

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.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1338748

**Efficient simulation and utilization of a parallel digital signal
processing architecture**

Foundoulis, William James, M.S.

Rice University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

RICE UNIVERSITY

EFFICIENT SIMULATION AND UTILIZATION OF A PARALLEL
DIGITAL SIGNAL PROCESSING ARCHITECTURE

by

WILLIAM JAMES FOUNDOULIS

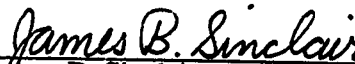
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

MASTER OF SCIENCE

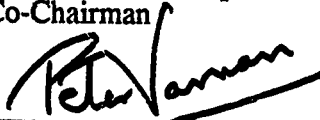
APPROVED, THESIS COMMITTEE:



J. Robert Jump, Professor of
Electrical and Computer Engineering,
Co-Chairman



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



Peter J. Varman, Assistant Professor of
Electrical and Computer Engineering

Houston, Texas
May, 1989

**Efficient Simulation and Utilization of a Parallel
Digital Signal Processing Architecture
by
William James Foundoulis**

Abstract

In this study we discuss the development and validation of an efficient and accurate execution-driven simulation of the Texas Instruments Odyssey System, a parallel configuration of digital signal processors. We also evaluate the performance of a high-level parallel programming interface, Odyssey Concurrent C, designed to effectively utilize the parallelism available in the Odyssey architecture. Parallel versions of three dissimilar algorithms—merge sort, 2-dimensional convolution, and successive over-relaxation—have been run on both the Odyssey and the simulator. Quantitative differences between performance results obtained on the Odyssey and those predicted by simulation are enumerated, and shown to validate the accuracy of the execution-driven approach. The simulation is also shown to be efficient relative to the degree of accuracy obtainable. Finally, the Odyssey Concurrent C utilities are shown to provide a flexible and effective mechanism for managing parallelism in the Odyssey environment.

Acknowledgements

I wish to thank the many faculty, friends, and family who have helped to make this effort possible. Drs. Jump and Sinclair, my committee directors, have always been available for questions/comments/suggestions, and were exceptionally patient when work was progressing very slowly. Dr. Peter Varman was kind enough to lighten my grading load during this busy semester, as well as serve on my committee. And a special thanks to Dr. Joe Cavallaro, who has provided keen insights into the unique pressures facing a new faculty member.

I am also indebted to the co-workers with whom I have shared this experience. The invaluable assistance of Rick Covington and Sridhar Madala was critical to this work, and was always given freely despite their own pressing deadlines. Also, Trinanjan Chatterjee, Bill Dawkins, Vijay Debbad, Kshitij Doshi, Sandhya Dwarkadas, Steve Ingels, Grant Lauderdale, Varun Mehta, Rajat Mukherjee, and Nigel Waites have been the source of innumerable enlightening discussions during my stay here.

Finally, I would like to express my heartfelt thanks to my parents, whose support through many difficult years has been unwavering.

Table of Contents

1	Overview	1
2	Odyssey System Architecture	3
2.1	Introduction	3
2.2	Processors	3
2.3	Memory	5
2.4	Interconnection Network	6
3	Odyssey Concurrent C	7
3.1	Introduction	7
3.2	Design Factors	8
3.3	OCC Primitives	9
3.4	Programmer's View	12
3.5	Performance Measurements	12
3.6	Summary	15
4	The Rice Parallel Processing Testbed	16
4.1	Introduction	16
4.2	Concurrent C	17
4.3	CSIM	18
4.4	ASIM and ASIMP	18
4.5	Profiling	19
4.6	Odyssey Architecture Modeling	22

5	Algorithms	25
5.1	Introduction	25
5.2	Merge Sort	26
5.3	2-Dimensional Convolution	28
5.4	Successive Over-Relaxation	30
6	Results	37
6.1	Introduction	37
6.2	Sources of Error	37
6.3	Validation	41
6.4	Prediction	54
7	Conclusions	61
7.1	Summary of Results	61
7.2	Future Work	63
	References	65

Chapter 1

Overview

With the recent emphasis on parallel processing and the availability of true concurrent machines, the need has arisen for methodologies concerned with using, analyzing, and evaluating parallel architectures. This thesis deals with the effective use of an existing parallel processing architecture as well as the efficient and accurate simulation and performance evaluation of that architecture.

The architecture discussed here is the *Texas Instruments Odyssey System*, a parallel machine consisting of digital signal processing chips interconnected through a shared-bus communications network. A high-level parallel programming interface, *Odyssey Concurrent C*, has been developed in order to more easily program the Odyssey. The simulation of the Odyssey runs on the *Rice Parallel Processing Testbed*, a software tool used to simulate the execution of concurrent algorithms running on parallel architectures. Specifically, the contributions of this work include the following:

- 1) The use, testing, debugging, and performance evaluation of a high-level parallel programming interface designed to make effective use of the architectural features available in the Odyssey architecture.
- 2) The development of an efficient and accurate package for simulating and evaluating the performance of the Odyssey system.
- 3) The implementation of several concurrent algorithms on the Odyssey, including a validation of the efficiency and accuracy of the simulation package in 2), and an

analysis of the effectiveness of the Odyssey Concurrent C routines in 1).

The remainder of this thesis is organized as follows. Chapter 2 describes the Odyssey system architecture. Chapter 3 is a discussion of the Odyssey Concurrent C user interface which was used to program the Odyssey, and the results of some basic performance measurements. Chapter 4 is a brief overview of the Rice Parallel Processing Testbed used to simulate the Odyssey. Especially important here are the final two sections discussing profiling and architecture modeling. Chapter 5 describes the parallel algorithms which were implemented on the Odyssey. Chapter 6 is a compilation of the results of running and simulating these algorithms on the Odyssey, as well as a discussion of the significance of these results. Finally, Chapter 7 summarizes the important results of the thesis and discusses some possible extensions to this work.

Chapter 2

Odyssey System Architecture

2.1 Introduction

The *Texas Instruments Odyssey System* is a parallel configuration of digital signal processing chips [1] which runs in a *NuBus*-based environment such as exists on the *Texas Instruments Explorer*.¹ Figure 2.1 shows the important architectural features of a single board in an Odyssey system. An Odyssey board consists of four processor modules, each containing a TMS32020 or TMS320C25 processor chip, several types and speeds of memory, and bus interface circuitry. Processors communicate through a shared-bus interconnection network which allows every processor access to all off-chip memory in the system. An Odyssey system may contain anywhere from 1 to 16 boards for a maximum of 64 processors. Each of these architectural components is discussed in more detail below. For full documentation on the Odyssey System, see [2].

