INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Validation of the Rice Parallel Processing Testbed using sorting algorithms

Ingels, Stephen Clark, M.S.
Rice University, 1989
RICE UNIVERSITY

VALIDATION OF THE RICE PARALLEL PROCESSING TESTBED USING SORTING ALGORITHMS

by

STEPHEN CLARK INGELS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE MASTER OF SCIENCE

APPROVED, THESIS COMMITTEE

J. Robert Jump, Professor of Electrical and Computer Engineering, Director

James B. Sinclair, Associate Professor of Electrical and Computer Engineering

John K. Bennett, Assistant Professor of Electrical and Computer Engineering

Houston, Texas
April, 1989
Validation of the Rice Parallel Processing Testbed 
Using Sorting Programs 
by 
Stephen Clark Ingels 

Abstract 

The Rice Parallel Processing Testbed (RPPT) is software package for simulating the execution of parallel computers. The RPPT employs execution driven simulation to perform simulation efficiently. In this thesis, the demonstration that the RPPT is accurate is made by comparing the performance of programs run on real parallel computers to the performance predicted by the corresponding simulations. A distributed quicksort and the Global Distribution Local Sort algorithm are each implemented for the Intel iPSC 16 node hypercube and for a system of 7 Sun 3 workstations connected by a local area network and running the V-System. Error of the corresponding simulations is generally less than 20%. Explanations for discrepancies and suggestions for reducing error are presented.
Acknowledgements

My two years in the Electrical and Computer Engineering Department at Rice University have been two very good years for me. I have had the privilege of working with people whose knowledge and opinions I respect and whose company I enjoy. I will miss them all.

I thank my committee members Dr. Robert Jump, Dr. Bart Sinclair, and Dr. John Bennett, for their constructive advice. I also thank Sridhar Madala and Rick Covington, who wrote the code for most of the RPPT and who showed me how to use it.
## CONTENTS

1  **Introduction**  
   
2  **Sorting Algorithms For the RPPT**  
   2.1 Sorting Algorithms Based on Partitioning  
   2.2 Sorting Algorithms Based on Merging  
   2.3 A Hybrid Approach  
   
3  **The Rice Parallel Processing Testbed**  
   3.1 The Profiler  
   3.2 Concurrent C, a Language for Parallel Programming  
   3.3 CSIM, a Discrete Event Simulator  
   3.4 ASIM, an Architecture Simulator  
   
4  **Modeling the V-System**  
   4.1 V-System Overview  
   4.2 Implementation of Distributed Quicksort on the V-System  
   4.3 Implementation of Global Distribution Local Sort on the V-System  
   4.4 The Ethernet Model  
   4.5 Profiling the V-System  
   4.6 Measuring V-System Process Creation Cost  
   4.7 Measuring Operating System Overhead for Send, Receive, Reply  
   4.8 Time Required for Process Switching  
   4.9 Global Distribution Local Sort on the V-System: Simulation vs. Reality  
   4.10 Distributed Quicksort on the V-System: Simulation vs. Reality  
   
5  **Modeling the Intel Personal Super Computer**  
   5.1 iPSC Overview  
   5.2 Implementation of Distributed Quicksort on the iPSC  
   5.3 Implementation of Global Distribution Local Sort on the iPSC  
   5.4 Modeling the iPSC Communication Network  
   5.5 Profiling  
   5.6 Measurements of Software Overhead for Message Passing  
   5.7 Process Creation and Process Switching Times  
   5.8 Global Distribution Local Sort on the iPSC: Simulation vs. Reality  
   5.9 Distributed Quicksort on the iPSC: Simulation vs. Reality  
   
6  **Implications of this Work for the RPPT**  
   6.1 Flaws and Difficulties Peculiar to this Work  
   6.2 Positive Aspects of the Methodology Used in this Work  
   6.3 Suggestions for Improving the RPPT  
   6.4 Successful Features of the RPPT  
   
References  

1  
7  
7  
12  
16  
18  
18  
21  
24  
25  
29  
29  
31  
31  
32  
33  
36  
36  
40  
40  
45  
48  
48  
52  
53  
54  
56  
59  
61  
61  
65  
68  
68  
69  
70  
71  
73
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Message Passing Times for the V-System</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>GDLS on the V-System</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>DQ on the V-System</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Sequential Code Times for the iPSC</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>GDLS on the iPSC</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>DQ on the iPSC</td>
<td>66</td>
</tr>
</tbody>
</table>
CHAPTER 1
Introduction

The Rice Parallel Processing Testbed (RPPT) is a software package which can be used to simulate concurrent programs running on parallel computers [1]. It uses a new simulation technique called execution driven simulation in order to achieve greater efficiency than other conventional simulation techniques. The objective of this work is to determine the accuracy with which the RPPT can perform simulations. In order to test the accuracy of the RPPT, experiments were developed to compare the performance predicted by RPPT simulations to the actual performance of programs on real parallel computers. The results of these experiments indicate that the RPPT can perform accurate simulations, and suggest ways of improving accuracy in some situations.

Simulation technology is important in the design and use of parallel computers. Before building a parallel computer, or modifying an existing parallel computer design, it is advantageous to be able to predict the effects of various changes. Because of the complexity of multiprocessor systems, however, it is often difficult to use analytical methods or conventional computer simulation techniques to make such predictions. Simulations must be used instead. Efficient and accurate simulations of parallel computers are hard to achieve, however. In addition to effects such as contention between instruction fetching and data fetching, which are present in uniprocessor systems, a multiprocessor simulator must accurately predict effects such as contention among processors for the interprocessor communication medium and synchronization among processes running on different processors. Three simulation techniques that have been used previously, distribution driven simulation, trace driven simulation, and instruction driven simulation, each deal with these challenges in different ways but each suffer some disadvantage [1].
Distribution driven simulations use a statistical characterization of the time required for various operations in order to calculate a probability distribution for the time required to execute a whole program. This calculation is rapid once the initial statistical characterization has been made, but these initial statistics may be hard to determine precisely (or accurately), and the resulting probability distribution may be very broad.

In trace driven simulations, the program that is being studied is first run on a real machine with trace statements inserted into it in order to record various events, such as each processor’s communication on a shared bus. The simulator then uses the trace to determine how the execution would have occurred with different communication speeds or processor speeds, for instance. The disadvantage of trace driven simulations is that different parallel computer architectures can change the order of events in a program’s execution, so that the trace does not remain valid.

Instruction driven simulation can overcome the problems with accuracy suffered by distribution driven and trace driven techniques. An instruction driven simulator is a program interpreter that, in addition to emulating each instruction of the program, records the amount of time that the instruction takes and models any synchronization delays. The disadvantage of this approach is that it is quite slow since, for every instruction in the simulated program, the simulator may have to execute hundreds of instructions.

The RPPT uses a new technique called execution driven simulation, which is similar to instruction driven simulation but incurs far less overhead. The main difference between execution driven simulation and instruction driven simulation is much like the difference between a compiled language and an interpreted language. In this approach, simulation mechanisms are compiled into the program at process interaction points. There is no simulator per se; instead the modified program records its own progress. During the
simulation, the program's sequential code executes at full speed, then pauses briefly at the
process interaction points for the added simulation code to keep track of elapsed time.

Execution driven simulation is much faster than instruction driven simulation as long as
process interaction points occur relatively infrequently, since simulation overhead is incurred
only at these points. Thus execution driven simulation is particularly well suited to
modeling parallel computer systems with distributed memory. (For the purposes of this
thesis, distributed memory means that each memory unit may be accessed by only one
processor unit.) In parallel computers with shared memory, however, one processor may
affect the execution of other processors every time it accesses a shared memory unit. In its
present form, therefore, the RPPT does not simulate parallel computers with shared memory.

The RPPT is designed to facilitate the study of a wide variety of parallel programs on a
wide variety of distributed memory MIMD computers. The RPPT provides Concurrent C, a
parallel programming language, with which parallel programs are written without
dependence on the specifics of parallel computer hardware. Separate means are provided to
specify processor characteristics and the geometry and protocols of the interconnection
network. The RPPT can then link a Concurrent C program with an architecture specification
to generate a complete simulation program. This makes it easy to study the effects of
hardware alterations on the performance of a particular program, or to evaluate the suitability
of a particular parallel computer architecture for running a range of programs.

An RPPT simulation can generate many kinds of useful information. Each time a
program reaches a process interaction point it examines flags to see what kind of information
to collect and record. A simulation can be set up to output this information as a trace of
software execution, of message traffic in the interconnection network, or of other events.
Such traces can be routed into the RPPT's graphic output program in order to better visualize
the concurrent sequences of events, which is especially useful for debugging purposes. Other information can be analyzed as it is generated and output as statistical summaries at the end of the simulation. Statistics concerning processor utilization, interconnection network traffic, the time each process spends waiting for messages or semaphores, etc., are useful for performance analysis.

There are several potential sources for error in execution driven simulations. The first is in the measurement of the time required to execute segments of sequential code. This must be done so that at process interaction points it is known how much time has elapsed during the previous sequential execution. Currently, this measurement requires the structure of each segment of sequential code in the simulation to bear close resemblance to the structure of the corresponding segment of sequential code for the real parallel computer. As processor architectures and instruction sets diverge from those of the processor on which the simulations run, this resemblance may become more remote resulting in less accurate measurements of execution time.

A second potential source of simulation error is in the specification of the interconnection network and communication protocols. In principle, it should be possible to use the RPPT to specify these exactly. The exact specifications of many real parallel systems may be hard to obtain, however. Simulation models of interconnection networks, therefore, will often be simplifications of the real interconnection networks.

Finally, the operating system routines are a likely source of simulation error. On a real parallel computer, these routines are called at process interaction points and at time slice boundaries. In the simulation, analogous RPPT routines are called at these times. It is difficult, however, to be sure that time slicing is synchronized with program execution in the same way in the simulation as in the real parallel computer. Also it may be difficult to
estimate the time required by the operating system routines on the real parallel computer based on the time required by their RPPT analogs. This is because the real routines are often optimized for speed and efficiency at the expense of simplicity. In addition, many of the routines are interrupt driven.

Determining just how much simulation error actually occurs is the objective of this thesis. The approach taken is to compare the time required for programs to run on real parallel computers to the time predicted by running the same programs on simulations of these parallel computers. To begin with, simple programs are run that test mainly a single aspect of the simulation capability, such as the ability to simulate sequential code or to simulate message passing times accurately. After each test the simulation parameters are adjusted to compensate for simulation error. Finally, full scale programs that test multiple interacting aspects of the simulation capability are run.

Two real parallel computer systems (chosen because of their availability) are used in this validation effort. One of these parallel systems is a group of Sun 3/50 workstations connected by the campus Ethernet local area network and running the V-System [2]. The other is Intel's iPSC, a machine containing sixteen 80286 based processor nodes connected in a four dimensional hypercube geometry [3]. The architecture models for these machines are based on published information and manufacturers' specifications.

Two sorting algorithms are used as the full scale programs to test the RPPT. Sorting is an application that exercises parallel computer architectures in ways that previous validation studies for the RPPT have not, so tests using sorting programs are sensitive to the accuracy of simulating different aspects of parallel computation. The previous studies utilize programs that calculate FFTs [4], multiply matrices, evaluate partial differential equations, and calculate eigenvalues and eigenvectors [5]. Sorting differs from those in that it is
nonnumeric: the main arithmetic operation in the sorting programs is 16- or 32-bit comparison, whereas the others make extensive use of floating point multiplication and division. In addition, whereas in each of those algorithms the control flow is nearly independent of the data, control flow in the sorting programs is data dependent at two levels of granularity. At the finer level, each element in an array to be sorted is compared either with other elements or with a constant value and, based on this comparison, a decision is made regarding whether or not to move the element and where to put the element. At the coarser level, some sorting schemes partition the elements into subarrays based on their values, resulting in different sizes of subarrays to which the sorting program must respond appropriately. Thus tests of simulation error using sorting programs emphasize accuracy in simulating control flow and reduce the emphasis on numeric calculations.

The following chapters first describe the sorting algorithms. Next the mechanisms of the RPPT are outlined so that it will be clear how each component participates in the overall simulation. With this material as background, the next chapter describes the V-System, the ways in which V-System parallel programming constructs were simulated with the RPPT, and the results of running the sorting programs on both the real and the simulated V-System. The next chapter does the same for Intel's iPSC. The final chapter is a discussion of the results, including suggestions for reducing the simulation error.
CHAPTER 2

Sorting Algorithms for the RPPT

A large number of parallel sorting algorithms have been proposed. Many of them are for special purpose hardware such as arrays of very simple processors connected in a mesh or a shuffle configuration [6,7,8,9]. Other algorithms have been proposed for Single Instruction Multiple Data (SIMD) architectures [10]. Still another large group of algorithms have been proposed for Multiple Instruction Multiple Data (MIMD) machines that have shared memory [11,12]. The two algorithms that were selected for implementation on the RPPT were originally proposed for implementation on MIMD architectures lacking shared memory. The RPPT currently supports only simulation of distributed memory MIMD systems. Another important consideration in the choice of sorting algorithms was that neither the V-System nor the iPSC, which were used to validate the RPPT results, provide any shared memory.

