High Performance Cluster Computing:

Programming and Applications, Vol. 2

Edited Book

by

Rajkumar Buyya
(rajkumar@dgs.monash.edu.au)

School of Computing Science and Software Engineering
Monash University
Melbourne, Australia

Chapter 19

Hardware System Simulation

D. K. SHARMAN[†] AND D. M. W. POWERS[‡]

[†]Motorola Australian Software Centre
Adelaide, South Australia

[‡]Department of Computer Science,
Flinders University of South Australia
Adelaide, South Australia

Email: *dsharman@asc.corp.mot.com, powers@ist.flinders.edu.au*

19.1 Introduction

Electronic devices are becoming increasingly larger, more integrated, powerful and difficult to design, so that simulation must play an ever-increasing and important role. With these increasingly large systems, simulation times are actually getting relatively longer with regard to the target systems being simulated [3]. The aim of this project was to ascertain the benefits of using a local area network as host for a parallelized simulator of coarser grain than has been implemented previously. The artifacts simulated in this project are parallel machines themselves. The choice of simulating parallel hardware comes from three factors, with the first two being able to effect the simulator's load. These factors are:

1. Each Processing Element (PE) can be varied in size/complexity,
2. The total number of PEs can be manipulated,
3. This is a real-world problem faced by engineers designing parallel hardware.

The design was primarily dictated by efficiency considerations. There are several ways in which network traffic can be minimized. The first is to use broadcast rather than point-to-point communication as much as possible, and we saved further on overheads by designing our protocol using the User Datagram Protocol (UDP) directly.

19.2 NEPSi

The Network Enabled Parallel Simulator (NEPSi) is the test bed for this project. It consists of a VHDL (Very High Speed Hardware Integrated Circuits Hardware Description Language) simulator augmented with communication primitives so that it can be distributed across a network of workstations.

19.2.1 ASIMUT

NEPSi is based on ASIMUT, a VHDL simulator which is part of the ALLIANCE CAD package[2]. The alliance CAD package is primarily intended as a teaching tool and as such supports only a subset of VHDL. However, it has been used in serious applications by the team at MASI and several large projects have been reported using the package. Most industry standard packages are priced in the tens of thousands of dollars plus annual licenses and high end workstations are required to do anything of a serious nature on them. ASIMUT, on the other hand, is happy to work with virtual memory, will run on a low end SPARC (an ELC at that) and comes in at a cost we can afford: nothing.

Concurrent Distributed Processing

NEPSi is a concurrent distributed simulator. By this we mean that the processes that comprise the simulator are not only spread across distributed machines, but that multiple copies of the simulator may also be run on each machine. Thus we not only have concurrency provided by the distribution process, but we also have concurrency provided through time sharing on each machine. This allows for flexibility in matching computation time and communication latency.

19.2.2 NEPSi as a Client Server Architecture

Data Distribution

UDP provides for broadcast transmission of a packet onto a subnet. Two network configurations were examined. The first setup is where packets would be broadcast

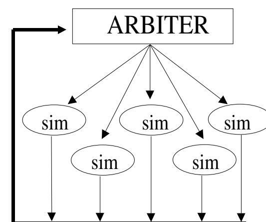

Figure 19.1 Communication flow in NEPSi.

from the arbiter and all return packets would be sent back to the arbiter for reassembly and broadcast as shown in Figure 19.1. The alternative setup was for each simulator to broadcast its results and reassemble the resultant global broadcasts once all simulators had transmitted their data.

Early experimental results showed that the first setup was more appropriate. It reduced the cost of, or facilitated, a number of operations better than the alternative. Specifically, the arbiter needs to be the hub for the following reasons:

1. The arbiter needs to synchronize the start up of the client simulators. For process migration to be implemented, a centralized register of processes/machines is required.

2. The arbiter needs to compile the separate signals into one message to be broadcast. If all clients assemble their own packets, then network traffic will increase dramatically. If a central arbiter with 5 sub-nets is used then it needs to send only 5 broadcast messages (one to each sub-net); if a more distributed system is used, then each node will have to transmit 5 packets (one to each sub-net). Thus if we have 10 nodes in the network, we would have the following number of packets:

    - central arbiter 5 broadcast messages + 1 from each node =15
    - distributed control 5 messages from each node = 50

    Of course, if a single sub-net is used, then costs are about equal (10 pkts without arbiter, 11 pkts with an arbiter). Note, too, that a single UDP packet is of limited size, so multiple packets will be required once a large number of signals need to be transmitted over the network. Since the number of machines on each subnet is limited and it is necessary to use more than one subnet, we have adopted the centralized arbiter model.

3. Once large simulations are run, we need to detect bus conflicts. There are two ways in which this can be achieved. The arbiter can do the signal detection or every node can do its own conflict detection through merging of bussed signals from other nodes. The second method generates more redundant computation, which can be done in parallel with other computation in the central arbiter method.

For these reasons, the arbiter is the central point for all communication. This allows for broadcast messages to be used rather than point to point as is used in other message-passing schemes such as Parallel Virtual machine(PVM)[5], [6]. This increases parallelism dramatically, which is of utmost importance, although it is predicated on the total number of signals (G) being of the same order as the number of signals per node (L), the network overheads(H) & the number of sub-nets(S). The costs are:

central arbiter: S*(H+G)+N*(H+L)

distributed control: S*N*(H+L)

19.2.3 Overview of the NEPSi Network

NEPSi is comprised of many parts, each one relying on the others, and in total all working together to perform the simulation. The next sections deal with the implementation of each part of the network. We begin by looking at the simulation engine, NEPSi ASIMUT, then we look at the arbiter, the daemons, and finally the VHDL parser used to split the description into leaf cells.

NEPSi Client - ASIMUT

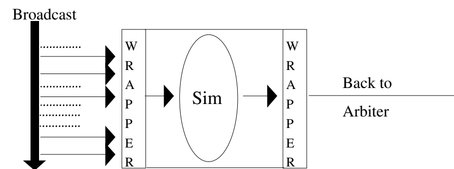

Figure 19.2 NEPSi client is an augmented ASIMUT simulator.