2.2 Processors

The processors we are currently using on the Odyssey are *Texas Instruments TMS320C25* digital signal processors [3]. The TMS320C25 uses a modified Harvard architecture, meaning program and data spaces are distinct and accessed through separate buses. The fundamental advantage of the Harvard architecture is that program (code) and data (operand) fetches can occur simultaneously.

¹ Odyssey, NuBus, and Explorer are trademarks of Texas Instruments Inc.

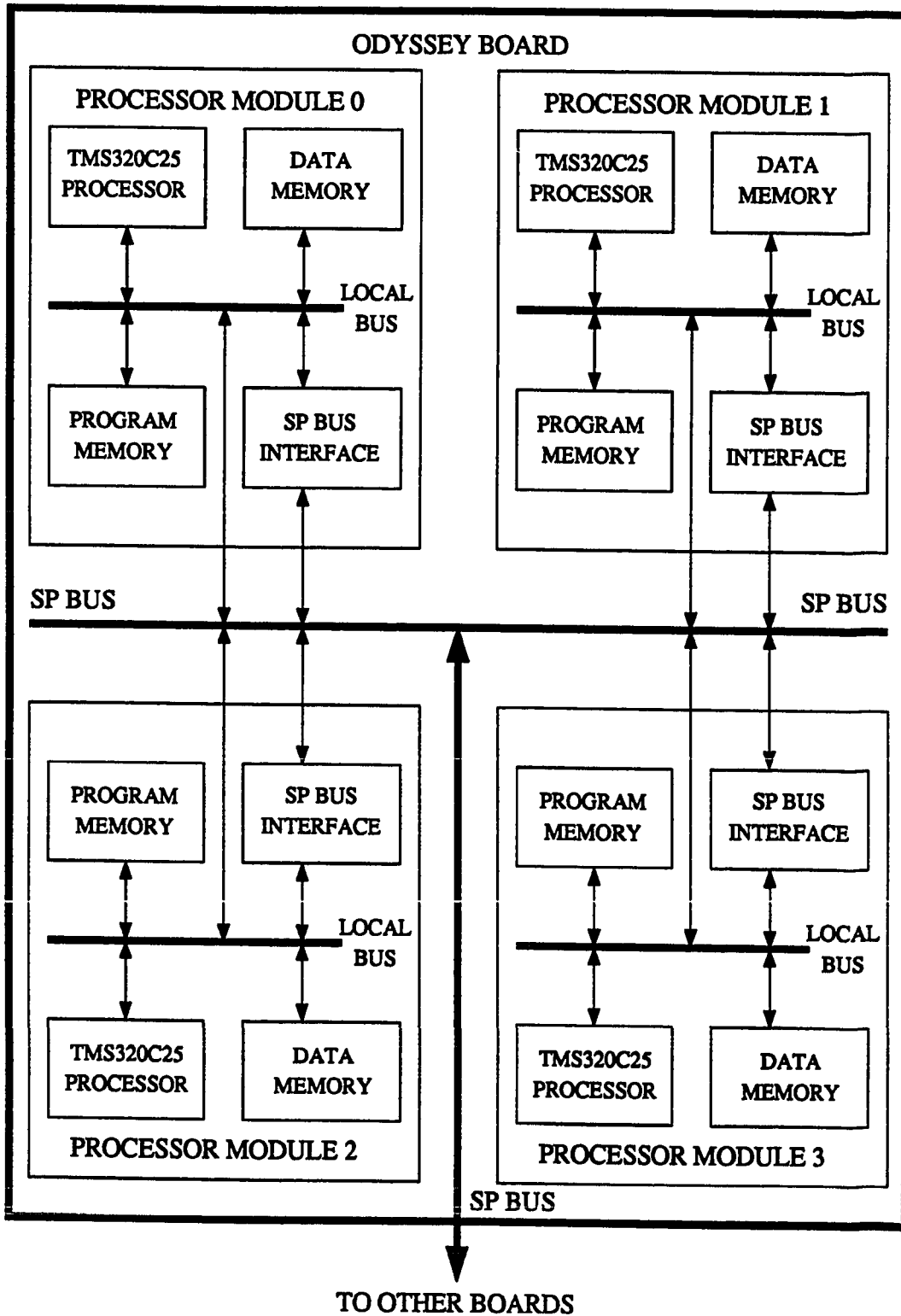


Figure 2.1: Odyssey board system architecture.

The TMS320C25 was designed primarily for low-level signal and image processing applications [4]. Consequently, many special hardware and instruction set features are provided explicitly for this purpose. A hardware instruction pipeline is used to speed program execution. This, combined with the Harvard architecture, allows many instructions to be performed in a single cycle. Since the processors on the Odyssey are clocked at 5 MHz, execution speeds near 5 MIPS can be achieved. The architecture also supports a mode whereby instructions can be repeated up to 256 times. The instruction pipeline and repeat mode features allow very efficient multiply/accumulate and block data move operations to be performed. Efficient block data moves are critically important in an architecture where several different address spaces are available.

A C compiler (preprocessor, parser, code generator, and assembler/linker) is also available for the TMS320C25. With minor exceptions, it implements a standard version of C. For full documentation on the TMS320C25 C compiler, see [5, 6].

2.3 Memory

Several different memory spaces are addressable in the TMS320C25 processor architecture. Those which exist in an Odyssey board configuration are discussed here. 544 16-bit words of *on-chip* (zero wait-state) data memory are available on each processor. This memory can be considered a large register set, and most instructions which use data residing in on-chip memory can be executed in a single cycle. An additional 64K words of *local* (off-chip) data memory is also available to each processor in the system. This memory has one wait-state, and instructions whose data are in local memory generally

take 2–3 instruction cycles to complete. The *program* space consists of 8K words of zero wait-state memory per processor. All code running on each processor is contained in this memory space. The final memory class of concern here is *global* memory. Memory is said to be global relative to a given processor if it is in the local data or program address space of another processor. The Odyssey has hardware enabling all processors to access global memory.

2.4 Interconnection Network

Processors in the Odyssey system are interconnected via the *Signal Processing (SP) bus*, which is a modification of the proposed IEEE NuBus standard [7]. The SP bus is a shared-bus network which allows any off-chip memory location in the entire collection of system boards to be accessed by any processor in the system. SP bus arbitration is fair in the sense that a request for the SP bus from a given processor will be serviced before any later request from some other processor. Hence, no bus starvation is possible, and repeated simultaneous requests for bus service from multiple processors will be serviced in a round-robin fashion.

Special hardware features are provided in each processor module allowing access to the SP bus. A single global memory access (16-bit word) over the SP bus takes a minimum of 1.6 μ s not including software overhead. The total time necessary to access a single word including software overhead is about 4 μ s. However, we will see in Chapter 3 that by transferring blocks of data rather than one word at a time global memory may be accessed at the rate of about 2 μ s per word.