This chapter presents an overview of sorting algorithms for MIMD computer architectures lacking shared memory, including the two algorithms that were implemented on the V-System, the iPSC, and the RPPT models of these two architectures. Some implementation issues for these two algorithms will be discussed later in the chapters that describe the facilities that are available in the V-System and on the iPSC.

2.1. Sorting Algorithms Based on Partitioning

Most sorting schemes, regardless of the type of digital computer on which they are implemented, involve one or both of two fundamental procedures: partitioning and merging. Partitioning is a process by which elements of an array are redistributed into subarrays so that the range of values of the elements in each subarray does not overlap the range in any other
subarray. In the simplest case, all of the elements of the array that have a value less than some partitioning value are put into one subarray while all of the elements with values greater than or equal to the partitioning value are put into the other subarray. By recursively partitioning the subarrays until they consist of single elements (consistently moving the higher valued elements toward the same end), the array will be completely sorted. This is, in fact, the basis of Hoare’s quicksort algorithm [13, 14].

Parallel sorting algorithms based only on partitioning can be described as hard split / easy join [15]. The work is distributed to the various processors in the form of partitions which do not have overlapping ranges. Splitting up the array this way is hard compared to sending a group of consecutive elements to each processor. But, because the ranges of elements at each processor do not overlap, no merging is required when the subarrays have been sorted and are ready to be joined; they can simply be concatenated.

There are several different techniques for partitioning. An intuitively simple way of partitioning an array into $P$ subarrays is to choose $P-1$ partitioning values plus an implied partitioning value of positive infinity. Each element of the array is compared with the partitioning values and copied into a buffer corresponding to the first partitioning value greater than its own value. If the number of subarrays to be generated is a power of 2, it is useful to arrange the partitioning values as the branches of a binary tree [16]. Then partitioning requires time proportional to $N \log_2 P$, where $N$ is the number of elements and $P$ is the number of subarrays. To complete a sorting procedure based on this kind of partitioning, the subarrays may be sorted separately and then concatenated. Some practical difficulties arise, however, due to the fact that it is not known in advance how large the resulting subarrays will be; block moves of the subarrays will be necessary if enough space is not allocated. Even more block moves will be required if it is desired to write the growing subarrays over the part of the original array that has already been examined. If the elements
are organized as a linked list these problems can be avoided.

The quicksort-style partitioning scheme is perhaps the most popular means to partition an array into two subarrays [14]. In this routine, two pointers are positioned at the beginning and at the end of the array to be partitioned. At each step the lower pointer is incremented until the element it points to is greater than the partitioning value, and the upper pointer is decremented until it points to an element that is less than or equal to the partitioning value. If the two pointers have crossed each other, i.e., the upper pointer points to the element just below the element to which the lower pointer points, then the array has been partitioned; the upper pointer points to the upper end of the lower subarray, and the lower pointer points to the lower end of the upper subarray. If the two pointers have not crossed, then the two elements to which they point are swapped, and the process continues. The main reason for the quickness of the quicksort is that the inner loop of its partitioning algorithm consists only of a comparison and an increment. In C this is written as

```c
while(array[++i]<partition_val);
```

A second advantageous feature is that only those elements which do not belong in their current position are moved.

An important consideration in any algorithm based on partitioning is how to choose the partitioning values. In the case of repeatedly partitioning into two subarrays, if the partitioning value is chosen so that the subarrays are exactly half the length of the subarray from which they were generated, then the entire sorting process will require $\log_2 N$ levels of partitioning, each taking time proportional to $N$. When one subarray is larger than the other, the larger subarray may require more levels of partitioning and the efficiency of the algorithm will be less than optimal. In the worst case, when the small subarray always has length one, the larger subarray will require another level of partitioning for every element in it, and the entire sorting process will take time proportional to $N^2$.
When sorting algorithms are parallelized it is especially important to partition the array into nearly uniformly sized subarrays so as to distribute the workload evenly among processors. For partitioning into two subarrays, a common means of choosing the partitioning value is to use the median of the first, middle, and last elements in the array [17]. For arrays that are initially nearly sorted, using this median of three value does a good job, especially in comparison to simply using the value of the first element in the array as was proposed in Hoare’s original paper [13]. Schemes that involve sampling more than just three elements have also been proposed. Yang, Huang, and Chow [18] propose using order statistics to choose partitioning values. For generating $P$ partitions, they would take $n(P-1)$ samples, sort them, and use for partitioning values the $(n+i)$th sample, $i=1,2,...,P-1$. They describe a means of choosing the optimum value of $n$. Noga [19] proposes that the original array be sampled, the minimum and maximum of all samples determined, and initial partitioning values chosen that will divide the range of sampled values into even subintervals. Next the sampled elements are partitioned into these subintervals, generating a frequency histogram. New partitioning values are then calculated based on the frequency histogram.

One difficulty inherent to parallel sorting algorithms based only on partitioning is that the first partitioning pass must be performed by a single processor. Thus the first partitioning is a bottleneck.

Almes has implemented a simple parallel sorting program based on partitioning [20]. It is a distributed quicksort that runs on a collection of Sun workstations connected by an Ethernet local area network and running the V-System. The array to be sorted is first partitioned using the standard quicksort partitioning procedure. The smaller of the resulting arrays is sent to a server machine for further partitioning and sorting while the client machine continues to partition the larger of the subarrays. Almes presents some optimizations that allow the client machine to continue doing useful work even when all of the servers are
already busy and cannot accept newly partitioned subarrays, and optimizations that allow the server machines to have servers themselves.

Almes' analysis of the distributed quicksort gives the following equation for execution time, \( T \), where \( N \) is the number of elements to be sorted, \( p \) is the number of processors, \( a \) is the time per element for partitioning, and \( b \) is the time per element for moving a subarray.

\[
T = aN \log_2(N/p) + N [(2a+b)(p-1) - a \log_2 p] / p
\]

Almes' fastest version very nearly achieves the speedups predicted by this formula.

We implemented Almes' distributed quicksort for the V-System, the Intel Hypercube, and the RPPT models of both of these systems. In this implementation, the array to be sorted is initially at the client machine, as described above. The client sends partitions to the server nodes where they are sorted. The client machine has a server process running on it that can be utilized if all of the remote servers become busy; this prevents the client from wasting time. The server nodes do not have servers themselves.

An alternate sorting strategy that we did not use is the radix sort, which is popular for uniprocessor architectures. The radix sort uses what is essentially another partitioning scheme in which only a part of each element is considered during each comparison. In a binary radix sort the elements of the array are first partitioned into buffers according to the values of their least significant bit. Then the subarrays are concatenated in order in the original array space. In the next phase, the elements are partitioned according to the values of their next least significant bit but otherwise preserving the ordering of elements generated in the first partitioning. That is, if two elements do not differ in the bit position that is being examined in the current partitioning phase, then their relative order is unchanged. After as many partitioning stages as there are bits in each element, the array has been sorted. A radix sort of this kind takes time proportional to the number of elements sorted and to the number
of bits in each element. This is faster than the $N \log_2 N$ time required for sorts in which elements are compared to each other (so called compare-exchange sorting schemes) if the number of bits per element is less than $\log_2 N$. However, because all of the elements must be examined in each partitioning phase, the usefulness of radix sorting may be limited to the purely sequential sorting phases within parallel algorithms.

Other variations on radix sort are possible. Rather than partitioning the array based on the value of a single bit at a time, it is possible to consider more bits at a time. For example, 8 bits of the element may be used during each stage to select one of 256 different buffers into which the element may be moved. Also, if the elements are organized as a linked list, then the elements do not need to be moved at all; they are simply linked into one of 256 lists. After linking all of the elements into the 256 lists, the lists are then linked to each other in order.

2.2. Sorting Algorithms Based on Merging

Merging is the other fundamental procedure of sorting and is, in a sense, the inverse of partitioning. In merging, two sorted subarrays are combined into a single sorted array. An array of $N$ elements can be completely sorted in $\log(N)$ stages by merging pairs of single elements into subarrays of length 2, merging pairs of subarrays of length 2 into subarrays of length 4, etc. Each merge stage takes time proportional to the number of elements merged, as will be seen below.

Parallel sorting algorithms based only on merging are sometimes described as easy split / hard join [15]. While algorithms based on partitioning require a good choice of partitioning values to achieve optimum distribution of work among processors, algorithms based on merging, with their easy splitting, can easily achieve an optimum distribution of work among processors. This is one major advantage for merge-based parallel sorting algorithms.
A simple example of merging two sorted sequences of numbers is as follows. Pointers are set to the beginnings of the two subarrays and to the beginning of the result array. The elements selected by the pointers in the subarrays are compared and the lesser of the elements is copied into the position marked by the pointer in the result array. The pointer in the subarray from which the element came and the pointer in the result array are advanced. This is repeated until one subarray pointer has reached the end of its subarray, and then the remainder of the other subarray is copied into the result array. A merge in this fashion requires time proportional to the number of elements merged.

Sorting algorithms based on merging were implemented on the HEP parallel computer [21]. The merging phases of the algorithms are organized like a binary tree. In the first phase, processors pair off and one of each pair of processors sends its subarray to the other where it is merged with the subarray already there. In each subsequent phase, new pairs are formed among the remaining processors that have subarrays, and subarrays are merged again. With each phase the number of active processors is reduced by one half until, in the final phase, one of the two remaining active processors sends its subarray to the other, where it is merged into the final sorted array. These algorithms suffer from a disadvantage similar to that suffered by the partitioning-based parallel sorting algorithms, namely that in the final merge phase no parallelism is possible. Using 16 processors and quicksort to sort each subarray initially, a speedup of about 6.5 was obtained for sorting 8192 elements. Singgih et al. also present a sorting algorithm based on Batcher's odd-even merging [7] but its speedup is less.

Batcher's bitonic merge [7] is a technique that is more complicated than the simple merge just described, but more amenable to parallelization. In a bitonic merge, two sorted subarrays, one nonascending, the other nonascending, are concatenated, forming a single bitonic array. To complete the merge, the bitonic array is sorted into a monotonic array in a
special way. Let the bitonic array consist of \( N \) elements, where \( N \) is even. Elements are compared in pairs, element \( i \) with element \( i+N/2 \). For each pair, the lesser valued element is replaced in the lower ordered position and the greater valued element is placed in the higher ordered position. This compare and swap stage results in two subarrays of length \( N/2 \). Each subarray is bitonic. Furthermore, all of the elements of the lower ordered subarray have values less than any of the values of the elements of the higher ordered subarray. See Stone [6] for a more intuitive alternative to Batcher's proof of this result. Since the resulting subarrays are bitonic, this process is applied recursively until the whole array has been sorted into a monotonic sequence of numbers.

The conversion from bitonic to monotonic requires \( \log_2 N \) rounds of \( N/2 \) compare-exchange steps each. Because of the regularity of the comparisons that must be performed, it is possible to use \( N/2 \) comparison-exchange units simultaneously at each stage, and accomplish the data transfers by means of a perfect shuffle interconnection topology. Thus a merge can be accomplished in time proportional to the number of stages, \( i.e., \log_2 N \). Furthermore, since there are \( \log_2 N \) merges that are required to accomplish a complete sort of \( N \) numbers (not bitonic), an overall time proportional to \( (\log_2 N)^2 \) is required. Thus, if \( N/2 \) processors are available and can be configured as a perfect shuffle, bitonic sorting is a very fast algorithm.

Bitonic merging can also be used in MIMD systems. One of its advantages is that it enables all processors to participate during the entire merging procedure thus overcoming the bottleneck that exists in the binary tree merging procedure. Consider a bitonic array consisting of one non-descending subarray at one node of the multiprocessor and one non-ascending subarray at a second node. Each processor sends its subarray to the other. The first processor compares each element \( i \) of its own subarray with each element \( i \) of the other subarray, keeps the minimum of the two, and discards the maximum of the two. The second
processor does the same but keeps the maximum of each pair of compared elements and discards the minimum. The result of this is that each processor now has a bitonic subarray, and all elements of the array at the first processor have values less than or equal to the least value of the elements at the second processor. When both processors have sorted their subarrays, then the whole (but still distributed) array has been sorted. (It is not necessary to continue the bitonic merge process recursively, nor is it desirable since more efficient sorts are possible when the whole subarray is present at a single node).

Now consider a bitonic array consisting of a nondescending subarray distributed evenly between nodes 1 and 2, and a nonascending subarray distributed evenly between nodes 3 and 4. Such subarrays may have been produced by the bitonic merge described in the preceding paragraph. To sort this bitonic array, the odd processors would send their subarrays to each other and the even processors would send their subarrays to each other. The lesser numbered processor of each pair would keep the minima and the greater numbered of the pair would keep the maxima. The odd processors sort their resulting bitonic subarrays into nondescending subarrays while the even processors sort their resulting bitonic subarrays into nonascending subarrays. The final result of this process is that there are two bitonic subarrays. The lesser valued subarray is at nodes 1 and 2. The maximum of the elements in the bitonic subarray at nodes 1 and 2 is less than or equal to the minimum of the elements in the bitonic subarray at nodes 3 and 4. Each of these bitonic subarrays may now be sorted as described in the preceding paragraph to produce a completely sorted array distributed among 4 processors.