As can be seen in Figure 19.2 a simple function logically wraps around the ASIMUT simulator. This wrapper functionally is actually inside ASIMUT, as the simulator, as it stood, had no interactive input or Programming Language Interface(PLI). The wrapper function grabs its assigned signals from the broadcast packet and sends its resultant signals back to the controller.

The Arbiter

The arbiter is the heart of the NEPSi infrastructure; it is responsible for arbitrating all communication between the simulators and daemons. Figure 19.3 displays all the communication paths within the NEPSi network. As can be seen, outbound communication from the arbiter takes two forms: arbiter to daemon and arbiter to simulator. The arbiter is responsible for initiating the communications network. This is accomplished by issuing a ping to the daemons, arc one. On receiving the ping command the daemons simply respond with their daemon ID-number, arc two. The arbiter uses this information to establish which daemons are available and uses the response times as a rough indicator of the load on the replying machines. Given that we have n modules to be simulated, the arbiter will select the top n machines (in response time) as the hosts for the simulators. If there are more modules to be simulated than there are available machines, the arbiter will increase the number of simulators per machine to accommodate the required amount. The required number of simulators is then sent to the daemon in arc three. Arcs three and four represent the daemon interacting with the OS to invoke the first simulator. The

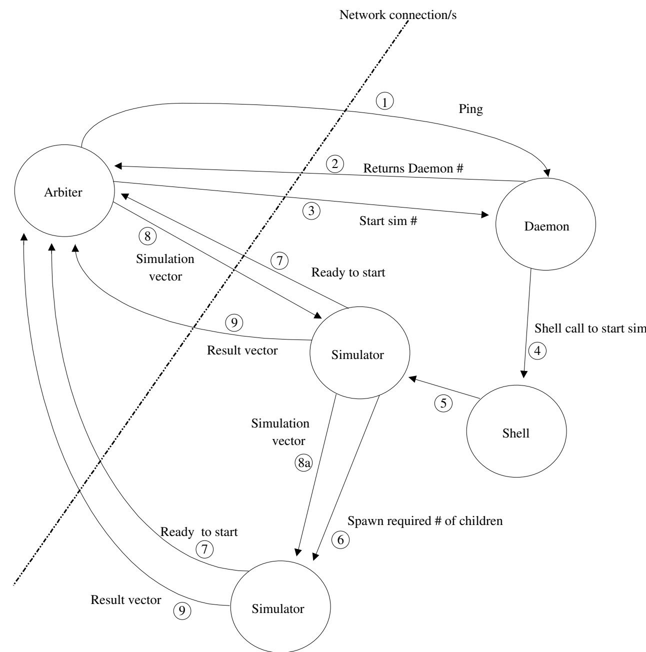

Figure 19.3 Communication flow diagram for NEPSi network.

first simulator started on each machine is then responsible for forking the desired number of children, shown here as arc six. The reasons for starting additional simulators on each machine using the simulator itself are twofold:

- We get a performance gain by the simulator cloning itself instead of the children having to parse and set up all the data structures: We get a bonus from the memory manager.
- All the communication is sent to one port on all the machines, and only one process can be bound to a port, so the parent has to open pipes to its children before it forks.

The fork is represented by arc 6 in Figure 19.3. Once all the internal data structures are set up and any required children have been created, each simulation process then informs the arbiter that it is ready to begin simulation, arc 7. The arbiter then transmits the simulation vector to the parent simulators on each machine, arc 8, and the parent gives this to the child, arc 8a. All simulators then send their result back to the arbiter, arc 9. The arbiter then assembles the signals for retransmission and sends them out again on arc 8. This process continues until the last test vector has been simulated. Then the arbiter sends a special test vector that informs all

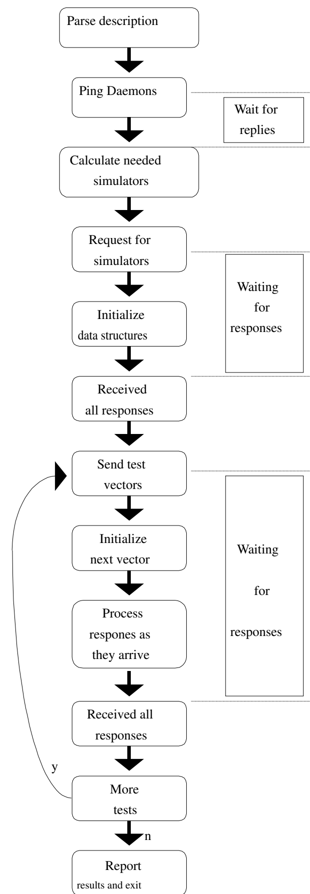

Figure 19.4 Operations performed in a typical simulation.

the simulators that they should exit. Daemons remain alive, sleeping, waiting to start up the simulators again.

These test vectors are represented compactly so that each byte reflects 4 signals. This allows us to have a total of approximately 4500 signals in a packet, depending upon the number of PEs in the design and the number of inter-PE signals. The simulation cycle has been designed in such a way that communication and calculation could be interleaved as much as possible. As can be seen in Figure 19.4, there is a delay slot after each message is passed to the simulators where the arbiter is waiting for a response. As much of the calculations needed to be performed has been rolled into these slots. Figure 19.4 illustrates the operations carried out during

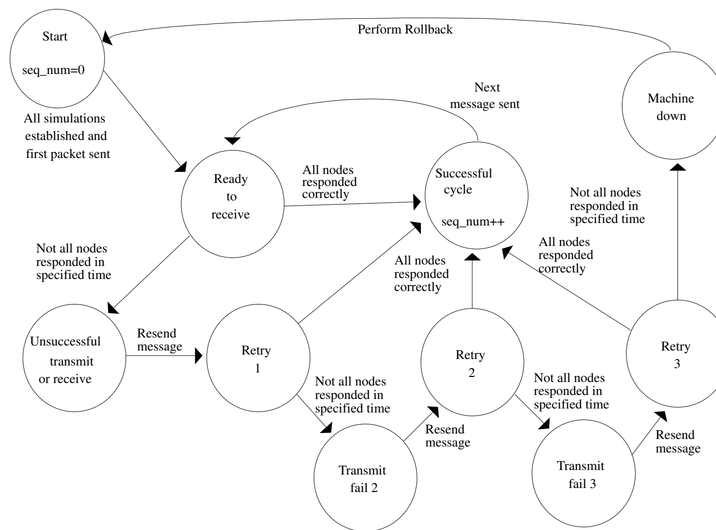

Figure 19.5 State transition diagram for arbiter communication model.

the execution of a simulation session. A lot of the initialization of the data has been folded into the slots where the arbiter is idle waiting for the responses from the simulators. Most of the test vector is constructed while waiting for the results of the present cycle. This reduces the amount of processing time the arbiter needs while the simulators are idle and minimizes the sequential bottleneck of the simulation algorithm.