Chapter 3

Odyssey Concurrent C

3.1 Introduction

Programming parallel algorithms for the Odyssey system has in the past required a detailed knowledge of the low-level hardware features available in the architecture. It also required at least some assembly language programming to properly manipulate these hardware features [8].

As a result of the availability of a C compiler for the Odyssey processors, and the desire to create a less intimidating programming environment, we have implemented a high-level parallel programming interface for the Odyssey system. This environment, *Odyssey Concurrent C (OCC)*, insulates the programmer from most of the architectural details of the system. It also allows relatively quick algorithm design and prototyping since all source code can be written in C.

Facilities are provided in OCC for interprocessor communication and synchronization. Message passing is the basic communications model, and various semaphore handling operations are also available. These basic operations allow enough flexibility so that virtually any coarse-grained parallel algorithm can be programmed effectively using the OCC primitives.

3.2 Design Factors

In developing OCC, the most critical design decision involved the choice between providing a message-based communications facility or implementing a shared-memory model for the Odyssey system. The decision was made based primarily on the following observations:

- 1) Although all memory is accessible from all processors, the Odyssey belongs to the class of shared-memory multiprocessors known as *Non-Uniform-Memory-Access* systems. This is because global memory accesses are several times slower than local accesses. Consequently, the indiscriminate use of global memory can be very inefficient.
- 2) Processors cannot utilize global memory in the same fashion as local memory. Any global data and/or program code must first be read into local memory to be used. This makes the design of a traditional shared-memory model difficult.
- 3) Instruction pipelining, along with special block data move and repeat mode capabilities, make moving blocks of data within local memory much more efficient than moving one word at a time. Additional hardware built into each processor module allows similarly efficient block data moves over the SP bus. It follows that since communication among the processors is most efficiently handled through the movement of blocks of data over the SP bus, a message-passing protocol is the most natural model to implement.
- 4) Since the SP bus is shared among all processors it can easily become a bottleneck, precluding the efficient execution of very fine-grained algorithms on the system.

Hence, architectural considerations limit us to using algorithms exhibiting medium- to coarse-grained communication patterns. Interprocessor communication in algorithms of this type tends to occur at relatively infrequent intervals, but involves transfers of large amounts of data. This again lends itself to a message-based communications facility as the most efficient means of using the architecture.

For these reasons, message passing was chosen and implemented in OCC as the fundamental mechanism for interprocessor communication. The resulting system presents the user with a high-level interface with which parallel algorithms can be quickly and effectively programmed for the Odyssey. The available message-passing and synchronization facilities are discussed in the following sections. For full user documentation on OCC, see [9, 10].

3.3 OCC Primitives

Both synchronous and asynchronous message-passing facilities are available in OCC. Synchronous primitives consist of *SendRequest()*, *ReceiveRequest()*, and *Reply()*. *SendRequest(id,count,sbufptr,rbufptr,reqtype)* sends a request message of *count* words to the processor identified by *id*. The request is copied from the buffer pointed to by *sbufptr*. The reply returned by the receiver is copied into the buffer pointed to by *rbufptr*. The request sent is of type *reqtype* and the call returns with the size of the reply message in words. The sender blocks until it receives a reply from the receiver, so no buffering of the request is necessary.

ReceiveRequest(block,bufptr,reqtype,typeptr,cntptr) receives the next request of type *reqtype*. The request is copied into the buffer pointed to by *bufptr*. The *reqtype* parameter can be used to selectively receive requests. For example, the type could be the sender's processor identification number, or it could correspond to the current stage number of the algorithm. A receive may be *blocking* or *non-blocking*. If *block* is 0, the receive will return immediately if no requests of the proper type are available. If *block* is 1, the receive will wait indefinitely for the next request. The type of request actually received is placed in *&typeptr*, while the number of words received is left in *&cntptr*. The call returns the identification number of the sender.

Reply(id,count,bufptr) sends a reply message of *count* words to processor *id*. The reply message is copied from the buffer pointed to by *bufptr*. A processor that is replied to should be blocked in a *SendRequest()* and waiting for a reply from the receiver. These synchronous routines can be used to implement request-reply transactions in a client-server application or any time processors must exchange data. They are also effectively used in algorithms where processors reach communication points at approximately the same time.

Asynchronous message passing is accomplished with the *SendMessage()* and *ReceiveMessage()* primitives. Buffers on each processor are statically allocated at compile time and determine the maximum message length and maximum allowable number of outstanding messages supported by the system. *SendMessage(id,count,bufptr,msgtype)* sends a *count* word message of type *msgtype* to processor *id*. The message is copied from the location pointed to by *bufptr*. If a free buffer exists at the receiver, *SendMes-*

sage() will buffer the message and return; otherwise, the sender blocks until a free buffer becomes available. No reply or acknowledgement is given when a message is actually received (as opposed to just being buffered) by the intended processor, and deadlock-free operation is the responsibility of the algorithm designer.

ReceiveMessage(block,bufptr,msgtype,cntptr,typeptr,idptr) receives the next available message of type *msgtype*. The *msgtype* parameter is used the same way here as in the synchronous primitives. The message is copied into the buffer pointed to by *bufptr*. The receive may be *blocking* or *non-blocking*, again similar to the synchronous routines. The number of words received, type of message received, and identification number of the sender are returned in *&cntptr*, *&typeptr*, and *&idptr*, respectively. Asynchronous message passing is less space efficient than synchronous, but can be useful for algorithms where communication points are reached at widely varying intervals on the various processors in the system.

Message passing inherently involves a synchronization operation. However, other primitives are also available in OCC to handle pure synchronization operations which would be awkward to accomplish with message passing alone. These facilities are implemented with *semaphores* manipulated by Dijkstra's classic *P()* and *V()* semaphore operations. A *V(id,semptr)* operation increments (by one) the semaphore located at address *semptr* in the local memory space of processor number *id*. A *P(id,semptr)* decrements the specified semaphore by one if it is positive; otherwise, the calling processor suspends until some other processor releases the semaphore with a *V()* operation. Semaphores are used to control access to one or more resources through proper initialization and use.

For instance, semaphores are extensively used in the OCC message-passing routines to control access to the SP bus as well as OCC data structures. Semaphores can also be used to force synchronization points in a program if necessary.

3.4 Programmer's View

The OCC programmer's view of the Odyssey system consists of the following:

- 1) several processors, each of which may be running independent copies of the same code or completely different algorithms;
- 2) a maximum of one process per available processor (the lack of a true operating system on the board disallows the use of such things as multiprogramming, time slicing, and dynamic process creation/migration);
- 3) a set of message-passing and synchronization facilities as discussed above; and
- 4) a standard C language interface, subject to the constraints of the architecture (such as memory size limitations) and the TMS320C25 C compiler (such as available data type sizes).