Similar procedures may be followed for bitonic arrays distributed among any power of 2 processors. Rules for determining when processors keep minima and when maxima, and when they sort into nondescending and when into nonascending subarrays, are not too complicated. Communication can be reduced by nearly half by using a kind of handshaking
between pairs of processors before they exchange data. During the handshaking, each pair of processors determines the index where their nonascending and nondescending subarrays cross. (See Stone [6] for more on the idea of crossover points.) Then only those elements on one side of the crossover point need to be exchanged. Furthermore, by using this scheme minima and maxima do not need to be calculated, and no buffer space except that originally allotted for the first subarray is needed.

2.3. A Hybrid Approach

Sometimes arrays are stored not at a single node in a multiprocessor system, but in distributed form with each segment of the array at a different node. In this situation strategies other than split, sort, then join can be developed to introduce parallelism. One of the many interesting strategies to cope with such a situation, and the second of the two sorting algorithms that I have chosen to implement, is the Global Distribution Local Sort (GDLS) algorithm [22] which could be described as a parallel hard split / parallel hard join approach. In GDLS, a coordinating process collects samples from each segment of the array in order to establish partitioning values that will divide the final sorted array into approximately equal sized subarrays, one piece for each processor. These partitioning values are then broadcast to all of the processors participating in the sort. Each processor then partitions its segment of the array around the partitioning values and sends each of the subarrays to the proper remote processor. Processor zero receives all of the subarrays less than the first partition value, processor one receives all of the subarrays between the first and second partition values, etc. When all of the subarrays have been redistributed, each processor sorts them into what becomes its new local part of the array.

The principal advantage of GDLS over many other algorithms is that GDLS transfers each element of the array no more than once (not counting the original distribution and final
concatenation that are required if the array of elements begins and ends at a single processor rather than remaining distributed). Additionally, to the extent that the coordinating process has established proper partition values, each processor is active during nearly the entire procedure. Chow and Marc propose that partitioning be done by constructing a balanced binary tree from the partitioning values, and steering each element of the local array toward the proper subarray as it descends the tree. Partitioning in this manner takes time of order \((N/P)\log_2 P\), where \(N\) is the number of elements and \(P\) is the number of processors and thus the number of subarrays that must be generated. The local sorting takes time of order \((N/P)\log_2(N/P)\). The combined time, ignoring communication overhead, is of order \((N/P)\log_2 N\), which is \(P\) times faster than a sequential quicksort. If system hardware can handle communication while the processors continue to partition, then this speed can be achieved.

In order to time the GDLS algorithm, it is convenient to have the initial array begin and end at the same node of the multiprocessor system, that is, to abandon the idea of sorting a distributed file. This means that an initial distribution phase is added to the beginning of the sorting routine and a final collection phase is added to the end of the routine. The resulting algorithm could be described as an easy split / parallel hard split / parallel hard join / easy join algorithm. As will be seen in the results chapter, this appears to be a relatively good algorithm in spite of the added distribution and collection phases. Another change to the original algorithm is that instead of using a binary tree sort during the parallel hard split phase, multiple quicksort style partitions are used. To avoid the problem of sampling, the random numbers that are sorted are distributed with uniform probability between 0 and 999. Since the partitioning values are known ahead of time, there is no need for the manager process to communicate these values to each processor.
CHAPTER 3

The Rice Parallel Processing Testbed

The goal of the RPPT is to model the execution of parallel programs on parallel computers. The RPPT consists of three main parts in addition to the parallel program itself and a specification for the parallel computer. The first is a profiler, which provides a means of determining the time required to execute sequential pieces of code. The second is a concurrent programming language that provides mechanisms for creating and controlling multiple tasks that run concurrently and communicate with each other. (Tasks include those which are part of a parallel program and those which are used in an architecture model to simulate the hardware.) The third is a discrete event simulator that is used to simulate the sequence of events that occur in the parallel computer as it executes the parallel program, based on the timing information provided by the profiler and the control information provided by the concurrent language environment. Each of these three main parts of the RPPT will be discussed in turn, along with explanations of how more peripheral parts interface to them.

3.1. The Profiler

The RPPT profilers [5] have evolved from the bb (Basic Block) profiler [23]. A basic block is a section of code with no branches into or out of the middle of it. If one instruction of a basic block is executed one time, all the rest of the instructions in that basic block will also be executed one time. The profiler identifies the beginnings of basic blocks by the presence of labels in the assembly language code. These labels usually mark places to which execution may branch. A basic block ends when the next basic block begins, or at a branch instruction.
At the beginning of each basic block $bb$ inserts new instructions that increment a counter for that basic block every time that basic block is entered. As the modified program runs, it keeps track of how often each of its basic blocks are entered. The added code has no effect on the execution of the original code, except that it becomes a little slower. The new code amounts to an overhead of only three or four extra instructions per basic block. At the end of the program the counters may be examined in order to determine how many times each basic block was executed. Programmers who wish to improve the speed of their programs may then concentrate their efforts on the most heavily used sections.

Tprof (a Timing Profiler) is a modification of $bb$ [5]. Basic blocks are identified as before, and new accounting code is inserted at the beginning of each basic block. Instead of the new code in each block incrementing a counter for that particular block, however, the new code adds the number of instructions in the block to a single counter shared among all of the blocks. Thus, upon completion of each basic block, the counter contains the total number of instructions executed so far in the program. Dividing this instruction count by the instruction execution rate for the particular processor gives a rough estimate of the time elapsed since the program began. This estimate can be used for simulation experiments.

The instruction count does not necessarily provide a good estimate of the time that a program executes. This is because different instructions may require different amounts of time; a particular program may have a larger than average share of faster or slower instructions. An improved Tprof has been written, therefore, that counts clock cycles for the instructions in each basic block, and inserts at the beginning of each basic block code that will update the shared counter with the number of cycles required for that basic block's execution [5]. To estimate the elapsed time for a program that has been profiled in this way, the cycle count is divided by the cpu clock frequency.
In the simplest case, a program is written, compiled, profiled, and executed all on the same computer and operating system. Since the RPPT, which runs on Sun computers under Unix, must be able to simulate the execution of real parallel programs on machines other than Suns running Unix, profiling must work in these situations as well. Cross profiling was invented to allow this [5]. In cross profiling, a C program is compiled to the assembly code level on the host machine (a Sun running Unix) and on the target machine. The basic blocks in both of the assembly code listings are determined. The cross profiler counts the cycles for each basic block in the target assembly code listing and uses the count as part of the code added to the corresponding basic block in the host assembly code. Thus, while the program is running on the host machine, as each basic block is entered, the cumulative cycle count is updated by the number of cycles that the corresponding basic block takes to execute on the target machine.

In order to match the basic blocks on the host machine with those on the target machine, the C code is first passed through a filter that adds identifying statements to function bodies, for-loops, switch statements, etc. The identifying statements indicate the type of C language construction that they are marking and the number of each occurrence of a given type; e.g., "function body number five", or "switch number three". These identifying statements are passed through to the assembly code level by the C compiler and are then modified by a second filter that turns them into assembly code comments. The cross profiler generates a table, based on these comments, that records which basic blocks are associated with which C language constructions.

Counting cycles has, to date, not been an exact procedure. Work currently in progress seeks to eliminate several sources of error in counting cycles. One source of error is that conditional branches require different amounts of time when they are taken than when they are not taken. The second source of error encountered in the machines that are currently
being modeled is that when a branch is taken, an instruction pipeline is flushed and must be refilled before the instructions begin to execute at full speed again. Work in progress deals with both of these problems by using a more complicated accounting mechanism at the beginning of each basic block. The new mechanism adds a number of cycles to the shared counter if the basic block was entered by falling through from the block immediately before it in memory, and a different number of cycles, usually greater, if the block was entered by a jump from some distant block. Other instruction and data caches, which are likely to be present in machines to be modeled in future work, will introduce additional sources of error. Perhaps the most difficult problem occurs when other processors share memory, in which case the time required to execute a basic block no longer depends solely upon the state of the processor upon entry into the basic block and the instructions in the block, but also depends on the activities of the other processors.

3.2. Concurrent C, a language for parallel programming

A good profiler is all that is needed to examine the execution of a sequential program on a single processor. The goal for the RPPT, however, is to study parallel programs as they execute on MIMD computers. Such programs consist of multiple processes. Each process runs as a sequential program but interacts with other processes at particular points during its execution. The profiler can be used to analyze each process's sequential code. Concurrent C [24] provides the mechanisms for running processes in parallel and modeling the process interaction points.

Concurrent C is written in C, plus about 10 lines of assembly code. It consists mainly of two parts: the main() routine, which initiates the Concurrent C driver process and then calls the user's main routine called UserMain(); and a set of subroutines that the user's program may call to create new processes, to suspend execution, to activate a suspended
process, to wait until child processes have terminated, and to interact with processes by means of semaphores and synchronous or asynchronous message passing. The Concurrent C driver is responsible for removing the next process from one of several ready lists and activating it when the currently running process suspends or terminates. Each ready list is a list of the processes that are active on a logical node. Typically, all of the processes that are assigned to one processor are assigned to the same logical node, although more than one logical node may be assigned to a processor.

Processes in Concurrent C are "lightweight"; they all run in the same address space and switching contexts among them is quick. Context switching takes three forms. The first form is a call to Transfer(desc), in which the currently running process transfers control to the process whose descriptor is given by desc. The second form is a call to Suspend(), in which the currently running process transfers control back to the process that transferred control to it. The third form of context switching occurs when a process reaches the end of its code and does a return-from-subroutine; each process's stack is initialized so that its final return-from-subroutine "returns" it to the TerminateProc() routine which does some cleaning up, then transfers control back to the Concurrent C driver process. The typical execution sequence is that the Concurrent C driver takes the next process from the ready list and transfers to it. When the process either terminates or suspends, control passes back to the Concurrent C driver. The driver takes the next process from the ready list and transfers to it, and so on. Processes other than the driver generally do not call Transfer().

The routine Activate() is also frequently used. Prior to being activated, a process may be waiting for a message to arrive, or for some other event to occur. When the event occurs and it is possible for the process to continue executing, Activate() is used to place the process at the end of the ready list for that process's logical node. Thus control will eventually pass to the process (unless another process on the logical node does not relinquish control).
Process state information is stored in the process descriptor and on the process’s stack. Each process is allocated its own array to use for its stack space. The process descriptor is a structure that has fields for recording information about whether the process is running, ready to run, waiting for a message, waiting to be activated by another process, etc., and fields to put the process into various queues and to enqueue messages that arrive for the process. For purposes of context switching, the most important fields in a process’s process descriptor are prevDesc, a pointer to the process descriptor of the process that transferred control to it, and fields in which all of the processor registers can saved.

In the typical scenario outlined above, the interplay between the state information stored in the processes’ descriptors and stacks and the state information in the processor’s registers is as follows. Suppose process Y has just suspended and control has passed to the Concurrent C driver. The processor is executing the driver’s code and using the driver’s stack. The driver gets the process descriptor of the next process on the ready list, process X for example, and calls Transfer(). Within Transfer(), the prevDesc field of process X is set to point to the driver’s descriptor so that when process X suspends, control will return to the driver. The processor’s registers are all stored in the driver’s descriptor and then reloaded from process X’s descriptor. Since the registers include the frame pointer and stack pointer, the processor state at this point is the same as what existed when process X was last in Suspend(), except for the program counter which continues to execute code in the Transfer() routine. But Transfer() returns at this point, so the processor begins executing code in process X as if process X had just returned from Suspend(). When process X eventually calls Suspend() again, Suspend() writes the state of the registers into process X’s descriptor, loads the registers from the descriptor selected by prevDesc, i.e. the driver’s descriptor, and returns. Execution of the driver process resumes as if it had just returned from its most recent call to Transfer().
3.3. CSIM, a discrete event simulator

While Concurrent C provides mechanisms for running multiple processes concurrently and the means by which they may communicate with each other, it does not provide a means for keeping track of time. CSIM [25], a layer of software that runs on top of Concurrent C, provides mechanisms for keeping track of time, and for ordering events in time. CSIM provides an event list and an active process count. Each item in the event list consists of a pointer to a process descriptor and a time, among other data. A process is scheduled to run in the future by inserting it into the event list in chronological order. The active process count is a record of how many processes are active at the current point in simulation time. When the active process count reaches zero, all events that were scheduled to occur at the current time have occurred; simulation time is then advanced to the time at which the next process on the event list is scheduled to run, and that process is activated. While a process is running, it may activate other processes. Process activations and suspensions must use CSIM routines so that the active process count remains valid. (This means that CSIM provides its own versions of Suspend() and Activate(), and its own semaphore routines.)