NEPSi Daemons

The NEPSi daemon is simply a small program that acts as a remote shell. It has a specific port to which it listens and to which requests are made.

19.2.4 Communication Model

Communication in NEPSi is based directly on UDP, an unreliable communication medium in which packets are not guaranteed to arrive and if they do, they may arrive out of order. This sounds worse than it actually is. Packet loss is not as bad as one might expect, and this is a strength as much as a weakness of UDP. We have defined a communications model for NEPSi which exploits this.

Answers as Acknowledgment

As UDP is an unacknowledged protocol, the sender receives no notification of whether the packet arrived at its destination or not. This increases the speed

obtainable from UDP as packets can be sent without having to wait for acknowledgment packets that would double the network overheads. So to make sure that our packets have arrived, we use the result as an acknowledgment. To facilitate this, it is necessary to number all packets with a sequence number. We send only one packet to each subnet in the system. Therefore there will be varying number of replys to each packet, as different subnets have different numbers of computers, and each computer can have more than one simulation process on it. Figure 19.5 shows the state transition diagram for the arbiter's communication scheme. As can be seen, the arbiter simply transmits its data and waits for a reply. If all the nodes reply correctly with the same sequence number, then it proceeds to simulate another vector. All packets received with the wrong sequence number are ignored. If the time limit is reached and not all simulators have replied correctly, then we enter the missed transmission state. A packet could have been lost on the simulator side, or we may have lost a packet on the arbiter side. When this occurs, we simply resend the last message. If the simulator receives this second transmission, it has either seen it before and simulated it, or it didn't receive it in the first place. If it has seen it before, it simply retransmits its last result with the previous sequence number. If it hasn't seen it before, then it simulates it and sends the response. This is why the arbiter can receive packets with the wrong sequence number. The reason why we have so many retries is that we need to know if a machine has gone down (or is very busy). If we retransmit four times unsuccessfully, then we assume that a machine has gone down. We then enter damage control mode where we kill all the simulators and do a rollback start. For brevity and clarity, Figure 19.5 has a large number of states involved in the rollback process represented as one transition; similarly, the startup process has been omitted for clarity. The state transitions for the simulators side of the communications scheme is very simple. It waits forever until it gets a message and replies to all given messages, with the last result it generated.

Checkpointing & Rollbacks

To make the system more secure with regard to long execution times, it has been necessary to implement a means by which checkpointing can be implemented. At a predetermined time interval, measured in simulation cycles simulated, the arbiter transmits a special packet that instructs the simulators to save their internal states to a file. Each simulator then saves its state to a file with a unique name based on its node number. These files are stored in a centralized account which relies upon the network file system (NFS) to save the files on the one machine. If a rollback is deemed necessary, the arbiter then performs the following tasks:

1. instructs all simulators to terminate;
2. ensures that all simulators have terminated;
3. pings the daemons;
4. recalculates what simulators are required on which machines;

5. asks the daemons to do a roll_start;

6. begins processing again from rollback point.

The rollback point is the simulation test vector immediately following the vector which preceded the checkpoint. The simulators are not guaranteed to be simulating the same node number after a restart, but this has no effect on simulators as all the checkpoint files are stored in a central point from which they all have NFS access. This scheme works well with two notable exceptions:

- If the machine with the host account in which the checkpoints are stored goes down, then the simulation cannot continue.

- All machines in the NEPSi network must reside within a single NFS domain.

19.2.5 VHDL Models Used

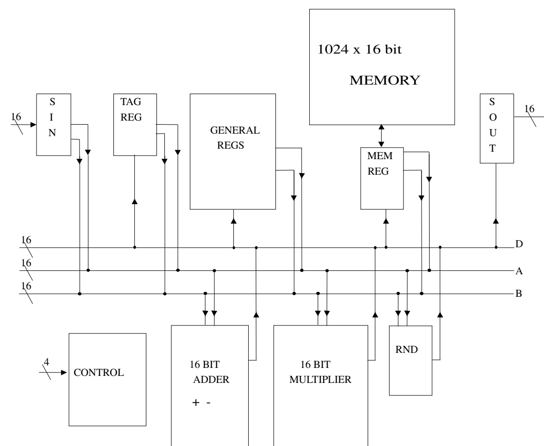

Figure 19.6 TNP processor.

The hardware we chose to simulate was the Toroidal Neural Processor(TNP) [4] which was designed to be used in neural network experiments, and it is theoretically infinitely scaleable. Large extensive test files were available for the TNP and experienced users (e.g., the author) of the TNP were available.

TNP Processor

Each of the modules in Figure 19.6 is written as a behavioural description. Interconnections between these modules is described in the top level structural description. The modules range in size from a single register to a 1024 sixteen bit memory array.

The varying size of these modules allows us to examine at what point the communication versus computation tradeoff favors switching to a parallel implementation.