In general, OCC presents the user with a relatively simple view of the Odyssey system, handling the more complex implementation details internally.

3.5 Performance Measurements

Some basic performance measurements of the OCC message-passing primitives are given in Figures 3.1–3.3. Figure 3.1 plots the total time necessary to send a synchronous message. This time includes the *SendRequest()* to a remote processor (that is blocked

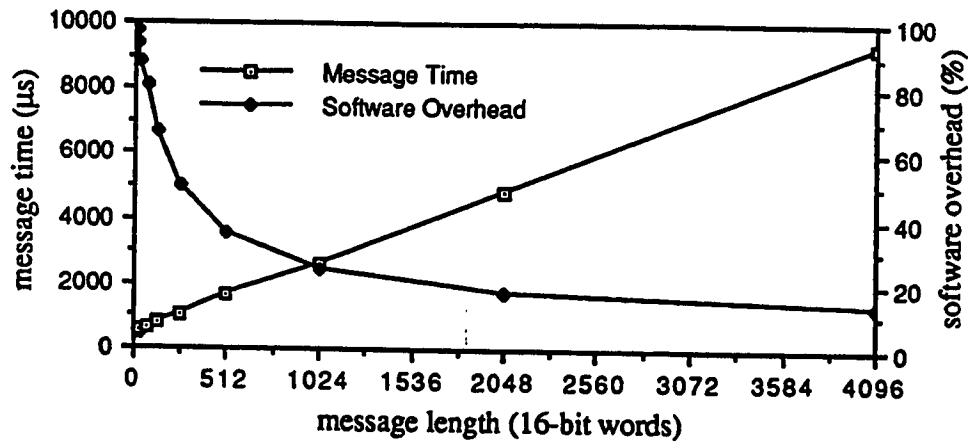


Figure 3.1: Synchronous Message Passing Time and Software Overhead vs. Message Length.

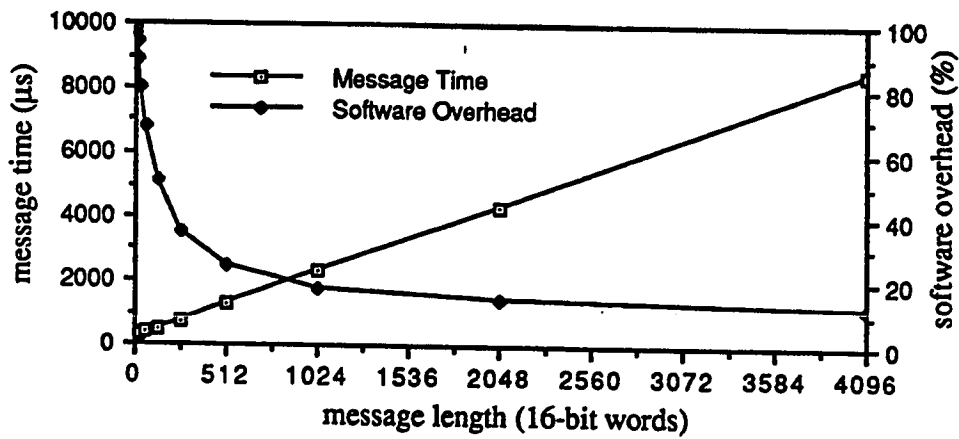


Figure 3.2: Asynchronous Message Passing Time and Software Overhead vs. Message Length.

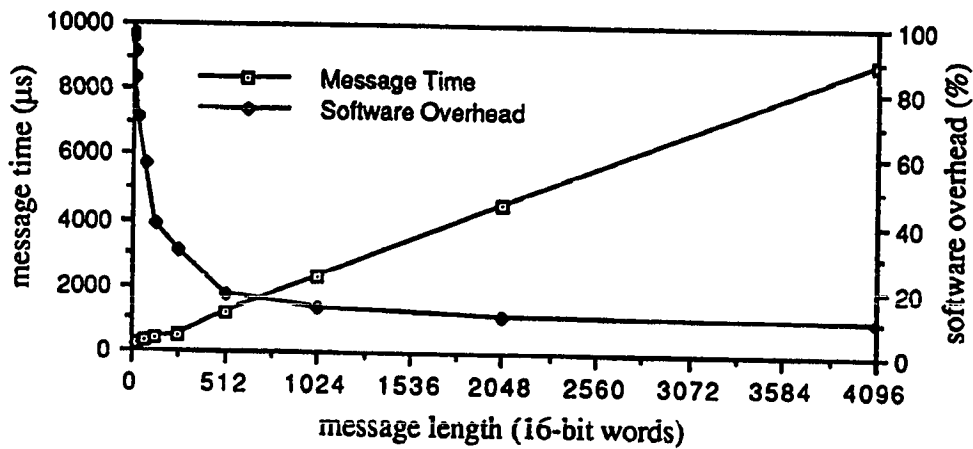


Figure 3.3: Message Passing Time and Overhead (Assembly-Coded Routines).

on a *ReceiveRequest()*), the message transfer to that remote processor, and a zero-length *Reply()* from the receiver in acknowledgement to the sender. Also plotted in Figure 3.1 is the percentage of the total message passing time attributable to software overhead. This includes everything except SP bus protocol overhead and transmission time.

The plot shows that software overhead is fixed at approximately 500 μ s, regardless of message length. The SP bus transfer time works out to just under 2 μ s per 16-bit word of the message. For reasonably large (i.e., >512-word) messages, total message passing time is near 2 μ s/word. This is in comparison to the aforementioned 4 μ s needed to access a single word over the SP bus using hand-written assembly code.

Figure 3.2 is similar to Figure 3.1 except asynchronous message passing is used. In this case, the total time consists only of the *SendMessage()* call, which returns after transferring the message to the receiver's buffer space. Software overhead here is 260 μ s, or about half that of the synchronous case. From the sender's viewpoint, asynchronous message passing can be faster than synchronous because the sender need not block for the duration of the transfer. However, the total time from the send to the end of the receive will actually take longer in the asynchronous case since an extra buffering step is necessary.

Figure 3.3 plots message-passing time using some synchronous block transfer routines, hand-coded in assembly language, from another source. Software overhead is 150 μ s, or about a third that for the synchronous OCC utilities. These routines implement a much simpler message-passing protocol than the OCC utilities and so are not directly comparable. If exactly comparable routines were written in assembly language, software

overhead would probably be about half that of the C compiled versions. In any case, the important point to note is that for reasonably large messages, differences in overall message-passing time using C-coded and assembly-coded primitives is quite small. Given the advantages of using C, it is apparent that little would have been gained by programming the OCC routines in assembly.