In addition, CSIM provides state variables, which are global integer or floating point variables. In addition to its value, each state variable has pointers to a list of conditions that depend on it. Conditions are Boolean expressions involving state variables, and each condition has an associated list of processes. A process may wait for a condition to become true by linking its descriptor into the list of processes associated with the condition and then suspending. Each time the value of a state variable is changed (by a call to AssignStatevar()), CSIM checks to see if any of the conditions involving that state variable have become true. If a condition has become true, CSIM removes all of the processes that are waiting on that condition from the list associated with that condition and activates them.
3.4. ASIM: an architecture simulator

ASIM is functionally equivalent to Concurrent C except that ASIM routines take into account the passage of time [5]. ASIM appears to higher levels of software to be identical to Concurrent C. It provides all of the same routines, such as SendMessage(), ReceiveMessage(), P(), V(), Activate(), Suspend(), that Concurrent C does, except the names of the ASIM routines have the prefix "Asim" added to them. Calling conventions are the same, and the behavior of the routines is identical except for accounting for the passage of time. Thus a simple filter can convert a Concurrent C program into an ASIM program. In practice, however, an ASIM program generally includes calls to routines that are provided by ASIM but have no analog in Concurrent C.

ASIM uses CSIM routines to simulate the execution of parallel programs. Concurrent C provides the functions Block() and Unblock() which take ready list numbers as arguments. The Concurrent C driver will not transfer to processes on a blocked ready list. In ASIM one ready list is assigned for each physical node, and blocking of the ready list is used to model access to the cpu. When the Concurrent C driver transfers to a process the process executes a sequential region of code without interruption. When it reaches a point in the code where it interacts with other processes, for instance by sending or receiving a message, the process must account for the processing time that the previous sequential code required. To do this accounting, the process blocks its ready list and then calls the CSIM routine DelayProcess() to delay for the amount of time that the sequential code required. The amount of time to delay is calculated from the cycle count that the profiled code has accumulated since the last accounting. ASIM allows a UserScale factor to be used as a multiplier for the cycle count in order to correct for systematic inaccuracies of the profiler. The accumulated cycle count is reset to zero after each accounting. When the process returns from the delay it unblocks its ready list. All of this accounting is handled within the subroutine Account().
There are some subtle points about the Account() routine. When any process returns from a call to DelayProcess(), it is activated on the node where it was when it made the call. Since a process that is doing an Account() has blocked the ready list for its node, it must move to an unblockable node before calling DelayProcess(), or it will be activated onto a blocked node and never wake up; also the node will never become unblocked. Therefore the process moves to node -1 before calling DelayProcess() and moves back to its original node after the delay is over. Another sublety is that blocking a ready list can reduce the number of processes that are active, so CSIM's active process count must be reduced by the number of processes that were on the ready list, and when a ready list is unblocked the active process count must be increased. The Block() and Unblock() routines both return the number of processes on the node's ready list.

Process interaction points are assumed to occur only within the ASIM routines for message passing, semaphore handling, process activation and suspension, etc., so that accounting can be done entirely within the ASIM routines. In many parallel processing systems, including the V-System and the iPSC, multiple processes may execute on a single processor in a time sliced mode. For some programs, this mode of operation may make a difference in the programs' behavior. When this turns out to be the case, it is possible to model time slicing in RPPT by using a modified profiler [5]. In addition to inserting instructions to update the cumulative cycle count, the profiler must insert instructions to compare the cycle count to the time slice cycle count and call the AsimNice() routine if the time slice has been exceeded. AsimNice() is a routine that calls Account(), activates the calling process onto the end of the ready list at its node, then suspends. The effect of this is that the next process on the ready list will get a chance to run. Time slicing was not used for the simulations described in this thesis.
Profiling generally cannot be applied to system calls since the code that ASIM executes may be very unlike that executed on the modeled system. Instead, software overhead for system calls, such as for sending a message, may be measured, and formulas supplied to ASIM in the form of accounting functions. For example, the routine $\text{AsimSendMessage()}$ does accounting both before and after the message is sent. In these places it calls $\text{AsimSendMessageAcct1()}$ and $\text{AsimSendMessageAcct2()}$, respectively, to find out how much processor time to account for. The arguments for these routines are the source node, destination node, and message length. The architecture modeler must supply these routines. Different accounting functions may be supplied for each architecture to calculate the amount of operating system accounting that needs to be done in each of these places.

The communication hardware of a parallel processing system is modeled through the routine $\text{UserSend()}$. $\text{UserSend()}$ is an internal routine that is used by ASIM routines whenever a process on one processor must communicate with a process on another processor. $\text{AsimSendMessage()}$, an asynchronous message passing routine, makes a good example. When a process calls $\text{AsimSendMessage()}$ to send a message to a remote node, $\text{AsimSendMessage()}$ creates a new process to simulate the transmission of the message while the sending process continues to execute. The message handler calls $\text{UserSend()}$. $\text{UserSend()}$ delays for the amount of time that it takes for the message to be transmitted to the remote node. $\text{UserSend()}$ returns when the message has arrived at the remote node, so the message handler then puts the message into a linked list of messages at the process descriptor of the receiving process and, if the receiver has suspended awaiting a message, activates the receiver.

$\text{UserSend()}$ uses CSIM queues, state variables, etc., in order to take into account all of the other message traffic that may be present. In the simple bus model, for instance, $\text{UserSend()}$ causes the calling process to wait on a semaphore that indicates whether or not
the bus is busy. If another message is being transmitted already, the message handler is put on the queue for the semaphore, and suspended until all of the previous processes have used the bus. When the process obtains the semaphore, it delays for the amount of time needed to transmit the message, and then releases the semaphore and returns.

An architecture modeler must supply a UserSend() routine for each architecture model. The Ethernet and hypercube models are described in the chapters on V-System and iPSC, respectively. Architecture models also include an initialization routine to set the cpu speed at each node.
CHAPTER 4
Modeling the V-System

4.1. V-System overview

The V-System [2] is a distributed operating system that runs on Sun workstations connected by an Ethernet LAN to each other and to one or more file servers. Processes are organized as teams. A team is one or more processes that run in the same address space. The kernel resides at each node and consists of multiple teams. Among the kernel processes are a team server, which is the only process that may create new teams (it creates teams at the request of other processes as well), a virtual graphics terminal server, which presents the user with a window, and an Ethernet server, which provides the interface to the Ethernet. Processes communicate by message passing. User processes obtain kernel services by sending request messages to the appropriate kernel process. Message passing is synchronous, maintaining a single thread of control. Additional threads of control can be initiated by process creation or loading. Processes can be started using Create(), which causes them to use the same memory space as the creating process and enables fast process switching. Alternatively, processes can be started in their own address spaces by using the LoadProgram() call. Asynchronous message passing can be derived by creating a server process that calls the V-System’s synchronous message passing routines.

Synchronous message passing in the V-System is a simple and low overhead means of communication. Send() transmits a 32 byte message to the receiver. Sixteen bytes of the message are used to indicate a message type and to specify a part of the sender’s address space from which or into which the receiver may move data. The remaining sixteen bytes have no standard function and may be used to transfer small amounts of data to the receiver.
The call to Send() returns when the receiver has finished moving data to and from the sender and has executed a Reply(). The reply is also 32 bytes in length, 16 bytes of which may be used for transferring data.

The receiver executes a Receive(), which suspends the receiver until the message arrives. If the amount of data to be transferred each way is sixteen bytes or less, then the receiver simply executes a Reply(). Otherwise, before replying the receiver may execute a MoveFrom to get data from the sender’s address space, or a MoveTo to put data into the sender’s address space. In a MoveFrom, a packet is transmitted to the sender’s kernel requesting transmission of a specified part of the accessible address space. The sender’s kernel responds by transmitting the requested data. In a MoveTo, the receiver transmits a block of data to the sender and the sender’s kernel puts the data into the specified position. The sender’s kernel then transmits an acknowledgement. In either the MoveFrom() or the MoveTo() it is the receiver of the original 32 byte message that controls the data movement. When the receiver is finished moving data, it replies, thus terminating the data transfer sequence. To summarize, each data transfer sequence consists of a Send() by the sender and a Receive() followed by any number of MoveFrom() and MoveTo calls and a Reply() by the receiver.

The V-System at Rice University has been upgraded to use a "blast protocol" for its Ethernet activity [26]. The blast protocol is an optimistic bulk data transfer protocol. Its fundamental assumption is that in most multipacket transmissions, the next packet follows the previous packet without packets from other sources arriving between them. By capitalizing on this assumption, and using the scatter-gather capability of the LANCE Ethernet interface chip, it is possible to receive the incoming data directly into its final destination rather than receiving it into a common buffer and then copying it into its final destination. Throughputs as high as 8.0 Mbits/sec have been achieved with this protocol.
The fundamental assumption breaks down, however, on a moderately loaded Ethernet and the performance reverts to only slightly better than the normal pessimistic data transfer protocol. Since the V-System at Rice University shares an Ethernet with a large number of other campus computers, high data transfer rate variability and poor modeling result when the system is used at any time other than the least busy times of the day.

4.2. Implementation of Distributed Quicksort on the V-System

Distributed Quicksort (DQ) consists of a root process and a receiver process on the root node, and branch processes on the branch nodes. The root process partitions the original array and, as each partition is generated, sends it to the next available branch process. Each branch process sorts the arrays that are sent to it, then returns the array. In order not to block the sender, the branch replies immediately to the root when the subarray is passed. When the subarray has been sorted it is sent to a receiver process that is in the same team as the root process. The receiver process puts the subarray into its proper position in the whole array and records the fact that that branch is available to sort another subarray.

The algorithm, as implemented, is not particularly good for two reasons. First, when the root process sends a subarray to a branch process it must wait for the branch process to reply before it can continue to partition. Second, the receiver process at the root node is an unnecessary complication; it was intended to prevent the branch processes from having to wait for the root process to receive their messages, but waiting at the branches would not make the sorting slower. An approach that obviates these problems would be to have the root process receive requests for work from branches and reply with subarrays to sort.

4.3. Implementation of Global Distribution Local Sort on the V-System

Global Distribution Local Sort (GDLS) uses one manager process on the node from which the program is started and a distributor process and a sorter process on each of at least
two other nodes. The manager process starts up the distributor and sorter processes, initializes the array to be sorted and then gets the starting time. The manager then sends equal fractions of the original array to each of the distributor processes. The distributors use a quicksort-like routine to partition their subarrays. Each time a new partition is generated a new process is created to send the partition to the appropriate sorter. The creation of this new process enables the distributor to keep on partitioning rather than waiting for the sorter to reply, thus avoiding a potential difficulty with the V-System’s synchronous message passing.

The distributors terminate when they have finished partitioning. When a sorter has received one partition from each of the distributors, it begins to sort them into a single subarray using a sequential quicksort. In order not to delay the other sorters which may be waiting for a partition from the distributor on the local node, the sorter is given a priority just less than that of the distributor. (The distributor is given priority \texttt{REAL\_TIME1}, preventing most other processes that may be on the node from running and perturbing the measurements.) As each sorter finishes its sorting, it returns the subarray to the manager where it is concatenated into the final sorted array. To get the arrays in the correct order, the manager uses the V-System routine \texttt{ReceiveSpecific()} to receive from each sorter in turn.

4.4. The Ethernet model

The number of packets is calculated for each message when UserSend is called. One process is created for each packet. The packet processes deliver their packets in order, and UserSend waits until the process representing the last packet terminates. Packet processes first wait on a queue at the node from which they originate. Then they contend for the Ethernet with packet processes from other nodes. A CSIM state variable indicates how many packet processes are transmitting on the Ethernet.
A packet process begins transmitting when it sees that the Ethernet is idle. After a one-way propagation delay (called the collision window), the signal has reached all of the other nodes on the net, and the state variable is incremented to indicate that the packet is being transmitted. If a packet from another node began transmission just before this packet started, then the state variable will be greater than one. The packet process checks the state variable at the end of the collision window and if the value is greater than one it generates a jamming signal. If the transmission appears to be clear so far, the packet process delays for another one-way propagation delay before checking the state variable again. If the state variable is equal to one, indicating that this packet is the only one on the Ethernet, then this packet process continues until the packet has been completed. If, on the other hand, another packet process began transmission after this packet was started, but before the state variable was made nonzero, it will be jamming at this point, so the state variable will be greater than one. In this case, this packet process must stop transmitting, but does not need to jam. It calculates how long to wait before trying again based on a truncated binary exponential backoff scheme. Rather than dropping a packet after too many collisions as the real ethernet protocol would do, the model restarts the backoff scheme.

4.5. Profiling for the V-System

The V-System has its own C compiler called cc68. This compiler is a modified version of the portable C compiler. Because of its origins, cc68 does not produce very efficient code. When it references an array, for instance, it loads the index into a register, left shifts the index depending on the size of the items in the array, adds the array's starting address to get the address of the array variable, and finally does a register indirect memory reference to get the item of interest. The Sun Unix C compiler, in contrast, does the same thing with a single instruction using the register indirect with displacement addressing mode. Because of such
differences, it was not possible to profile directly the code that was to be executed during a simulation and have it produce cycle counts that were very close to the actual cycle counts, even though both real and simulated programs were being run on Sun 3/50 machines.