19.2.6 Tests for TNP

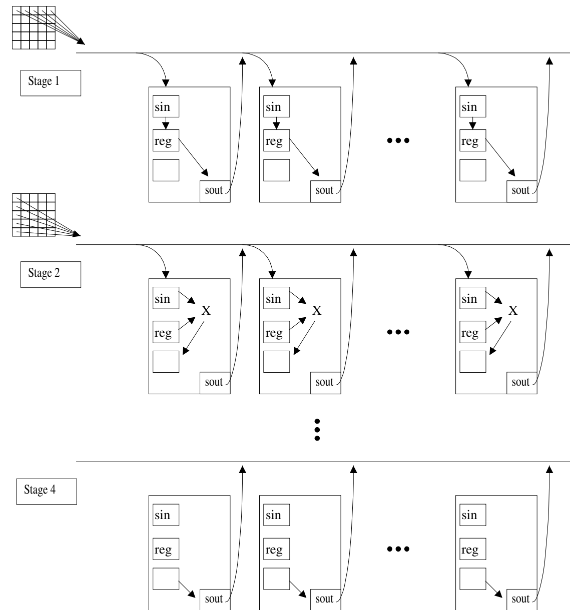

Figure 19.7 Stages of matrics multiplication.

As a small example of the kind of task the TNP was designed for, a vector dot product was calculated for the majority of our tests. Figure 19.7 outlines some of the stages involved in the multiplication. A row of the first matrix is first loaded into the array (stage 1 of diagram), then a column of the second matrix is passed through the array with a partial sum building up in each element as the column is passed through (stage 2). Once this is completed, all the products are passed out with the last element summing the products as they pass and then this is passed out (stage 4). This process is then be repeated for all the remaining rows and columns, but to aid test turn around times we do only the first row and column.

TNP Assembly Tests

Test programs were written for TNP arrays varying from 1 to 64 nodes. These programs were then used as the simulation vectors while the experimental variables were manipulated, which are as follows

- Number of machines used

- Number of processes on each machine
- Number of PEs per simulator
- Distance between simulators (all simulations on campus, or some at a remote campus)

These four variables and the combinations thereof provided for a large array of experiments.

19.2.7 Initial Tests

The following results were obtained from running a simple test program/pattern file through a simulated TNP array. The array consisted of 10 PEs in a linear array. The output from each PE is fed into the input of the next. Input to the array is from within the pattern file and the output is obtained from the 10th PE. The NEPSi network consisted of 10 machines on 4 subnets. Each machine was minimally loaded (0.0-0.05) with nova and baldrick being the exception. These had a load of 1 to 1.1. The graph in Figure 19.8 details the simulation times for a sequential (standard ASIMUT) simulation of the PE array and the time taken for NEPSi to do the same simulation as averaged over 10 runs. As can be seen, simulation times varied extensively with boomer being the fastest and karl being the slowest. NEPSi averaged at least half the execution times of most machines, and was up to 5 times faster than the slowest machine. To compensate for load

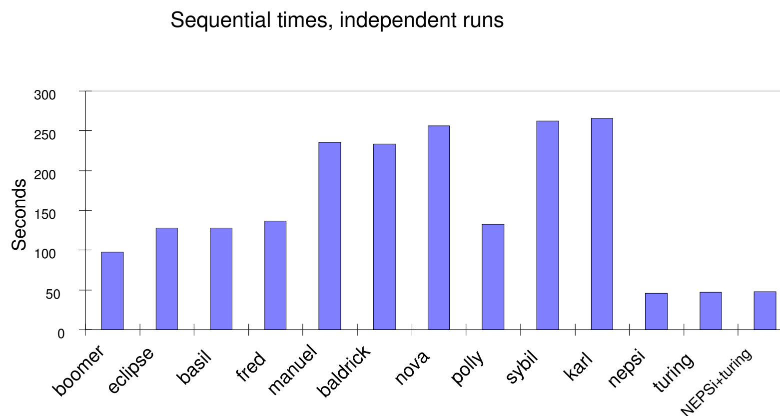

Figure 19.8 All machines compared to NEPSi.

differences, a further 10 iterations were run on each machine with NEPSi running concurrently. This increased the load on each machine, but it was intended to be a crude way of maintaining some load consistency between the test runs performed

on NEPSi and the sequential version. As can be seen in Figure 19.9, the concurrent runs were slowed down with a consistent amount. From this we can assume that the sequential times vary directly with the load on each machine and the amount of physical resources available to them.

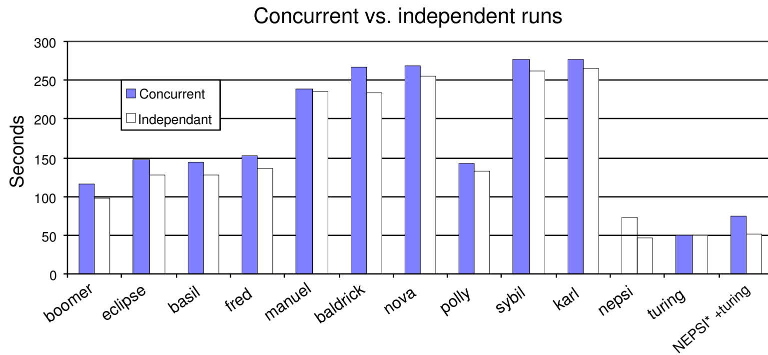

Figure 19.9 Concurrent runs compared to independent runs.

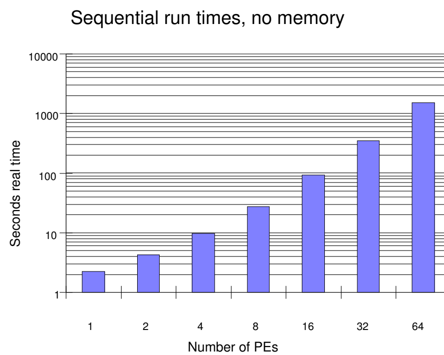

Figure 19.10 Average cycle times on Turing.

Since the NEPSi version is done in lock step, its simulation time will directly depend on the speed of the slowest machine in the network. In both Figures 19.8 and 19.9 we see that the best machine (NEPSi) is around 5 times faster than the slowest machine. The machines are all either Sparcs or SuperSparcs, but already the factor of two difference in power has meant that the faster machines are underutilized. The asterisked machines are presented for comparison and were not part of the original experiment. Karl was then supplanted by turing, an UltraSparc, after the initial experiments.