3.6 Summary

The applicability of the OCC utilities is dependent upon several factors including architectural considerations, flexibility, ease of use, and the efficiency resulting from their use. The power of the Odyssey processors and the fundamental limitations of the SP bus interconnection network result in the system being most effectively used with algorithms exhibiting relatively coarse-grained communication patterns. In these algorithms, processors perform significant amounts of computation on a large amount of data and then exchange blocks of data among themselves. Transferring blocks of data between processors is inherently a message-passing operation, and hence, algorithms such as this are easily programmed using the OCC routines.

From an efficiency standpoint, we have seen that global block data transfers of sufficient length are about twice as fast on a per word basis as accessing a single word over the communications network. This again leads us to conclude that message passing is the obvious choice for this architecture, and that the algorithms most amenable to running on the Odyssey will be those which can be designed so that messages are of relatively large size.

Chapter 4

The Rice Parallel Processing Testbed

4.1 Introduction

A major result of this work is the design and validation of an efficient and accurate simulation of the Odyssey system. The simulator was implemented using the *Rice Parallel Processing Testbed (RPPT)*, a large software project running under UNIX² on Sun Workstations.³ The RPPT is based on a relatively new simulation technique called *execution-driven* simulation [11, 12, 13]. Execution-driven simulation is an efficient method of simulation whereby a real concurrent program is run directly on the simulator using the simulating processor's own compiler and instruction set. This is a more flexible and accurate mechanism for simulation than would exist in a simulator based upon *statistically-derived* workloads. It is also much more efficient than either a *trace-driven* approach or an *instruction-level* simulation, which must emulate the detailed effects of the simulated system's instruction set. In summary, the RPPT attempts to simulate with a high degree of efficiency while at the same time generating performance predictions that are only slightly less accurate than a costly emulation of the target architecture.

The major components of the RPPT consist of the following.

- 1) Concurrent C: a pseudo-concurrent programming language.
- 2) CSIM: a discrete-event simulator.

² UNIX is a trademark of AT&T Bell Laboratories.

³ Sun Workstation is a trademark of Sun Microsystems Inc.

- 3) ASIM: a library of modified Concurrent C routines which account for simulation time.
- 4) ASIMP: a preprocessor which replaces Concurrent C calls with their corresponding ASIM calls.
- 5) A set of assembly language timing profilers (TPROF) which insert instructions into assembly code to increment a globally available counter. Also of importance here is a methodology (cross-profiling) for inserting profiled timings from the simulated system into the simulator's code.
- 6) A library of architecture models which account for the transmission time of messages over the simulated interconnection network between processors.

Each of these subsystems is discussed in greater detail below, with emphasis placed on those aspects specific to simulating the Odyssey.

4.2 Concurrent C

Concurrent C is a parallel programming language consisting of a collection of C utility routines that allow process management, synchronization, and interprocessor communication [14]. Concurrent C processes may be created, forked, activated, suspended, and joined. Pure synchronization is by means of the classic semaphore operations while interprocessor communication is accomplished with synchronous and asynchronous message passing. The syntax and logical properties of the Odyssey Concurrent C routines discussed in Chapter 3 were derived from a subset of the Concurrent C facilities.

It is important to note that a Concurrent C program runs in a uniprocessor environment as a single UNIX process. This “pseudo-concurrency” maintains the logical properties

of a true concurrent program (such as synchronization) but does not take into account the execution time or interconnection network delays which would exist in an actual parallel environment.

4.3 CSIM

CSIM (C SIMulation package) is a process-oriented discrete-event simulator [15, 16]. It is built on top of Concurrent C and primarily adds event queue capabilities to the basic Concurrent C facilities. CSIM provides a set of predefined types and a collection of utility routines which manipulate these types. The available types include *semaphores*, *queues*, *state variables*, and *conditions*. A flexible statistics collection facility is also included.

CSIM is the mechanism through which simulation time is consistently maintained in an RPPT application. CSIM data structures and calls to CSIM routines are also the means by which the architectural model of the simulated system is implemented.

4.4 ASIM and ASIMP

ASIM (Architecture SIMulator) consists of modified versions of the Concurrent C procedures to which CSIM data structures and calls have been added to account for execution time in an RPPT simulation.

ASIMP (Architecture SIMulator Preprocessor) is a lexical analyzer, run as a pre-processor to a Concurrent C program, which changes all Concurrent C calls in the program to their corresponding ASIM calls. In this way, a basic Concurrent C program is transformed automatically into a form suitable for simulation.

4.5 Profiling

4.5.1 TPROF

TPROF (Timing PROFiler) is a collection of instruction set profilers, each matched to a specific processor, which analyze the *basic block* structure of an assembly language program and insert instructions into each block to increment a counter by the estimated execution time of that basic block. Basic blocks are sections of code which are guaranteed to execute sequentially on a single processor. A basic block begins with the first instruction following either a label or a branch, and ends with either the last instruction before a label or a branch. The fundamental advantage of basic block profiling is that a counter tracking cumulative simulation time need only be incremented at basic block intervals rather than at each instruction, thereby eliminating much simulation overhead.

Estimated execution times are derived from timing tables generally published with the documentation for a chip. For the TMS320C25, timing tables are available from which cycle counts may be determined for any instruction executing in any memory space available to the processor in the Odyssey system. The version of TPROF specific to the TMS320C25 uses these tables to calculate the execution time for each basic block it encounters.

4.5.2 Cross-Profiling

The various versions of TPROF may be used as standalone profilers. However, this necessitates running the RPPT on a processor with the same instruction set as that of the simulated system. To avoid this limitation (which would preclude the use of the RPPT

in simulating the Odyssey), a method called *cross-profiling* has been developed whereby the profiled basic block timings from the simulated processor's assembly code may be inserted into corresponding basic blocks in the simulating processor's assembly code [17, 13]. The major requirement is the availability of a C compiler for both processors, each of which generates code for the major C language constructs with similar basic block structure. Fortunately, this is typically the case for compilers that do not perform much optimization.

The processors of interest here are the TI TMS320C25 (the simulation *target* architecture) and the Motorola 68020 (the RPPT *host* Sun). Cross-profiling is performed by automatically inserting markers into original C code which will "fall through" to the assembly code level during compilation. These markers are placed at locations in the source code which define the overall block (as opposed to basic block) structure of the program. For example, markers are placed before and after *for*, *while*, and *if* statements, before blocks of sequentially executed code, etc. After compiling to the assembly-code level, corresponding profilers are run on the files for the target and host. The host profiler simply marks the basic blocks, while the target profiler both marks blocks and maintains a table of basic block times. Finally, using corresponding basic block boundaries and the placement of the markers from the original C source code, the basic blocks of the target and host are matched and timing information derived for the target is inserted into the corresponding blocks of the host. In this manner the RPPT simulation, which runs on a 68020 processor, uses the basic block timings generated by the TMS320C25 profiler.