To overcome this difficulty, the programs were cross profiled. The simulation code was filtered through the Sun Unix C preprocessor, then moved to the V-System where cc68 compiled it to the assembly language level. The cc68 assembly code was then profiled and the resulting cycle counts for each basic block used for the corresponding basic block in the code compiled by the Sun Unix C compiler.

The effectiveness of the cross profiling was tested using a sequential quicksort program. Since the parallel sorting programs were expected to spend the majority of their time doing computations very similar to those in the sequential quicksort, if the cross profiling was successful with the sequential quicksort it should be successful with the parallel sorting programs as well. The sequential quicksort was run for array sizes ranging from 125 to 256000 integers. The real times measured are averages of five runs, each using the same set of numbers from a pseudorandom number generator. The clock resolution is five milliseconds. The simulations were run with UserScale values of 1.0 and 1.06. As can be seen in figure 4.1, simulated times for UserScale = 1.06 are very close to the real times. The worst fit is for arrays of size 500, where the simulated time is about seven percent too small.
Figure 4.1

Comparison of sequential quicksort run on the V-System and on the V-System simulation at two UserScale values.
4.6. Measuring V-System process creation cost

The GDLS algorithm uses process creation to allow the distributor processes to continue working while the created process sends a partition and blocks waiting for a reply. It was necessary therefore to establish the cost of process creation. To establish this cost with accuracy, given the coarseness of the system clock, the total time for creating up to 140 new processes was measured and divided by the number of processes created. Regardless of the variations in number of created processes, their stack sizes, and the arguments that were passed to them, process creation consistently cost eleven milliseconds per process.

4.7. Measuring Operating System Overhead for Send, Receive, Reply

A simple program was written to measure the amount of operating system software overhead that was involved in V-System message passing. A receiving process was first loaded onto a remote machine. Then the main process repeatedly called Send() to send to the receiver. When the receiver received a message, it called MoveFrom() and then Reply(). The number of bytes transferred using MoveFrom() was varied from 0 to 256000. The clock was read at the main process before and after each round of communication. The averages of 100 rounds at each transfer size are reported in column 4 of table 4.1. (The blast protocol enabled the transmission of over two million bits in 261 milliseconds, a throughput that is 78% of the Ethernet's 10Mbits/sec rating.)

The same program was then simulated without assessing any time for software overhead so that the communication delays were due only to the time that UserSend() indicated would be spent using the Ethernet. The V-System Send() routine was modeled by AsimSendRequest(). The V-System Receive() and MoveFrom() combination was modeled by AsimReceiveRequest(). The V-System Reply() was modeled by AsimReply(). These times are reported in column two of the table. (Note that 256,000 bytes in 210 milliseconds is
equivalent to a throughput of 97.4% of the Ethernet’s maximum bandwidth. Since there is no software overhead, a 100% throughput should have been obtained, if not for the Ethernet packet headers that were added.) The difference between the real times and the simulated times without software overhead was found to be 4.4 milliseconds per data transfer plus 0.185 microseconds per byte transferred. The 4.4 milliseconds was divided equally among calls to Account() in four places: in AsimSendRequest() before UserSend() was called, in AsimReceiveRequest() after the initial message arrived, in AsimReply() before UserSend() was called to transmit the reply, and in AsimSendRequest() after the reply arrived. The time per byte was assessed within AsimReceiveRequest(). A similar time per byte was assumed to be required for the MoveTo() operation, so a time per byte was also assessed within AsimReply(). The programs used in this thesis did not use MoveTo(), so this assumption remains untested.

The simulation times for the message passing program with software overheads included are given in column three of the table. The correspondence between the simulation times and the real times is fairly close. The real times are generally a little larger than the simulated times, probably because of traffic interfering with the blast protocol. Some of the larger differences occur when variability in the real times is greatest. For zero length data transfers, where there is no need to move data even if the wrong packet comes in next, the real time is less than the simulated time, although by less than a tenth of the standard deviation.

Data transfer times between teams on the same machine were measured in the same way. For local data transfer there was a 3.2 millisecond overhead per transfer and a 0.25 microsecond cost per byte moved. The cost per byte corresponds well with what was expected based on eight cycles for the MC68020 instruction "MOVE +(a0),+(a1)", seven cycles for a conditional branch, and a clock rate of 15 MHz.
Table 4.1

<table>
<thead>
<tr>
<th>bytes transmitted</th>
<th>simulated transmission times with ethernet delays only, no operating system overhead</th>
<th>simulated transmission times with accounting for operating system overhead</th>
<th>real transmission times in milliseconds (±std.dev.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.29</td>
<td>4.68</td>
<td>4.3 (±5.0)</td>
</tr>
<tr>
<td>125</td>
<td>.35</td>
<td>4.77</td>
<td>5.9 (±4.9)</td>
</tr>
<tr>
<td>250</td>
<td>.45</td>
<td>4.89</td>
<td>6.1 (±4.9)</td>
</tr>
<tr>
<td>500</td>
<td>.65</td>
<td>5.14</td>
<td>5.8 (±4.9)</td>
</tr>
<tr>
<td>1000</td>
<td>1.05</td>
<td>5.63</td>
<td>7.7 (±4.2)</td>
</tr>
<tr>
<td>2000</td>
<td>1.88</td>
<td>6.65</td>
<td>9.7 (±2.6)</td>
</tr>
<tr>
<td>4000</td>
<td>3.51</td>
<td>8.65</td>
<td>10.0 (±1.4)</td>
</tr>
<tr>
<td>8000</td>
<td>6.80</td>
<td>12.68</td>
<td>14.2 (±4.9)</td>
</tr>
<tr>
<td>16000</td>
<td>13.36</td>
<td>20.71</td>
<td>27.0 (±36.4)</td>
</tr>
<tr>
<td>32000</td>
<td>26.49</td>
<td>36.80</td>
<td>39.8 (±28.0)</td>
</tr>
<tr>
<td>64000</td>
<td>52.73</td>
<td>68.96</td>
<td>68.7 (±9.3)</td>
</tr>
<tr>
<td>128000</td>
<td>105.24</td>
<td>133.31</td>
<td>133.0 (±12.1)</td>
</tr>
<tr>
<td>256000</td>
<td>210.22</td>
<td>261.97</td>
<td>260.9 (±31.8)</td>
</tr>
</tbody>
</table>
Figure 4.2

Round trip communication times measured on the V-System and its simulation with and without overhead.
4.8. Time Required for Process Switching

Both sorting programs require some process switching. The time required for process switching should be accounted for in the AsimSuspend() routine, which is called each time a process suspends and control passes to the next process that shares the node. A test to measure the time required for process switching would involve running two processes and varying the number of times they transfer control to each other. It is difficult, however, to determine how many kernel processes may run in the time between the two test processes, and whether the same kernel processes will be active when other programs run. Furthermore, it may be difficult to find a way to cause the processes to transfer control to each other without also doing other things, such as transferring data, that would confound the measurement of the process switching time. A good test for measuring the time required for a process switch was not devised. Times for accounting within AsimSuspend() were arrived at by trial and error, using the sorting programs themselves. Unfortunately, this amounts to little more than a "fudge factor". The time assessed to AsimSuspend() ranged from 0ms to 5ms.

4.9. Global Distribution Local Sort on the V-System: Simulation vs. Reality

The accounting times obtained from the experiments described above were used to simulate GDLS, except that UserScale was set to 1.0 instead of 1.06. A five millisecond cost for process switching was arrived at by trial and error. Times reported for execution on the real system are the minimum of five runs each. In some cases more runs were made when the real times seemed unusually large or variable. It was assumed that long times and high variability in the real runs was probably due to other traffic on the Ethernet.

The resulting simulation times are compared to the real times in table 4.2. For the case of using two distributor/sorter nodes, the simulated times are consistently greater than the real
times. As the number of processors is increased, the actual times increase relative to the times predicted by simulation. A possible explanation for this trend is based on the fact that the number of data transfers for the GDLS algorithm increases as the square of the number of processors. With two nodes exchanging data, there is only one other node for each sorter to receive packets from, so there is little chance for interference with the blast protocol. With five nodes all sending data to each other, each sorter process is likely to receive initial messages from some distributor while it is in the middle of a bulk data transfer from another distributor. As contention for the Ethernet increases, the real blast protocol becomes less efficient, thus adding time to the system software overhead. Meanwhile, the simulations calculate the system software overhead in a simple manner that does not take contention into consideration; the simulated software overhead is always the average overhead as measured in the experiments described above. It may be difficult to model the blast protocol more accurately, however, without some changes in the organization of the RPPT. The problem is that the RPPT separates communication into a system software part and a hardware part, whereas the V-System blast protocol intertwines the two parts in a way that is much more complicated.

Inaccuracies in simulating the blast protocol explain the differences between the real and simulated times for large numbers of integers, but the blast protocol only takes effect when more than 1024 bytes, 256 integers, are moved per data transfer. For two processors the blast protocol takes effect for 1024 or more integers sorted; for three processors, 2304 or more integers; for four processors, 4096 or more integers; and for five processors, 6400 or more integers. Thus the blast protocol cannot explain the difference between the simulation and real times for small numbers of integers.

A possibility that can be eliminated as an explanation for the differences at small numbers of integers is that the cross profiling is less accurate for parts of the algorithm other
than the partitioning and sorting. While there may be some inaccuracy in the cross profiling, if it were to have a significant effect, that effect would be in the same direction regardless of how many processors were involved. In actuality, though, the simulated times are longer than the real times for 2 processors, and decrease relatively as the number of processors increases until the simulated times are shorter than the real times for 5 processors.
Table 4.2

<table>
<thead>
<tr>
<th>Integers sorted</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
<th>5 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sim secs</td>
<td>real secs</td>
<td>error %</td>
<td>sim secs</td>
</tr>
<tr>
<td>125</td>
<td>0.07</td>
<td>0.05</td>
<td>40</td>
<td>0.09</td>
</tr>
<tr>
<td>250</td>
<td>0.07</td>
<td>0.06</td>
<td>17</td>
<td>0.10</td>
</tr>
<tr>
<td>500</td>
<td>0.08</td>
<td>0.07</td>
<td>14</td>
<td>0.10</td>
</tr>
<tr>
<td>1000</td>
<td>0.11</td>
<td>0.09</td>
<td>22</td>
<td>0.12</td>
</tr>
<tr>
<td>2000</td>
<td>0.15</td>
<td>0.13</td>
<td>15</td>
<td>0.16</td>
</tr>
<tr>
<td>4000</td>
<td>0.23</td>
<td>0.21</td>
<td>10</td>
<td>0.22</td>
</tr>
<tr>
<td>8000</td>
<td>0.30</td>
<td>0.37</td>
<td>8</td>
<td>0.35</td>
</tr>
<tr>
<td>16000</td>
<td>0.74</td>
<td>0.70</td>
<td>6</td>
<td>0.60</td>
</tr>
<tr>
<td>32000</td>
<td>1.41</td>
<td>1.33</td>
<td>6</td>
<td>1.11</td>
</tr>
<tr>
<td>64000</td>
<td>2.79</td>
<td>2.67</td>
<td>5</td>
<td>2.16</td>
</tr>
<tr>
<td>128000</td>
<td>5.73</td>
<td>5.41</td>
<td>6</td>
<td>4.44</td>
</tr>
<tr>
<td>256000</td>
<td>11.90</td>
<td>11.33</td>
<td>5</td>
<td>9.09</td>
</tr>
</tbody>
</table>

GDSL for the V System, Simulation vs Best Case Real Execution Times with cross-profiler correction factor of 1.0. Sim time for process suspension.
Figure 4.3

Percentage simulation error for GDLS on the V-System using two to five processors.
4.10. Distributed Quicksort on the V-System: Simulation vs. Reality

The same accounting times were used for distributed quicksort as for GDLS. Two other sets of simulations were also run. In one, the process switch time was set to 3.5 ms instead of 5.0 ms; this improved the fit of the simulations for small array sizes. In the other, the process switch time was set to 3.5 ms and the UserScale was set to 1.06 as the sequential quicksort test indicated was appropriate; this improved the fit of the simulations at large array sizes for the cases where there were zero remote branches or three or four remote branches, but made the simulation times too large for the 1 or 2 branch cases.

The simulated times are compared to the actual times in table 4.3. The real times are averages of five runs each, except that runs that were unusually long were discarded under the assumption that unusually long runs were due to other traffic on the network.