19.2.8 Varying Numbers of PEs and Simulators

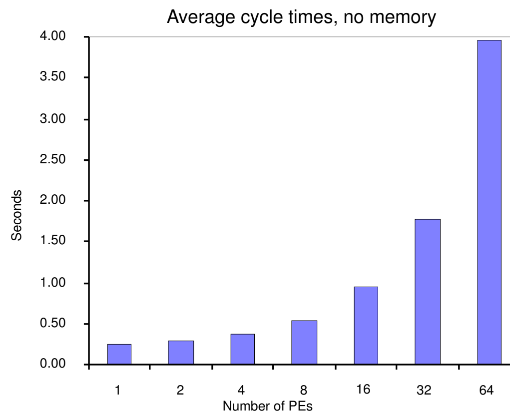

Figure 19.11 Average simulation times on Turing.

Once the initial tests had been conducted and had proved the applicability of our concept, we then moved on to slightly more complex testing. The following figures show the results for sequential runs on Turing using the simulation vector for a ring adder test as we varied the number of PEs simulated on the machine. Figure 19.10 shows the average simulation times for turing and Figure 19.11 shows its average cycle time. It should be noted that the simulation times are given on a

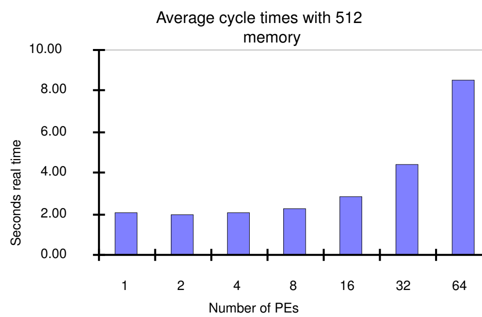

Figure 19.12 Cycle times on Turing with 512 byte memory.

logarithmic scale. The simulations were conducted using a reduced version of the TNP PE, which had only one memory location instead of 512 or 1024. Using the trivial memory aided in implementation and development as simulator start times

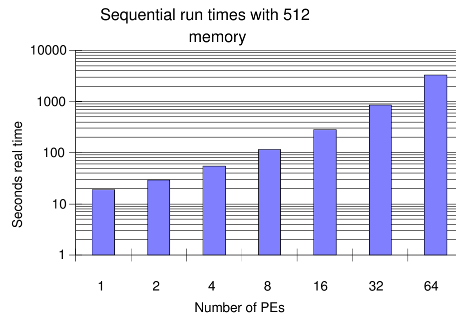

Figure 19.13 Run times on Turing with 512 byte memory.

were significantly reduced, as can be seen in Figure 19.14. The following Figures 19.12 and 19.13 display the simulation and cycle times for a TNP array that uses 512 bytes of memory in each PE. At first one would assume that simulating memory would be rather fast, as we recall the simulation is an event based simulator and as such must recalculate the internal state of all Binary Decision Diagrams (BDD) that have an input change state. But as all the memory is clocked, every memory location must be recomputed. Bryant [1] points out that simulating memories with BDDs is not efficient due to the high fanout that occurs as the BDD tree grows. Furthermore, we could see from the memory usage of the simulator itself that the BDD representation of the memory is inefficient. The memory usage of the simulator became extreme when simulating 64 PEs with a 1024 byte memory, with the executable growing to over 218 meg at runtime, with the result that at some point the process started to thrash and entered into a continuous I/O wait state. This happened on all machines, but just when they started thrashing depended on how much actual memory and how much swap space they had. Given that these processes also fork numerous children, the machines are liable to crash under these conditions and running tests of this size was considered antisocial in a shared resource environment. Therefore, the experiments with half the specified memory were undertaken instead, as presented above.

19.2.9 Matrix Multiplication on TNP

Once we had finished implementing and testing the basic simulator, we proceeded to extend it. The rollback/checkpoint system was implemented, packet compression, auto-daemon selection and top level preparsing/decomposition were added and the test file parser was developed. All these things either helped make the simulator easier to use or made it more stable. These additions added only a small amount to the startup times for the ring adder; it added around 4 seconds, which was largely due to the waits for the daemon startup/selection. With all these facilities in place we could then perform the tertiary tests. The results for these tests are

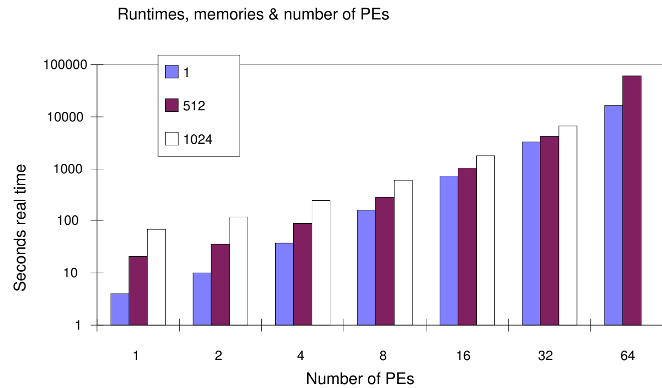

Figure 19.14 Comparison of memory sizes and number of PEs.

presented in Figures 19.15 and 19.16. Again we are using a logarithmic scale and the machines displayed are meant to represent the spectrum of the machines used in the NEPSi network. It should be noted that we used only 16 machines at a maximum, therefore the 32 and 64 PE simulations required that each PE simulate 2 and 4 PEs, respectively.

As can be seen, NEPSi is actually slower than turing for the simulations where there are less than 8 PEs being simulated. The slowest machines like manuel are only half the speed of NEPSi for a single PE simulation. This is due to the start-up time for NEPSi, which is comparatively larger for smaller simulations. As can be seen in Figure 19.15, this deficit is quickly made up in the larger simulations. Figure 19.16 details the cycle times for the Matrix multiplication as given in Figure 19.15. This is a very interesting graph and shows distinctly where the break-even points are for NEPSi. As can be seen, the cycle time for NEPSi is higher than for turing when the TNP arrays consist of less than 7 PEs. Comparing NEPSi directly to turing is not a good metric. NEPSi is constrained by the slowest machine in the network. In the case above, it is an ELC. What should become apparent from the graph is that we can obtain the performance of nearly twice that of an ULTRA Sparc from servers that are now very well dated. After we have an array larger than 7 Pes, NEPSi becomes viable. What is also of interest is the slopes of the lines. The turing line is nearly completely straight; this proves simulation speed is directly related to the simulation size. However, if we observe the line for boomer, we can see that the line increases in slope as we increase the number of PEs from 16 to 32. This shows the point where we also start to run out of compute resources (physical memory). Therefore, the time a simulation will take is directly related to the number of PEs being simulated and the amount of free resources available for this computation. Interestingly, the line for NEPSi is relatively flat until we reach the 8 PE mark. This may be due to three factors: Firstly, the arbiter selects the