The basic block matching between compiled TMS320C25 and 68020 code is not

perfect. Errors may arise from actual basic block mismatches, or because a basic block on the target has no correspondence with a basic block on the host. We shall see in Chapter 6, however, that for any reasonably straightforward non-trivial C source code, the error introduced by cross-profiling is quite small.

Code for which the original C source is either nonexistent or unavailable cannot be cross-profiled. For our purposes, this primarily applies to the software floating-point routines provided with the TMS320C25 C compiler. In order to account for the execution of these routines in a simulation, measurements must be made to arrive at average execution times for these functions. These times can then be inserted directly into the simulation code at the profiling stage.

Tables 4.1–4.3 enumerate the average times which were observed. Most of these execution times were relatively consistent (i.e., not data dependent). Addition and subtraction, however, exhibited great variability, and we shall see later that this causes additional inaccuracies to arise in simulations of routines that call the software floating-point functions.

Addition	Subtraction	Multiplication	Division	Negation
42.4	48.6	39.8	63.2	2.4

Table 4.1: Software floating-point timings (μ s) for basic operations.

EQ	NE	LT	LE	GE	GT
13	13.4	13.2	13.2	13.2	13.2

Table 4.2: Software floating-point timings (μ s) for logical operations.

Increment	Decrement	Int to Float	Unsigned to Float	Float to Int	Float to Unsigned
58	58	12.4	11	6.8	6.8

Table 4.3: Software floating-point timings (μ s) for miscellaneous operations.

4.6 Odyssey Architecture Modeling

Every RPPT application contains an architecture model which accounts for 1) the software overhead associated with interprocessor communication, and 2) the transmission time of a message over the interconnection network. Software overhead is handled by delaying the processor at appropriate points in the ASIM procedures (generally at the beginning and end of the routines). The length of delay is determined by a call to a user supplied routine which returns the time necessary for software overhead at that point. Bus transmission time is accounted for in a CSIM-coded function, *UserSend*, which is specific to the modeled architecture. *UserSend* is called at points in the ASIM routines where the simulated message transfer occurs, and delays the calling processor by the time needed to transmit the message. Each of these components of the architecture model is discussed more fully below.

4.6.1 Software Overhead

Generally, ASIM provides the ability to define initial and final software overhead in a communication routine as a function of the message size. Because of the internal structure of the Odyssey Concurrent C synchronous message-passing routines, however, overhead is dependent upon the order in which the sender/receiver pair arrive at an

interaction point. This could not be modeled precisely without the undesirable prospect of modifying the current ASIM software.

In order to sidestep this problem, we have approximated the true software overhead involved by using an average of the 2 distinct possibilities (i.e., 1: sender arrives first, and 2: receiver arrives first). Virtually all software overhead occurs before the bus transfer so no accounting is necessary after the transfer of the message. Table 4.4 gives software overhead cycle counts for the synchronous primitives in each case, and the average values which were used in the simulation.

	SendRequest()	ReceiveRequest()	Reply()
Sender Arrives First	153.8	328.2	267.2
Receiver Arrives First	307.4	412.4	267.2
Average	230.6	370.4	267.2

Table 4.4: Software overhead (μ s) for synchronous message passing.

The validity of this approach depends on the assumption that in a real algorithm the sender will arrive at a communication point before the receiver approximately half the time, and vice versa. Hopefully, the errors arising from overestimation will more or less cancel with those attributable to underestimation and lead to accurate results. We will see later that this is not always the case but, in general, is not a major problem.

4.6.2 Bus Transfer

The Odyssey communication routines have been implemented such that a semaphore controls use of the SP bus, allowing only one processor access at any given time. Measurements have shown that in this instance, transmission time over the SP bus is a

constant $1.94 \mu\text{s}/16\text{-bit word}$ of the message. Also, because of the aforementioned block transfer capabilities of the architecture, it is most efficient to transfer long messages in blocks of 256 words. This naturally leads to a need to packetize messages. Packetization time on the Odyssey takes $85.2 \mu\text{s}/256\text{-word block}$. Although packetization is actually software overhead, in the case of the Odyssey it more naturally fits into the bus transfer aspect of the architecture and, hence, is handled in UserSend.

When UserSend is called it calculates the total packetization overhead required for the message and adds this to the time needed to transmit the packets over the bus. It then waits (a P operation) on the semaphore controlling access to the SP bus, delays the processor for the required time, and finally releases (V) the semaphore.

Chapter 5

Algorithms

5.1 Introduction

The results contained in this thesis are mostly derived from the execution of parallel versions of three algorithms. The algorithms chosen were merge sort, 2-dimensional convolution, and successive over-relaxation. These algorithms were selected primarily because they exhibited relatively dissimilar computational aspects, which was desired in order to test the simulator as thoroughly as possible.

Although the algorithms are computationally very different, communications patterns are quite similar. This is primarily a result of the architecturally motivated need for course-grained communication structures and certain assumptions which were made for more or less arbitrary reasons. For example, all three algorithms define a master/slave relationship where one processor, designated the master, performs a superset of the operations handled by the other processors. In all cases, data is assumed to initially reside in the master's address space. The master, then, must distribute data to the slaves at the beginning of the algorithm and collect the final results at the end. This type of procedure is reflected in the results given in Chapter 6, where performance data is taken from timings made on the master. The following sections describe the algorithms implemented here.

5.2 Merge Sort

Merging is a basic sorting technique which combines two sorted (ordered) lists into a single sorted list. Formally, given two input lists (x_1, x_2, \dots, x_j) and $(y_{j+1}, y_{j+2}, \dots, y_k)$ such that $x_1 \leq x_2 \leq \dots \leq x_j$ and $y_{j+1} \leq y_{j+2} \leq \dots \leq y_k$, merging generates an output list $(z_1, z_2, \dots, z_j, z_{j+1}, \dots, z_k)$ such that $z_1 \leq z_2 \leq \dots \leq z_j \leq z_{j+1} \leq \dots \leq z_k$.

A merge operation begins by comparing the smallest elements in each input list and placing the lesser first in the output list. Then, ignoring the element already in the output list, the next smallest element is chosen from the remainder of the input lists and placed second in the output list. This continues until all input elements have been placed in sorted order in the output list.

A completely unordered list may be sorted by merging in stages. For a data size of N , begin with N ordered lists each of size 1. Merge these lists in pairs to give $\frac{N}{2}$ ordered lists of size 2. These lists can then be merged in pairs to get $\frac{N}{4}$ lists of size 4. Continue in this manner until a single ordered list of size N is generated. A total of $\log_2 N$ stages is necessary to perform the sort. Figure 5.1 shows an example for $N = 8$.