Again the blast protocol can explain the significant trend observed in these results. When there are no helper nodes, there is no message passing to remote nodes, so the blast protocol has no effect and the simulations closely predict the actual times. When there is one helper node, the blast protocol is used. There are not, however, multiple processes contending for access to the Ethernet and interfering with blasts. Therefore, the real times are better than expected. As the number of helper nodes increases, there can be more interference with blasts. When four helper nodes are used, enough interference occurs that the blast protocol performs more poorly than predicted by the simulations.
### Table 4.3

**Distributed Quicksort on V System**

Profiler correction factor of 1.06

3.5 milliseconds assessed for each process switch

<table>
<thead>
<tr>
<th>integers sorted</th>
<th>0 branches</th>
<th></th>
<th>1 branch</th>
<th></th>
<th>2 branches</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>sim</td>
<td>% error</td>
<td>real</td>
<td>sim</td>
<td>% error</td>
</tr>
<tr>
<td>125</td>
<td>.02</td>
<td>.02</td>
<td>-4</td>
<td>.02</td>
<td>.03</td>
<td>8</td>
</tr>
<tr>
<td>250</td>
<td>.06</td>
<td>.07</td>
<td>2</td>
<td>.05</td>
<td>.06</td>
<td>26</td>
</tr>
<tr>
<td>500</td>
<td>.12</td>
<td>.14</td>
<td>14</td>
<td>.10</td>
<td>.12</td>
<td>25</td>
</tr>
<tr>
<td>2000</td>
<td>.24</td>
<td>.26</td>
<td>8</td>
<td>.18</td>
<td>.21</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>.51</td>
<td>.60</td>
<td>16</td>
<td>.39</td>
<td>.46</td>
<td>19</td>
</tr>
<tr>
<td>8000</td>
<td>.88</td>
<td>.93</td>
<td>7</td>
<td>.62</td>
<td>.70</td>
<td>14</td>
</tr>
<tr>
<td>16000</td>
<td>1.51</td>
<td>1.58</td>
<td>4</td>
<td>1.02</td>
<td>1.20</td>
<td>17</td>
</tr>
<tr>
<td>32000</td>
<td>2.56</td>
<td>2.59</td>
<td>1</td>
<td>1.91</td>
<td>1.77</td>
<td>-7</td>
</tr>
<tr>
<td>64000</td>
<td>5.22</td>
<td>5.31</td>
<td>2</td>
<td>3.21</td>
<td>3.48</td>
<td>8</td>
</tr>
<tr>
<td>128000</td>
<td>10.71</td>
<td>10.92</td>
<td>2</td>
<td>6.55</td>
<td>7.89</td>
<td>21</td>
</tr>
<tr>
<td>256000</td>
<td>22.50</td>
<td>22.95</td>
<td>2</td>
<td>13.88</td>
<td>16.64</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>integers sorted</th>
<th>3 branches</th>
<th></th>
<th>4 branches</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>sim</td>
<td>% error</td>
<td>real</td>
</tr>
<tr>
<td>125</td>
<td>.03</td>
<td>.03</td>
<td>0</td>
<td>.03</td>
</tr>
<tr>
<td>250</td>
<td>.05</td>
<td>.05</td>
<td>-4</td>
<td>.05</td>
</tr>
<tr>
<td>500</td>
<td>.09</td>
<td>.08</td>
<td>-11</td>
<td>.09</td>
</tr>
<tr>
<td>1000</td>
<td>.11</td>
<td>.12</td>
<td>1</td>
<td>.11</td>
</tr>
<tr>
<td>4000</td>
<td>.32</td>
<td>.32</td>
<td>1</td>
<td>.31</td>
</tr>
<tr>
<td>8000</td>
<td>.52</td>
<td>.47</td>
<td>-9</td>
<td>.49</td>
</tr>
<tr>
<td>16000</td>
<td>.90</td>
<td>.85</td>
<td>-5</td>
<td>.85</td>
</tr>
<tr>
<td>32000</td>
<td>1.24</td>
<td>1.14</td>
<td>-8</td>
<td>1.25</td>
</tr>
<tr>
<td>64000</td>
<td>2.68</td>
<td>2.46</td>
<td>-8</td>
<td>2.61</td>
</tr>
<tr>
<td>128000</td>
<td>5.91</td>
<td>5.77</td>
<td>-2</td>
<td>5.10</td>
</tr>
<tr>
<td>256000</td>
<td>10.67</td>
<td>10.36</td>
<td>-3</td>
<td>10.20</td>
</tr>
</tbody>
</table>
Figure 4.4

Percentage simulation error for Distributed Quicksort on the V-System with one to five processors.
CHAPTER 5
Modeling the Intel Personal SuperComputer

5.1. iPSC Overview

The Intel Personal SuperComputer (iPSC) [3] at Rice University is a parallel computer consisting of 16 processor nodes connected in a four dimensional hypercube configuration much like the Cosmic Cube [27]. Nodes whose binary node numbers differ at only a single bit are connected by a communication channel to form an edge of the cube. Figure 5.1, from the iPSC manual, shows how an iPSC node is organized. The main processor at each node is an Intel 80286 running at MHz and connected to 2.5 megabytes of private memory. Adjacent nodes in the iPSC are connected by asynchronous communication hardware. Each node has one Ethernet-like connection to each neighboring node. Since the iPSC is designed to be expandable up to a seven dimensional hypercube configuration, at each node there are seven Ethernet interface chips to make the connections. In addition to communicating with neighbor nodes, all of the nodes are connected to another Ethernet over which a front end machine, called the cube manager, can load them, send them data, and receive results from them. The cube manager is the only processor that communicates with a disk storage unit or with the outside world via a terminal or a local area network.
Figure 5.1

Organization of iPSC Nodes.
The iPSC may be run under either Intel’s operating system or the Crystalline operating system [28]. The work reported in this thesis was all done using the Intel operating system. An important advantage of this operating system is that multiple processes may be run at each node. A process running on the cube manager may load new processes into the nodes by calling load(). Each process runs in a separate, protected address space.

Processes communicate by asynchronous message passing. The sender uses send() to transfer a message from its memory space to a buffer provided by the operating system at the receiving node. The call to send() returns as soon as the operating system on the sending node has registered the request to transfer the data. Alternatively, the sender can call sendw(), which will cause the sender to be suspended until the message has been transferred out of its memory space so that that memory can be altered without affecting the message.

Once a process has sent a message, the operating systems on the nodes handle the routing of the message. Messages between nodes that are not immediate neighbors must pass through intermediate nodes. When a message arrives at a node, the operating system at that node examines the message’s destination node number to see if the message has arrived at its destination or if it must be forwarded. If the message has arrived at its final destination, the message is copied into the buffer specified by the receiving process, or is stored until a process does specify a buffer.

There are two subroutines that a process may use to receive a message. The recvw() subroutine causes the receiving process to suspend until a message of the specified type has arrived. (A message type is assigned to a message by specifying an integer value for the type as an argument in the send() or sendw() call.) When recvw() returns, the message has been written into a buffer specified by the receiving process in arguments to recvw. The recv() subroutine does the same as recvw() except that it returns as soon as the operating system at
the receiving process’s node has logged the request to receive a message. The receiver must use the \texttt{status()} call to determine when a message has been received. \texttt{Recv()} was not used in the programs for this thesis.

A limitation of the iPSC message passing routines is that the maximum message length is 16K bytes. For the sorting programs described in this thesis, the array sizes were substantially larger than 16K, so that many of the messages that needed to be sent were too big. To overcome this difficulty, the new routines \texttt{LongSendw()} and \texttt{LongRecvw()} were written for both the iPSC programs and their matching simulations. These routines send messages 16K bytes at a time and recognize the first part less than 16K bytes in length as the last part of the message. The routines depend on the node operating systems to deliver the 16K parts in the order that they were sent. Messages from different sources must not have the same type or else they may be intermixed.

The \texttt{probe()} subroutine returns the next message of the specified type, or a negative number if no message of the specified type has arrived for the calling process yet. This is useful so that a process may determine which messages have arrived and then choose to receive a particular one of them. For the simulations, an equivalent of \texttt{probe()} was added to Concurrent C and ASIM. This new function, called \texttt{CheckMessage()}, does not interfere with other parts of the RPPT since it has no effect on the RPPT data structures, but merely returns information about the messages that are in a process’s queues.

Because the iPSC uses the 80286 processor at each node and in the cube manager, programs written for the iPSC must deal with the limitations of the 80286. The '286 uses 16 bit addresses and operates on 16 bit data. One consequence of this is that a standard integer in C for the iPSC is 16 bits in size. The programmer must specify \texttt{long} if a 32 bit integer is needed. For this work, 16 bit integers were sorted. (If 32 bit integers had been sorted, each
comparison of two integers would have involved separate comparisons of the upper and lower halves. In the simulation code, however, a 32 bit comparison is done in a single step. The real basic blocks, therefore, would not map accurately onto the simulation basic blocks during cross profiling.)

In order to increase the memory addressing capability to more than 64K bytes, the '286 uses a segmented addressing scheme, with 64K bytes being the maximum segment size. The memory segmentation affects programs in several ways. There is a distinction between calls to subroutines in the same segment (near calls) and calls to subroutines in other segments (far calls). Similarly, there are near pointers and far pointers. The C compiler for the iPSC allows the programmer to include with each variable or function definition a keyword that indicates whether it is near or far. Alternatively, a "memory model" may be specified at compile time in order to set defaults. The memory model used for the work described in this thesis was the "huge" memory model, in which all variables and functions are assumed to be far. In addition, the huge memory model causes arrays to be defined as huge, which means that special array arithmetic is used in order to allow the array to extend over several segments. Long integers must be used to index into a huge array.

5.2. Implementation of Distributed Quicksort on the iPSC

The implementation of DQ on the iPSC is straightforward in comparison with the implementation of DQ on the V-System. The first process to start running is a process on the cube manager that loads branch processes onto consecutively numbered nodes, starting with node 0, and then loads the root process onto node 0. The process on the cube manager then waits to receive timing results from the root process.

The root process initializes the array of pseudorandom integers. The same algorithm is used to generate the integers for this program as for all of the programs for this thesis, thus
the same integers are sorted in every case. The root process then reads the clock, which has a 5ms resolution, and begins sorting. The root process partitions the array. After each partition, the smaller of the resulting subarrays is dispatched to the next available branch process, where it is sorted. When all of the subarrays have been sorted and returned, the root process reads the clock again, checks to make sure the integers really are in order, and then reports the elapsed time to the process on the cube manager.

A branch node sends the subarray back to the root process as soon as it has been sorted, but the root process does not receive it until it is ready to send the next subarray to that branch or until the end of the program when all of the subarrays have been dispatched. The root process keeps track of which branch process is sorting each subarray. When the branch process returns the sorted subarray, it returns it in a message with a type that indicates which branch did the sorting. By using the probe() function for each message type in turn, the root process can find out which subarray is being sent and then receive it directly into its final position in the sorted array. Using probe() also allows the root process to receive from the remote branches in preference to the local branch.

DQ on the iPSC is a better implementation than DQ on the V-System. The root process does not have to wait for each branch process to reply before continuing to partition. There is not the unnecessary complication of a separate receiver process at the root node.

5.3. Implementation of Global Distribution Local Sort on the iPSC

GDLS on the iPSC is also straightforward. The first process runs on the cube manager. This process loads distributor and sorter processes on a specified number of consecutive nodes, and a manager process on the next node. The process on the cube manager sends messages to each process to inform it of the number of distributor/sorter nodes and the size of the array that is being sorted. It then waits for the timing results from the manager process.
The manager process generates the array of pseudorandom integers, reads the starting time, and then sends subarrays to the distributors. The manager then waits to receive the sorted subarrays from the sorters in order, using the message type to indicate at which sorter each subarray was sorted. When all of the subarrays have been received, the manager reads the ending time, double checks to make sure the array is sorted, and then reports the elapsed time to the process on the cube manager.

As each distributor receives its subarray to distribute, begins to partition and dispatch the resulting subarrays. Messages from the distributors to the sorters do not need to be typed since the sorters simply wait for as many subarrays to arrive as there are distributors; the order does not matter. When each sorter has received one subarray from every distributor it sorts the combined subarrays and sends the sorted result back to the manager. It is beneficial to prevent the sorter on each node from beginning to sort until the distributor on the same node has finished distributing, since to do otherwise would delay the other sorters and reduce parallelism. Since the node operating system does not support different process priorities, the best way to accomplish this is to have each distributor send the subarray to its local sorter last. As GDLS is currently implemented for the iPSC, this is usually but not always the case.

5.4. Modeling iPSC Communication Network

When the simulation is started, the architecture initialization routine creates one message handler process for each node of the simulated hypercube. These message handler processes simulate the passage of messages through the iPSC communication network. To simulate transmission of a message, the UserSend() routine first generates one packet for each 1024 bytes of the message. The packet contains fields that indicate the source process, the source node, the destination node, and the number of bytes in the packet. None of the actual message is included, however, since UserSend() does not need to do the actual data transfer.
The packet also contains a field that indicates if the packet is a data packet or an acknowledgement packet. The last packet of a message has a field set to indicate that it is the last. After generating the packets, UserSend() links them into the outgoing packets queue for the message handler routine at its node, sets the message handler’s event flag, then suspends. It is reactivated by the message handler at the destination node when the last packet arrives.