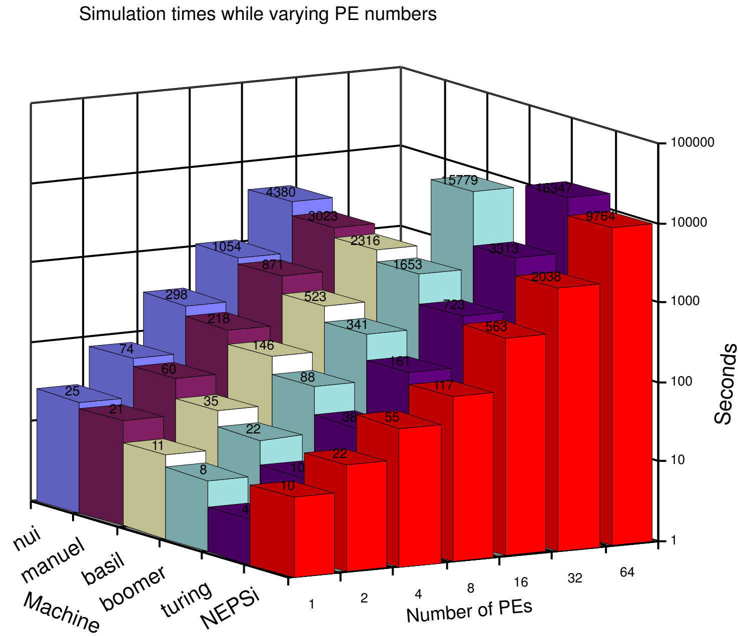

Figure 19.15 NEPSi and sequential version performing TNP matrix multiplication.

fastest machines first (network response plus some parsing in the daemon); as we go past eight machines, we start to use the slower machines and thus we increase the time required to simulate some of the PEs. Examining Figure 19.9 reveals that there are basically two levels of machine performance available in the Flinders computer science network. There are eight machines of relatively fast performance (this includes turing and boomer, which are faster again), then there is the second tier to which a lot of the student servers belong and some ELCs. These machine are nearly half the speed of the first tier machines. Once we go over the 8 machine mark we start using the slower machines. This is clearly visible in Figure 19.16; it is the first kink in the graph on the NEPSi line. Secondly, as we progress beyond 16 PEs we need to start doubling the number of PEs being simulated on a machine. Again we are doubling the simulation load on the machines so that the graph continues on the same slope as it did from the 8 PE mark. This increase in slope is not due to network overheads but is due to a resource shortage. If more machines were available then the graph may flatten out again. Thirdly, there is an effect upon the machine called manuel, which is the NFS server for the directory in which the executables and the test files are kept. This is the directory to which all the checkpoint files are written. Therefore, when we are simulating large numbers of PEs we get some NFS errors as the 64 simulators try to read and write to the same directory/file.

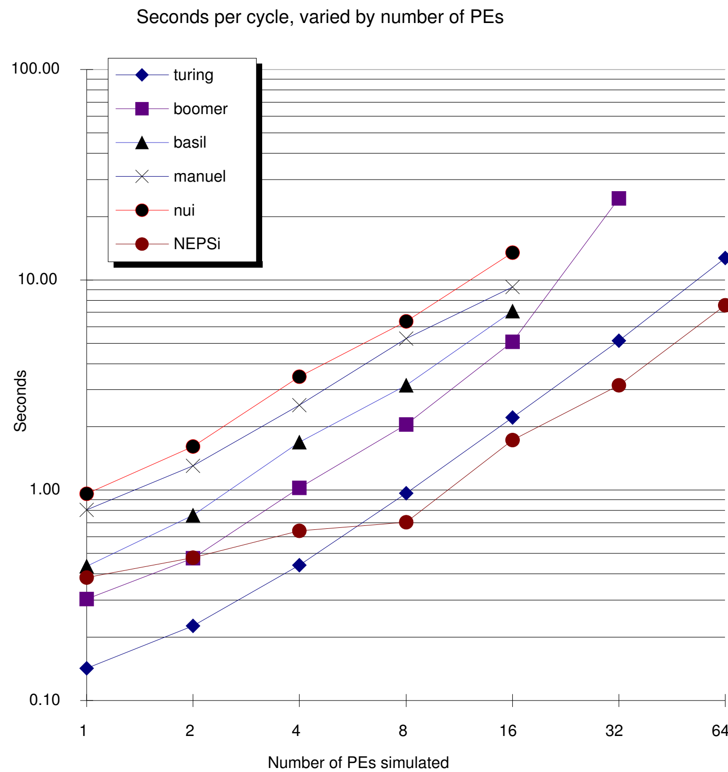

Figure 19.16 Machine comparrison with varying number of PEs.

Furthermore, as the arbiter is on a fairly crowded subnet, when all the responses start coming back from the simulators we may be flooding the network, although there appears not to be any significant increase in network collisions. This can be observed with the number of retries that are needed for a simulation; these increase with the number of PEs being simulated. However, many factors come into play here, including network load, the load on the arbiter, the machine it is running on and NFS server contention, etc. It is very hard to determine the exact cause of network behaviour as the experiments are conducted on a working academic network which is subject to all the trials and tribulations that students can put it through.

