0", Publication No. 9008020-000, DENELCOR INC., 17000 E. Ohio Place, Aurora, Colorado, Jun 1984.

[6] Gajski, D.D. and Peir, J-K, "Essential Issues in Multiprocessor Systems", Computer, Vol. 18, No. 6, Jun 1985, pp. 9-27.

[7] Hwang, K. and Briggs, F.A., "Computer Architecture and Parallel Processing", McGraw Hill Book Company, New York, 1984.

[8] Livny, M., "A Study of Parallelism in Distributed Simulation", In Proc. of 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 94-98.

[9] Zeigler, B.P., "Multifacetted Modelling and Discrete Event Simulation", Academic Press, London, 1984.

[10] Zeigler, B.P., "Discrete Event Formalism for Specification of Hierarchical Models", In Proc. of the 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 3-7.

ARTURO I CONCEPCION received the B.S. degree in Mechanical Engineering from the University of Santo Tomas, Manila, Philippines, in 1969, the M.S. degree in Computer Science from Washington State University, in Pullman, in 1981, and the Ph. D. degree in Computer Science from Wayne State University, Detroit, Michigan, in 1984. Since 1982, he has been involved with research, funded by the National Science Foundation, on the theory, design and implementation of distributed simulation. He is currently an Assistant Professor in the Department of Computer Science, Michigan State University. He is now involved in a research group which studies the distributed control, efficiency and reliability of distributed computer systems. His principal interests are in distributed operating systems, networks, distributed databases, and modelling and simulation. He is a member of the ACM, IEEE-Computer Society and Sigma Xi.

Department of Computer Science
Michigan State University
E. Lansing, MI 48824
(517) 355-2359