Each message handler executes a perpetual loop. In the first part of the loop, the handler waits until some process sets an event flag to indicate that a packet has arrived. The event flag for each node is a CSIM state variable, and a CSIM condition is set up to be true when the event flag is set. The message handler uses the CSIM WaitCondition() call to ensure that it is activated immediately upon the arrival of the packet.

When the message handler is activated, it finds the packet in a queue of packets for it and begins the second part of the loop. If the packet is an acknowledgement, the message handler simply discards it. (This model assumes that there is no loss of packets, so acknowledgements are useless. They are present in the model only because they are used in the real system.) If the packet is a data packet from another node, the message handler creates an acknowledgement packet and puts it in a queue of outgoing packets. There is one such queue for each dimension of the hypercube, representing each of the neighboring nodes to which packets may be passed. The message handler uses the most significant mismatched bit between the current node and the destination node to decide into which queue to link the packet. If the packet has reached its final destination and the packet is the last packet of the message, the message handler activates the process that originally called UserSend() to generate the packet. If the packet is to be passed on to another node, the message handler links it into the appropriate queue of outgoing packets.
In the third part of the loop the message handler transmits packets to its neighboring nodes on a round robin basis. To simulate transmission of outgoing packets, the message handler waits on a semaphore that represents the private Ethernet link between neighboring nodes. There is one such semaphore for each edge of the hypercube, and the message handler uses its node number and the link number to determine which semaphore to wait on. When the message handler obtains the semaphore, it delays for the amount of time necessary to transmit the packet, and then releases the semaphore. Then it moves the packet into the arriving packets queue at the neighboring node and sets the neighbor’s event flag.

After transmitting all outgoing packets, the message handler begins the loop again. If packets have arrived during the time that the message handler was transmitting, the event flag will be set and CSIM will not suspend the message handler when it waits on the condition. If no packets have arrived, then the message handler suspends.

5.5. Profiling

The iPSC uses an Intel 80286 microprocessor for the cpu at each node. Programs to be simulated on the Sun 3/50 must be cross profiled. The source for the simulations was preprocessed on the Sun, then transferred to the iPSC where it was compiled to the assembly language level using the huge memory model. The assembly code was then transferred back to the Sun where it was analyzed by the cross profiler. The cycle counts for its basic blocks were then used in the 68020 simulation code.

The merge program did an excellent job of matching basic blocks from the two assembly source files. For critical sections of the program, the partitioning and sorting loops in particular, the results of the merge program were checked by hand. This turned out to be necessary for some of the while() loops where the test was done at the beginning of the loop in the 68020 code but at the end of the loop in the 80286 code. The profiler had reversed the
counts assigned to the basic blocks at the beginning and end of the loops. This would be important when the loops were entered many times and exited after very few iterations.

The effectiveness of the profiling was tested using a sequential quicksort program. Since the quicksort consists of two passes, one that does partitioning followed by a second that does an insertion sort, the program was run in three ways to better check the profiling of each piece of code. In the first test, the main program called the insertion sort routine directly, bypassing the partition phase entirely. Sorting this way is extremely slow, so array sizes ranged from 125 integers to only 2000 integers. The results are shown in table 5.1. The cross profiled simulation consistently predicts a time that is 94% of the real time, indicating that the profiled cycle counts need to be boosted by a factor of 1.06.

In the second test, the partitioning phase was called but the final insertion sort pass was omitted. (The error checking at the end of the program naturally reported that there were very many sorting errors.) Again, the cross profiled counts needed to be boosted by a factor of 1.06 for the simulation to match reality. The resolution of the clock on the real system may have been responsible for some of the discrepancy noticed for small numbers of integers.

Finally, the whole quicksort program was called. This time the simulated results needed to be scaled up by a factor of 1.08. It was expected that the factor would be 1.06 as it had been for the previous two tests. A possible explanation for the factor being greater is that the insertion sort profiling provides a greater underestimate for cases in which only small arrays are sorted. In the quicksort, after the partitioning phase, the unsorted pieces are a maximum of 20 integers long for these programs. A good test would have been to run the insertion sort simulation with fewer than 20 integers in the array.
### Insertionsort Only

<table>
<thead>
<tr>
<th>ints sorted</th>
<th>real (ms)</th>
<th>sim (ms)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>245</td>
<td>229</td>
<td>1.06</td>
</tr>
<tr>
<td>250</td>
<td>1015</td>
<td>952</td>
<td>1.06</td>
</tr>
<tr>
<td>500</td>
<td>4010</td>
<td>3765</td>
<td>1.06</td>
</tr>
<tr>
<td>1000</td>
<td>16535</td>
<td>15528</td>
<td>1.06</td>
</tr>
<tr>
<td>2000</td>
<td>64230</td>
<td>60337</td>
<td>1.06</td>
</tr>
</tbody>
</table>

### Quicksort without Insertionsort phase.

<table>
<thead>
<tr>
<th>ints sorted</th>
<th>real (ms)</th>
<th>sim (ms)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>10</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>250</td>
<td>30</td>
<td>26</td>
<td>1.15</td>
</tr>
<tr>
<td>500</td>
<td>70</td>
<td>65</td>
<td>1.07</td>
</tr>
<tr>
<td>1000</td>
<td>175</td>
<td>163</td>
<td>1.07</td>
</tr>
<tr>
<td>2000</td>
<td>365</td>
<td>345</td>
<td>1.05</td>
</tr>
<tr>
<td>4000</td>
<td>830</td>
<td>782</td>
<td>1.06</td>
</tr>
<tr>
<td>8000</td>
<td>1915</td>
<td>1795</td>
<td>1.06</td>
</tr>
<tr>
<td>16000</td>
<td>4325</td>
<td>4050</td>
<td>1.06</td>
</tr>
<tr>
<td>32000</td>
<td>9495</td>
<td>8911</td>
<td>1.06</td>
</tr>
<tr>
<td>64000</td>
<td>20385</td>
<td>19179</td>
<td>1.06</td>
</tr>
<tr>
<td>128000</td>
<td>44805</td>
<td>42332</td>
<td>1.05</td>
</tr>
<tr>
<td>256000</td>
<td>97355</td>
<td>92169</td>
<td>1.05</td>
</tr>
</tbody>
</table>

### Whole Quicksort

<table>
<thead>
<tr>
<th>ints sorted</th>
<th>real (ms)</th>
<th>sim (ms)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>45</td>
<td>43</td>
<td>1.05</td>
</tr>
<tr>
<td>250</td>
<td>90</td>
<td>82</td>
<td>1.10</td>
</tr>
<tr>
<td>500</td>
<td>185</td>
<td>170</td>
<td>1.09</td>
</tr>
<tr>
<td>1000</td>
<td>390</td>
<td>357</td>
<td>1.09</td>
</tr>
<tr>
<td>2000</td>
<td>775</td>
<td>716</td>
<td>1.08</td>
</tr>
<tr>
<td>4000</td>
<td>1590</td>
<td>1467</td>
<td>1.08</td>
</tr>
<tr>
<td>8000</td>
<td>2965</td>
<td>2736</td>
<td>1.08</td>
</tr>
<tr>
<td>16000</td>
<td>6090</td>
<td>5620</td>
<td>1.08</td>
</tr>
<tr>
<td>32000</td>
<td>11795</td>
<td>10888</td>
<td>1.08</td>
</tr>
<tr>
<td>64000</td>
<td>24260</td>
<td>22446</td>
<td>1.08</td>
</tr>
<tr>
<td>128000</td>
<td>50665</td>
<td>47265</td>
<td>1.07</td>
</tr>
<tr>
<td>256000</td>
<td>108165</td>
<td>101357</td>
<td>1.07</td>
</tr>
</tbody>
</table>
5.6. Measurements of Software Overhead for Message Passing

A simple message passing program was run on both the iPSC and its simulation in order to determine the amount of overhead that operating system software contributed to the message passing times. The program set up a sending process on node 0 of the iPSC and set up a receiving process on a node according to the command line arguments. The program was run with the receiving process on node 1, node 3, node 7, and node 15 in order to test message passing with 1 hop, 2 hops, 3 hops, and 4 hops, respectively. The sender sent a total of 1000 messages to the receiver. The receiver received each message and immediately sent a message of the same size back to the sender. The sender recorded the time just before it sent each message and the time just after it received the reply.

The program was then run as a simulation with zero time assessed for software overhead for the message passing routines; the only time taken for message passing was due to the hardware delays simulated by the UserSend() routine. The difference between the simulated and the real times is shown in the top graph of figure 5.2. This difference is assumed to be due entirely to the software overhead. It was not possible to determine how much of the difference, if any, was due to inaccuracies of the UserSend() routine. Based on these differences, a formula for the software overhead was produced, parameterized by the number of whole kilobyte packets, the number of bytes in the remaining partial packet, and the number of hops between source and destination. The overhead for a receive was made independent of the number of hops, with all of the dependence on hops being assessed to the send overhead, although there was no indication that the simulation would have been worse if the overhead had been divided exactly equally between the receive overhead and the send overhead. The difference between the real times and the simulated times with software overhead included is shown in the middle graph. The percentage difference is shown in the bottom graph.
Real iPSC communication times and simulated iPSC communication times with and without software overhead included.
5.7. Process Creation and Process Switching Times

Since the message passing routines on the iPSC are asynchronous to begin with, new processes did not have to be created to avoid suspending to wait for a reply. Process creation time for the iPSC, therefore, was not measured. Process switching also was not measured; the problems mentioned for measuring the V-System process switching times apply here as well.

5.8. Global Distribution Local Sort on the iPSC: Simulation vs. Reality

The results from running GDLS on the iPSC and on the iPSC simulation are presented in table 5.2, and graphed in figure 5.3. Simulations were found to vary little from run to run, probably because the iPSC is available to only a single user at a time, so GDLS was run for each combination of array size and sorter nodes only once. The simulation multiplies the cycle counts by 1.08 according to the results of the sequential quicksort test. The message passing routines accounted for software overhead according to results from the round trip message passing test.

The simulation generally predicts times that are smaller than reality. At small array sizes, some of the inaccuracy may be due to the 5 ms resolution of the clock. Simulation underestimates are especially apparent for small array sizes and more than two sorter nodes. One explanation for this trend is that the UserSend() routine does not account for cpu time that message forwarding requires of intermediate nodes. When GDLS is run on more than two nodes, some message traffic must pass through intermediate nodes. Some experimental data indicate that there can be a 15% loss of cpu time to processes running on intermediate nodes when there is heavy message traffic through those nodes. UserSend() needs to be able to simulate an interrupt of these intermediate nodes but ASIM, in its current state, is unable to model interrupts. Work is in progress to make simulating interrupts possible. For large array
sizes, the lack of modeled interrupts is less of a problem probably because the message passing is a smaller proportion of the whole program.
Table 5.2

Real and simulated times for GDLS on the iPSC. Accounting functions and UserScale were set as described in the text.

<table>
<thead>
<tr>
<th>ints</th>
<th>2 nodes</th>
<th>3 nodes</th>
<th>4 nodes</th>
<th>5 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>sim</td>
<td>error</td>
<td>real</td>
</tr>
<tr>
<td>125</td>
<td>0.03</td>
<td>0.03</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>250</td>
<td>0.06</td>
<td>0.06</td>
<td>0</td>
<td>0.05</td>
</tr>
<tr>
<td>500</td>
<td>0.12</td>
<td>0.11</td>
<td>-8</td>
<td>0.13</td>
</tr>
<tr>
<td>1000</td>
<td>0.22</td>
<td>0.21</td>
<td>-5</td>
<td>0.20</td>
</tr>
<tr>
<td>2000</td>
<td>0.43</td>
<td>0.42</td>
<td>-2</td>
<td>0.30</td>
</tr>
<tr>
<td>4000</td>
<td>0.85</td>
<td>0.83</td>
<td>-2</td>
<td>0.57</td>
</tr>
<tr>
<td>8000</td>
<td>1.62</td>
<td>1.59</td>
<td>-2</td>
<td>1.11</td>
</tr>
<tr>
<td>16000</td>
<td>3.17</td>
<td>3.12</td>
<td>-2</td>
<td>2.20</td>
</tr>
<tr>
<td>32000</td>
<td>6.46</td>
<td>6.23</td>
<td>-4</td>
<td>4.47</td>
</tr>
<tr>
<td>128000</td>
<td>28.12</td>
<td>27.00</td>
<td>-4</td>
<td>19.56</td>
</tr>
<tr>
<td>256000</td>
<td>60.31</td>
<td>58.37</td>
<td>-3</td>
<td>41.68</td>
</tr>
</tbody>
</table>

Global Distribution Local Sort on iPSC
accounting in the system calls according to round trip test
UserScale set to 1.08 as indicated by sequential quicksort test
Figure 5.3

Percentage simulation error for GDLS on the iPSC.
5.9. Distributed Quicksort on the iPSC: Simulation vs. Reality

The results from running the Distributed Quicksort on the iPSC and on the simulation of the iPSC are given in table 5.3, and graphed in figure 5.4. Real times varied little from run to run, so the program was run only once for most combinations of array sizes and branch processes. The simulation used accounting in the message passing routines as established by the round trip message passing test, and a UserScale of 1.08 as established by the sequential quicksort test.