19.2.10 Variable Machines, Processes and PEs

To ascertain what effect simulation load has on throughput, a further test was carried out: We varied the number of PEs in a simulator over a specified range of

array lengths. Then we conducted the same set of tests, but we varied the number of simulators on each machine rather than the number of PEs. Figure 19.17 shows the effect of having more than one simulator on a machine (the array numbers were kept small so as to allow homogeneous servers to be used). The single PE single simulator had the best average cycle time. As can be seen, the single process usually returns the fastest average cycle times. However, you will notice that the average for the 2 simulators per machine is actually faster for the 8 PE simulation than for the one simulator version. This may be caused by a quicker start time or just variation in the network traffic. As can be seen, 4 simulators per machine reduce the performance by about half. This directly shows the effect of the increase in average cycle times for four simulators per machine, as displayed in Figure 19.16. This does not support the increase in average cycle time from one to two processes that is also evident in the same figure. Most likely, network variation is the cause; this point is taken up in Section 19.2.14. To find the relationship between communication and computation,

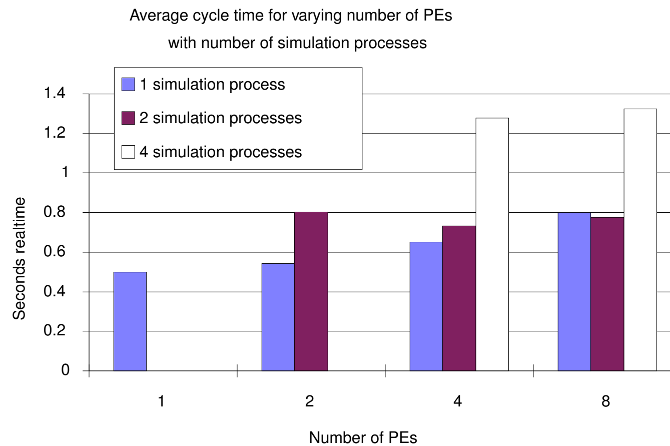

Figure 19.17 Varying number of PEs and processes.

a further experiment was undertaken where we measured the CPU time taken for a simulator to perform a simulation, and we compared this with the amount of CPU time used by the concurrent version. We used the machines that were as evenly matched as possible; thus, we had only eight machines available. As can be seen in Figure 19.18, we have been fairly successful in limiting the communication time compared to calculation. The distance between the NEPSi line and the bar is the communication time and system overhead for NEPSi, and the distance between the other line and the bar is the system overhead for the sequential version. A speedup factor of about 6 is evident again, and furthermore, we can see that the communications overhead is about a third of the time. The average node CPU

usage is also given, and this indicates how much processing has been interleaved by the distribution process.

19.2.11 Rollback

The rollback mechanism was implemented through each simulator producing a save file at a given point in the simulation. The saving of a checkpoint takes approximately half a second, apparently independent of the number of processes involved (the actual time is most likely hidden in NFS delays and disk buffers). The rollback mechanism can take anywhere between 6 and 15 seconds. The arbiter must make

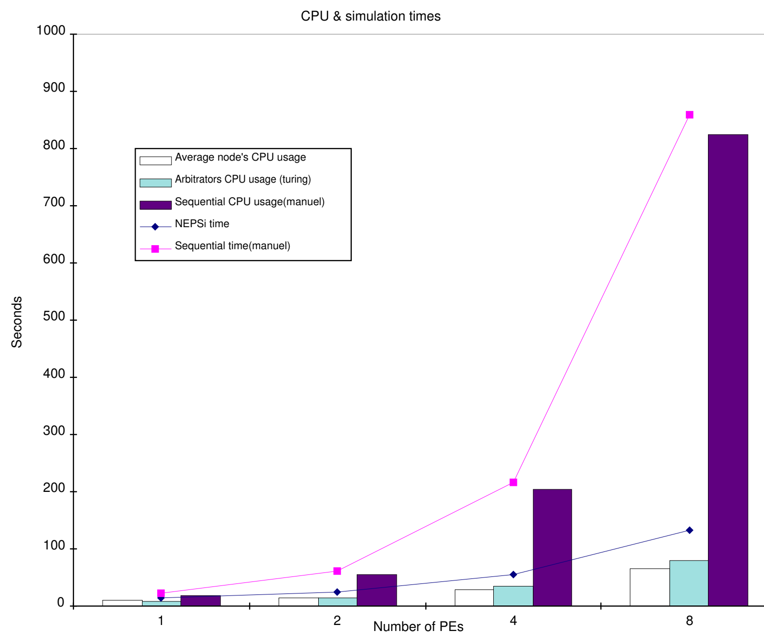

Figure 19.18 CPU usage and simulation time versus number of PEs.

sure that all the simulators are dead before it can ask the daemon to invoke a new set of simulations. When this occurs, primitive process migration takes place and the arbiter will select the best machines that it requires for the simulation. As can be imagined, the rollback process is quite costly and thus we need to make sure that if we are going to do one, we lose the least amount of time.

19.2.12 Multi Campus Simulation

The use of servers at Adelaide University served an interesting purpose. Adelaide University is connected to Flinders University via a 31 megabit link. The Flinders

computer science network is connected to this backbone via 2 other network hops. Using the network utility ping reveals that we typically have a 5-8 millisecond round trip. The network here at Flinders is an unruly beast and it does get choked (especially when the AI class is running a robot soccer simulation); on average, the network delay is anywhere between 1 and 4 ms. The experiments run using the Adelaide machine showed no statistically significant slowdown in the performance of NEPSi. Some large delays occurred, but these usually originate here at Flinders so the simulation would have been effected whether the simulation was run here or not. One disadvantage of using machines at Adelaide is that if we are to implement checkpoints and thus migration, we need to keep a consistent copy of all the checkpoints at both locations without NFS.