Merging two lists containing a total of N elements requires a minimum of $N - 1$ comparison operations. Also, $\log_2 N$ stages are needed to merge sort an initially unordered list. Hence, merge sort is an $O(N \log_2 N)$ algorithm, which is optimal for sequential sorting [18, 19].

The parallel version of merge sort implemented here is straightforward and requires little explanation. Initially, an unsorted list of size N exists on the master processor. The master distributes a section containing $\frac{N}{P}$ elements in turn to each of the $P - 1$ slave

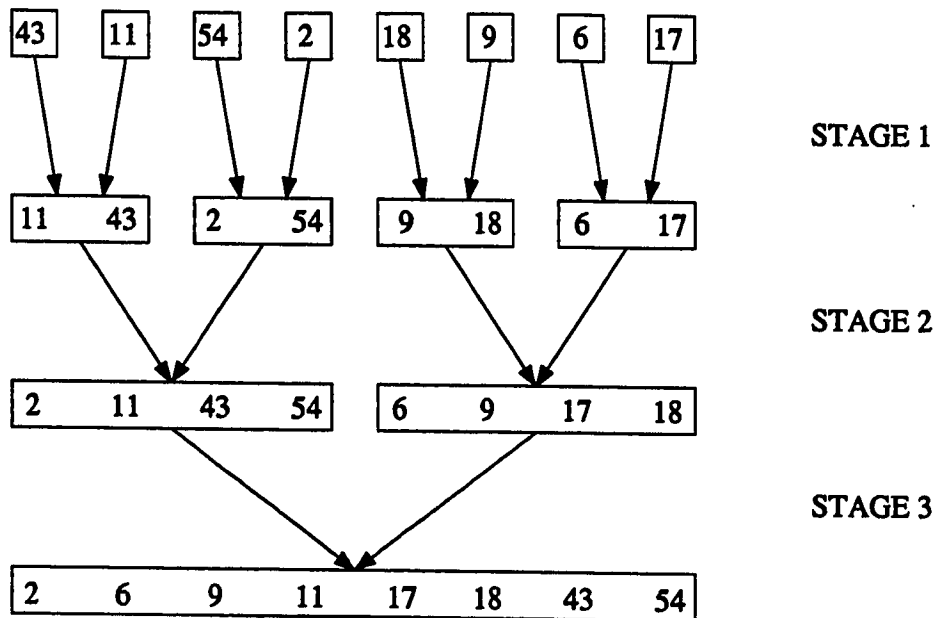


Figure 5.1: Example of sequential merge sort, $N=8$.

processors, keeping the last section for itself. All processors (including the master) then merge sort their sublists. Finally, the master collects the sublists from the slaves and sequentially completes the merge sort with the P sorted sublists of size $\frac{N}{P}$. In pseudo-code this looks like:

```

If (master)
    generate random data set of size N
    send  $N/P$  data elements to each slave
else /* slaves */
    receive data from master
  
```

Merge sort N/P data items

```

If (master)
    receive solution from slaves
    complete merge sort locally
else /* slaves */
    send partial solution to master
  
```

The computational complexity of this algorithm is

$$O\left(\frac{N}{P}\log_2\frac{N}{P} + N\log_2P\right), \quad (1)$$

which can be manipulated into the form

$$O\left(\frac{N}{P}\log_2N + N\left(\frac{P-1}{P}\right)\log_2P\right). \quad (2)$$

The first term in (2) is optimal for parallel sorting while the second indicates why this algorithm is only useful for $P \ll N$. We discuss this more in Chapter 6.

5.3 2-Dimensional Convolution

2-dimensional convolution is a fundamental filtering operation in digital image processing. It is frequently the first operation performed once an image is acquired. There are several ways to perform 2-d convolution, some involving Fast Fourier Transform techniques. Here, we have chosen to implement 2-d convolution in the most straightforward fashion.

Referring to Figure 5.2, u is an $N \times N$ image matrix and k is a 3×3 *kernel*. A kernel may be other sizes but most commonly is 3×3 . Convolution of the kernel with the image is performed by overlaying each point in the image with the kernel and replacing the image value at that point by the sum-of-products of corresponding kernel and image values.

Hence, at each point (i, j) in the image we update as in

$$\begin{aligned} u_{i,j} = & k_{1,1}u_{i-1,j-1} + k_{1,2}u_{i-1,j} + k_{1,3}u_{i-1,j+1} + k_{2,1}u_{i,j-1} + k_{2,2}u_{i,j} + \\ & k_{2,3}u_{i,j+1} + k_{3,1}u_{i+1,j-1} + k_{3,2}u_{i+1,j} + k_{3,3}u_{i+1,j+1}. \end{aligned} \quad (3)$$

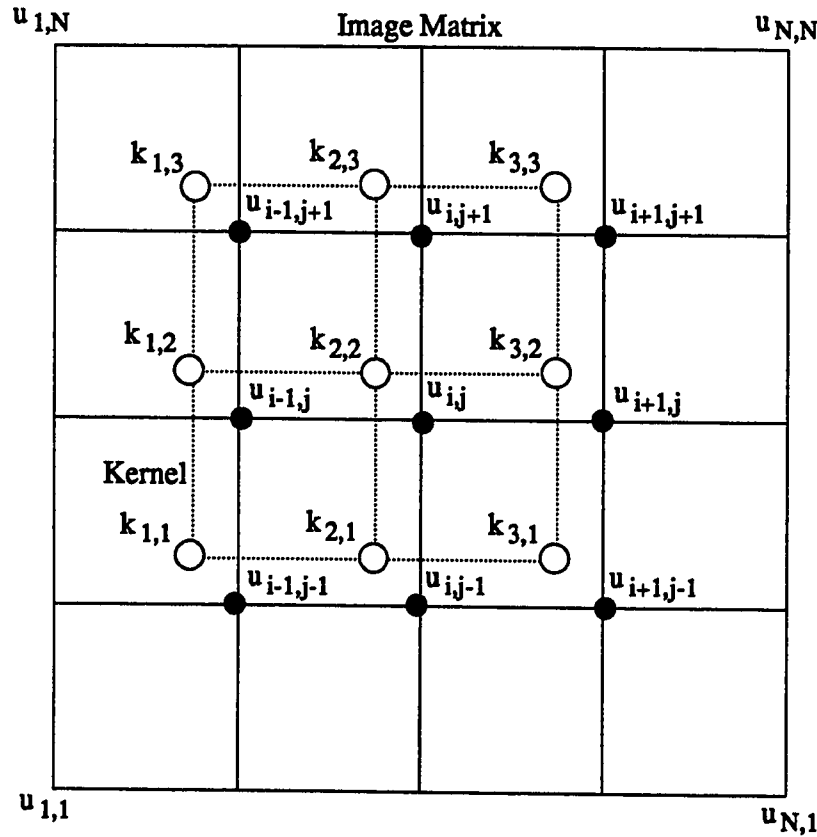


Figure 5.2: Convolving a 3x3 kernel with an NxN matrix.