The trends for this program are less easy to analyze. The simulation generally predicts times that are less than reality as it did for the GDLS program. There are some exceptions, however, and the exceptions seem to lack a pattern. It is possible that the exceptions are due to race conditions that can develop in this algorithm. If the root process is waiting for any branch to become available to sort a subarray, and the first branch to send a message to the root is the local branch, then the next subarray will be sorted locally. If, on the other hand, a remote branch is the first to send a message to the root, then the next subarray will be sorted remotely. Depending on the size of the subarray, the two cases will differ in the amount of time that they add to the overall execution. If the simulation message passing times do not exactly match the real message passing times, then it is possible that the simulation sorts a subarray locally when the real iPSC sorts the same subarray remotely, or vice versa. Although it was not checked, it should be possible for each branch process to record what subarray it sorted, in order to ascertain whether or not the simulation matched reality in this regard.
### Distributed Quicksort on iPSC

**accounting in the system calls according to round trip test**

*UserScale set to 1.08 as indicated by sequential quicksort test*

<table>
<thead>
<tr>
<th></th>
<th>0 branches</th>
<th>1 branch</th>
<th>2 branches</th>
<th>3 branches</th>
<th>4 branches</th>
<th>5 branches</th>
<th>6 branches</th>
<th>7 branches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ins</strong></td>
<td><strong>real</strong></td>
<td><strong>sim</strong></td>
<td><strong>error</strong></td>
<td><strong>real</strong></td>
<td><strong>sim</strong></td>
<td><strong>error</strong></td>
<td><strong>real</strong></td>
<td><strong>sim</strong></td>
</tr>
<tr>
<td>125</td>
<td>0.05</td>
<td>0.04</td>
<td>-0.20</td>
<td>0.03</td>
<td>0.03</td>
<td>0</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>250</td>
<td>0.10</td>
<td>0.09</td>
<td>-0.10</td>
<td>0.07</td>
<td>0.06</td>
<td>-0.14</td>
<td>0.07</td>
<td>0.05</td>
</tr>
<tr>
<td>500</td>
<td>0.20</td>
<td>0.18</td>
<td>-0.10</td>
<td>0.13</td>
<td>0.11</td>
<td>-0.15</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>1000</td>
<td>0.41</td>
<td>0.39</td>
<td>-0.5</td>
<td>0.27</td>
<td>0.24</td>
<td>-0.11</td>
<td>0.19</td>
<td>0.18</td>
</tr>
<tr>
<td>2000</td>
<td>0.86</td>
<td>0.81</td>
<td>-0.6</td>
<td>0.53</td>
<td>0.50</td>
<td>-0.6</td>
<td>0.41</td>
<td>0.37</td>
</tr>
<tr>
<td>4000</td>
<td>1.66</td>
<td>1.60</td>
<td>-4</td>
<td>0.98</td>
<td>1.02</td>
<td>4</td>
<td>0.69</td>
<td>0.70</td>
</tr>
<tr>
<td>8000</td>
<td>3.16</td>
<td>3.06</td>
<td>-3</td>
<td>1.81</td>
<td>2.05</td>
<td>13</td>
<td>1.47</td>
<td>1.55</td>
</tr>
<tr>
<td>16000</td>
<td>6.30</td>
<td>6.14</td>
<td>-3</td>
<td>3.48</td>
<td>3.83</td>
<td>10</td>
<td>2.63</td>
<td>2.73</td>
</tr>
<tr>
<td>32000</td>
<td>12.29</td>
<td>12.01</td>
<td>-2</td>
<td>6.73</td>
<td>6.68</td>
<td>-1</td>
<td>5.29</td>
<td>5.19</td>
</tr>
<tr>
<td>128000</td>
<td>53.44</td>
<td>52.68</td>
<td>-1</td>
<td>28.57</td>
<td>28.92</td>
<td>2</td>
<td>23.54</td>
<td>28.90</td>
</tr>
<tr>
<td>256000</td>
<td>117.80</td>
<td>116.41</td>
<td>-1</td>
<td>65.41</td>
<td>72.75</td>
<td>4</td>
<td>50.30</td>
<td>68.81</td>
</tr>
</tbody>
</table>

### Distributed Quicksort on iPSC

**continued**

<table>
<thead>
<tr>
<th><strong>ins</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
<th><strong>real</strong></th>
<th><strong>sim</strong></th>
<th><strong>error</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>0.04</td>
<td>0.03</td>
<td>-0.25</td>
<td>0.04</td>
<td>0.03</td>
<td>-0.25</td>
<td>0.04</td>
<td>0.03</td>
<td>-0.25</td>
<td>0.04</td>
<td>0.03</td>
<td>-0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>0.06</td>
<td>0.04</td>
<td>-0.33</td>
<td>0.06</td>
<td>0.04</td>
<td>-0.33</td>
<td>0.06</td>
<td>0.04</td>
<td>-0.33</td>
<td>0.06</td>
<td>0.05</td>
<td>-0.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>0.08</td>
<td>0.08</td>
<td>0</td>
<td>0.09</td>
<td>0.07</td>
<td>-0.22</td>
<td>0.09</td>
<td>0.07</td>
<td>-0.22</td>
<td>0.09</td>
<td>0.07</td>
<td>-0.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.16</td>
<td>0.16</td>
<td>0</td>
<td>0.16</td>
<td>0.16</td>
<td>0</td>
<td>0.17</td>
<td>0.16</td>
<td>-0.6</td>
<td>0.17</td>
<td>0.16</td>
<td>-0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>0.31</td>
<td>0.31</td>
<td>0</td>
<td>0.27</td>
<td>0.24</td>
<td>-0.11</td>
<td>0.27</td>
<td>0.25</td>
<td>-0.7</td>
<td>0.32</td>
<td>0.24</td>
<td>-0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>0.59</td>
<td>0.57</td>
<td>-3</td>
<td>0.58</td>
<td>0.57</td>
<td>-2</td>
<td>0.59</td>
<td>0.57</td>
<td>-3</td>
<td>0.59</td>
<td>0.57</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>1.21</td>
<td>1.19</td>
<td>-2</td>
<td>1.21</td>
<td>1.19</td>
<td>-2</td>
<td>1.21</td>
<td>1.19</td>
<td>-2</td>
<td>1.21</td>
<td>1.19</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16000</td>
<td>2.05</td>
<td>1.98</td>
<td>-3</td>
<td>1.97</td>
<td>1.89</td>
<td>-4</td>
<td>1.92</td>
<td>1.84</td>
<td>-4</td>
<td>1.88</td>
<td>1.84</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32000</td>
<td>5.30</td>
<td>5.19</td>
<td>-2</td>
<td>5.30</td>
<td>5.19</td>
<td>-2</td>
<td>5.30</td>
<td>5.19</td>
<td>-2</td>
<td>5.30</td>
<td>5.19</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128000</td>
<td>19.30</td>
<td>19.79</td>
<td>2</td>
<td>19.31</td>
<td>18.75</td>
<td>-3</td>
<td>19.31</td>
<td>18.75</td>
<td>-3</td>
<td>19.31</td>
<td>18.75</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256000</td>
<td>46.39</td>
<td>45.07</td>
<td>-5</td>
<td>46.23</td>
<td>45.00</td>
<td>-3</td>
<td>46.49</td>
<td>45.00</td>
<td>-3</td>
<td>46.77</td>
<td>45.00</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.4

Percentage simulation error for DQ on the iPSC.
CHAPTER 6

Implications of this Work for the RPPT

The following is a compilation of opinions concerning the RPPT that are based on the research experience described in this thesis. There are both good and bad aspects of the present work.

6.1. Flaws and Difficulties Peculiar to This Work

Some flaws and difficulties are peculiar to this effort to validate the RPPT and are a detriment only to the accuracy of these results but not necessarily to the usefulness of the RPPT for other work. One such difficulty is background traffic on the Ethernet. Background traffic reduces the reliability of the measurements of operating system software overhead for message passing. Background traffic also interferes with the execution of the sorting programs. While the effects of background traffic may be minor, it should be possible to obtain a better match between the simulated times and the measured times on a V-System with its own private local area network. Such a system is currently being assembled, and some simple validations for the RPPT should be run on it. On this system it would also be possible to run the V-System using its old message passing algorithms, thereby avoiding difficulties in modeling the blast protocol.

Another kind of difficulty is exemplified by the problems in measuring V-System process switching times and in determining the actual behavior of the V-System and iPSC message passing. In these cases the real computer systems devote far more software to these functions than it is desirable to include in the RPPT. Furthermore, for proprietary systems such as the iPSC, the documentation for this software is difficult to obtain. Since most real machines are likely to use complicated software, this may limit the accuracy that the RPPT
will achieve in modeling them. It will not, however, limit the usefulness of the RPPT for running detailed simulations of hypothetical machines and other machines that can be specified completely.

Finally, the use of excessively complicated programs, such as the Distributed Quicksort that was implemented on the V-System, is probably not a good method for validating the RPPT. There are too many parts of it that might be the source of the discrepancy between the real and simulated times. The other sorting programs are less complicated, but for them also it is difficult to identify the sources of the timing discrepancies. There was probably too much emphasis on writing fast sorting programs for this work. More emphasis should be on simplicity for the sake of validation.

6.2. Positive Aspects of the Methodology Used in This Work

The best approach for validating the RPPT is probably to start with the very simplest of programs. Purely sequential programs and programs that only pass messages in a simple way, such as were used to establish profiling scale factors and message passing software overheads, are good examples. More programs like this would be good. Once the simple programs are all being simulated accurately they should be combined as simply as possible as parts of a bigger program.

Validation using nonnumeric applications is important. For many numeric applications, the floating point operations can use more time than any of the other operations. In some cases it is possible that if the right amount of time is attributed to floating point operations, then the simulated time will be close to the actual time even if the rest of the operations are assigned the wrong times.
6.3. Suggestions for Improving the RPPT

The RPPT could be improved by the addition of some features. The RPPT at present runs all processes at the same priority, which is a weakness since it is often advantageous to assign different priorities to processes. (GDLS on the V-System is an example of this.) To implement process priorities, it may be possible to use time slicing in conjunction with minor modifications to the Concurrent C driver.

To model effects like the passage of messages through intermediate nodes of the iPSC, it must be possible to model interrupts. It may be convenient to use CSIM semaphores to represent each processor interrupt level. Such a scheme would have the additional advantage of simplifying the present accounting mechanism.

In order to make it possible to model communication strategies such as the blast protocol for the V-System, it should be possible to substitute new communication routines into ASIM. The complexities of the blast protocol occur at the operating system level. Since ASIM is the operating system level for RPPT simulations, changes should be made to ASIM rather than UserSend(), for example. Presently, ASIM routines such as AsimSendMessage() make calls to routines provided by the modeler that determine how much time to delay for software overhead. The changes needed to simulate the blast protocol are more extensive than altering the formula to calculate overhead. What is needed is to substitute a new AsimSendMessage(), and the other message passing routines as well, that actually model the sending and receiving of blasts. This should be possible without changing other parts of the RPPT, but care will be needed because of the complex interdependencies of the RPPT data structures.

A new approach may be needed to replace cross profiling in cases where the correspondence between simulation basic blocks and real basic blocks is poor. This would
have been the case for sorting long integers using the 80286 processor. Even for sorting short integers on the iPSC some hand corrections had to be made to the cross profiling because the basic blocks did not match exactly. If it were possible to define a new 32 bit data type for the simulations, and the operations for that new type, it might be possible to make the simulation handle 32 bit data in the same way that the iPSC processors do. Writing simulations as C++ programs might be a good way to provide for the creation of such new data types and operations.

Ideally, the experimenter would have complete control over the software on the machine being studied. In the sequential segments of code, the goal is to obtain a complete match between basic blocks on the real parallel machine and the basic blocks in the simulation code. It should be possible to achieve this by writing the real and simulation versions of a program in assembly code, or by constructing compilers that generate identical basic block structures. There are at least two ways to deal with operating system software, either to replace the parallel computer’s operating system routines with ones that match those in the RPPT or to do the reverse, as suggested above for the blast protocol. The RPPT must be made to simulate interrupts and DMA, since operating system software interacts with hardware extensively by these means. If the operating system routines can be made to match at the basic block level then profiling could be used to measure the time that they require.

6.4. Successful Features of the RPPT

There are many successful features of the RPPT. The efficiency of execution driven simulation is evident in the fact that all of the simulations for this thesis could be run in a total of only a few hours. Cross profiling is quite accurate, especially when the basic blocks match well. The simulations as a whole are fairly accurate, and only approach errors as great as 50% when the times are very short.
The RPPT proved to be useful for analyzing the behavior of parallel programs. On one occasion a program for the iPSC was deadlocking. When program tracing statements were added to the code, the added statements changed the timing enough that the deadlock did not occur. A simulation of the program also reached a deadlock. The addition of program tracing statements had no effect, of course, so it was possible to identify and correct the problem.
References