19.2.13 Effect of Migration / Load Balancing

The migration code was added to NEPSi in an attempt to counteract the effects of increased network traffic that is sustained for medium lengths of time (such as student machines being used for a class). When a machine reaches a certain level of activity, the arbiter may decide to migrate the simulation to another machine which is not as busy. This is implemented through the rollback-checkpointing mechanism. The arbiter will ask a parent process on a machine to fork another child. This is an extremely quick process. The simulation node to be migrated performs its

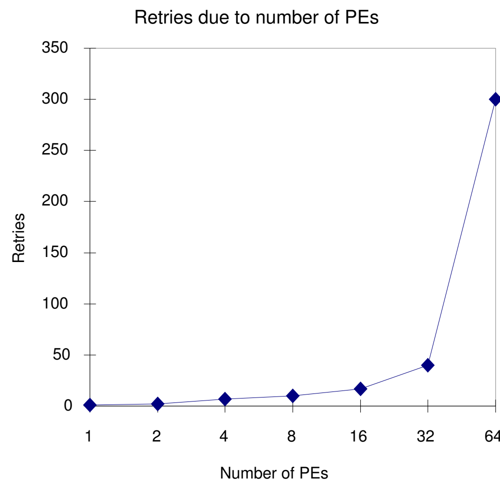

Figure 19.19 Retries versus number of PEs.

checkpoint at the normal designated point. Then on the next cycle the arbiter issues a special packet that informs the appropriate parent to spawn a new child using the checkpoint data from the to-be-spawned process. Then on the next cycle

everything continues as normal. Preliminary tests show that this process works but has not been tested to its fullest. A checkpoint takes approximately half a second to perform, and a fork migration takes roughly half to one second as well. As was seen in Section 19.2.10, the rollback mechanism when activated can take anywhere from 6 to 15 seconds to do. Therefore, the fork method is far superior in time.

19.2.14 Network Traffic Effects

Network traffic plays a very large role in this project. So, too, do machine loads. If a machine is highly loaded, then it cannot adequately serve the incoming packets or respond in the allocated time specified by the arbiter. This can then cause a retry, which may not actually be needed yet, but the arbiter assumes that it is. This timeout variable is thus crucial to NEPSi's performance. If set too low, the arbiter will send a retry too early; if set too high, then a missed packet will not be sent for an extended period, which wastes time for all processors. Various methods were employed to try to automatically determine a heuristic for the optimal timeout. None proved to be any more successful than others. The methods tried were an increasing timeout period, setting the timeout from the first packet received and an average. The network fluctuations in a teaching environment based on X-terminals are dramatic. A network that consists of homogeneous dedicated workstations should prove to far be less troublesome. Figure 19.19 shows the average number of retries that are encountered during a simulation with a varying number of PEs.

19.3 Discussion

This project has achieved its major goals of exploring the utility of network enabled parallel simulation in a very dynamic and unruly network environment. The NEPSi simulator was developed with the accompanying testing resources. We have observed in most cases a general simulation speed increase by a factor of 5. Larger simulations produce larger amounts of speed, up to a point. This point is governed not necessarily by communications contentions but by resource contentions. Invariably we ran out of processors or increased the number of simulations per machine, which had a much more significant effect on the speedup factor than network overheads.

There are some possible measures which should improve the results. The network here is relatively lopsided and quite a number of the machines used in the project are located on the same subnet. The relatively fast machine used as the arbiter was on this overused subnet and should ideally have been located on a less populated network node, so that packets get through with fewer collisions. Also, the simulator relies on the NFS file system so that all machines have a consistent view of a central data repository. This means, however, that the machine on which the repository exists must service all the demands from the simulators as well as do any simulation that may have been assigned to it. This became a problem when

we were conducting the 64 PE tests. Several NFS errors would occur during the simulation and on a couple of occasions this would cause the simulator to go into damage control mode and initiate a rollback; of course, this only made things worse and the whole simulation had to be aborted. This is a problem out of our control and the NFS errors experienced cannot be attributed to the simulator itself; it is at the mercy of the network environment. To counteract this problem, the simulator could be run so that when starting a simulation session all the relevant files are copied to the local temp directory on each host. This way the files would be local and would stop any problems with NFS. However, this then presents a problem for the checkpoint files in the event that a machine went down, but mirroring could be handled within a subnet if losing an entire subnet was deemed unlikely.

We also note that more useful speedup figures could be obtained with a homogeneous network. Also, while NEPSi has proven to be a highly effective implementation technique for hardware, the task allocation has been done in a very rudimentary manner. It would also be possible to have different functional entities on one processor and remove the present restriction that only one type of module can be simulated on each machine, allowing more freedom in the placement of modules.

Further reduction of the amount of network traffic on a single subnet should also be achievable by the following means:

1. Incorporate an arbiter into the daemon and have it transmit the message back to a central arbiter. This is what PVM does in some circumstances, but as explained earlier, this increases the number of packets required. As we saw, when a large simulation is being executed, the number of packets returned to the arbiter increases greatly and the subnet on which it resides can get flooded.

2. Equip the server on which the arbiter resides with multiple network adapters so that it becomes a hub for the simulation network. This will spread the traffic presently on the arbiter's subnet across several subnets. This becomes particularly attractive when one considers that one can equip a PC with several Ethernet cards for a very small outlay, and this would reduce the network contentions considerably.

Using the communications functions developed in this project, extending a commercial simulator should be the next logical step. Combining this with the above proposed extensions would produce a cheap extensible simulator than can make use of resources that are usually thought to be too slow to be of any practicable use. All this and more can be accomplished through employing networked enabled parallel simulation.

19.4 Bibliography

[1] R. E. Bryant. Bit-Level Analysis of an SRT Divider Circuit. *Technical report, CMU-CS-95-140*, Carnegie Mellon University, April 1995.

[2] CAO_VLSI team, ALLIANCE, ftp://ftp.ibp.fr/ibp/softs/masi/alliance/

[3] S. Gupta, and K. Pingali. Fast Compiled Logic Simulation Using Linear Bdds. *(TR95-1522)*, Cornell University, June 1995.

[4] S. R. Jones and K. Sammut. Learning in Linear Systolic Neural Network Engines: Analysis and implementation. *IEEE Transactions on Neural Networks*, vol. 5(4), pages 584-593, July 1994.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, Web version. http://www.netlib.org/pvm3/book/pvm-book.html, MIT Press Scientific and Engineering Computation, (ed.) Janusz Kowalik. Massachusetts Institute of Technology 1994.

[6] R. Rabenseifner and A. Schuch. Comparison of DCE RPC, DFN RPC, ONC and PVM. In Alexander Schill (ed.), DCE - The OSF Distributed Computing Environment, *International DCE Workshop, Proceedings*, pages 39-46, Karlsruhe, Germany, October 1993.