Using an NxN image and a KxK kernel, sequential convolution is an $O(K^2 N^2)$ algorithm [20].

For the parallel implementation, the master processor divides the image into P strips of $\frac{N}{P}$ rows, sending a strip to each processor. The master must also send a boundary row on either side of each strip to handle the kernel overlay at the boundaries. The processors then convolve in parallel their individual sections of the image matrix. When the convolution is completed, boundary rows are exchanged in preparation for a following algorithm. Finally, the slaves send results back to the master. In pseudo-code this looks like:

```

If (master)
    get image of size  $N^2$ 
    send  $N/P$  rows + 2 boundary rows to each slave
else /* slaves */
    receive  $N/P+2$  rows from master

```

Convolve N^2/P points

Exchange boundary rows with processors on either side

```

If (master)
    receive solution from slaves
else /* slaves */
    send solution to master

```

Computationally, this parallel version of 2-d convolution is $O\left(\frac{K^2 N^2}{P}\right)$, which would give linear speedup were it not for the communication requirements.

The boundary row exchange is not necessary for pure convolution since no computation is performed after this point. However, in a typical image processing application some operations would invariably follow the convolution and most likely need the boundaries. It was thought that including this exchange more realistically accounted for the interprocessor communication needed in the algorithm.

5.4 Successive Over-Relaxation

5.4.1 Theory

Successive Over-Relaxation (SOR) is an iterative method for solving elliptic partial differential equations resulting from equilibrium or steady state (boundary value)

problems. The classic example of an elliptic PDE solvable using relaxation methods is *Poisson's Equation* given by

$$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = \rho(x, y). \quad (4)$$

Here, (x, y) are spatial coordinates in the region of interest, $\rho(x, y)$ is the known source term (generally consisting of charge/mass density), and $u(x, y)$ is the desired solution composed of the field strength at every point in the region, subject to some boundary value conditions. If $\rho(x, y) = 0$ then Poisson's Equation reduces to *Laplace's Equation*.

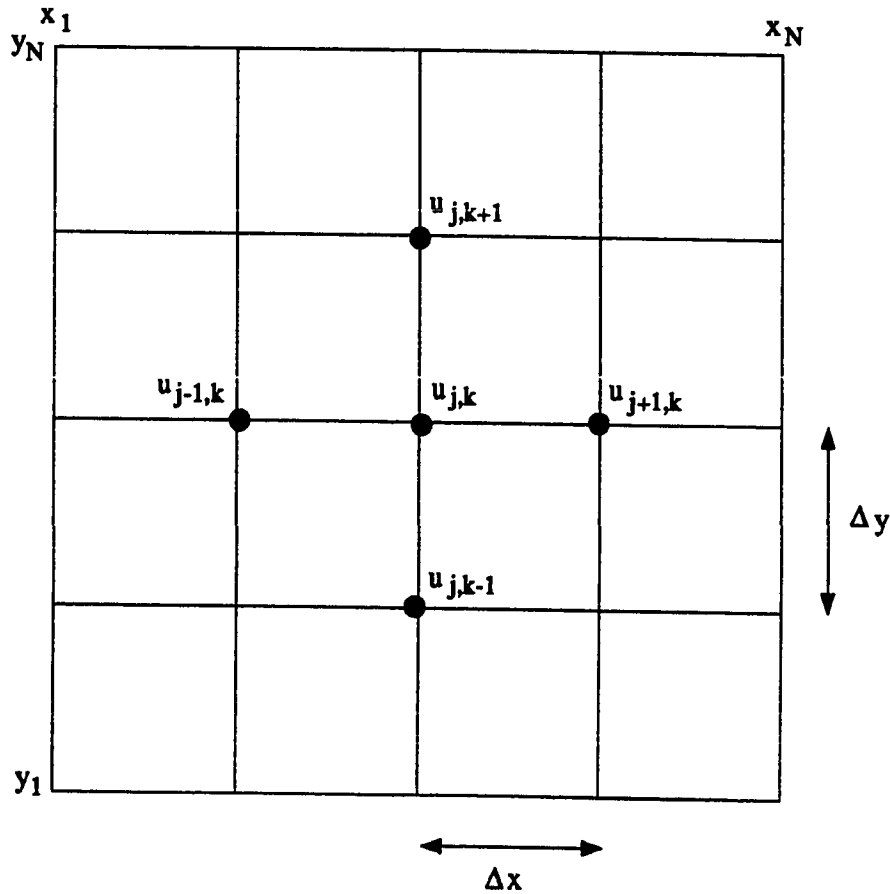


Figure 5.3: Grid over coordinate space for solving elliptic PDE.

By laying a grid over the coordinate space, as shown in Figure 5.3, $u(x, y)$ can be represented by its values at the set of points

$$\{(x_j, y_k) : x_j = x_0 + j\Delta x, y_k = y_0 + k\Delta y\} \quad (5)$$

$$j, k = 0, 1, \dots, N + 1$$

where Δx and Δy refer to the grid spacing. A finite difference representation of (4) can be obtained by approximating the second derivative at (x_j, y_k) using the nearest neighbor grid points. Letting $u(x_j, y_k)$ be written as $u_{j,k}$ and $\Delta x = \Delta y = \Delta$ we have

$$\frac{u_{j+1,k} - 2u_{j,k} + u_{j-1,k}}{\Delta^2} + \frac{u_{j,k+1} - 2u_{j,k} + u_{j,k-1}}{\Delta^2} = \rho_{j,k}. \quad (6)$$

Rearranging terms gives

$$u_{j+1,k} + u_{j-1,k} + u_{j,k+1} + u_{j,k-1} - 4u_{j,k} = \Delta^2 \rho_{j,k}. \quad (7)$$

The system of equations implied by (7) can be solved directly using linear-algebraic methods by transforming the problem to a matrix representation $\mathbf{A} \cdot \mathbf{U} = \mathbf{b}$ [21, 22]. However, for an $N \times N$ grid size this necessitates a matrix \mathbf{A} of size $N^2 \times N^2$. Consequently, space requirements generally limit this method of solution to problems of relatively small size. For larger sizes, these problems are generally solved using a relaxation method.

Relaxation involves iterating over the points in the grid. At each iteration, the grid points are updated to a closer approximation of the solution. Eventually, the procedure converges to a steady state value, which is the desired solution.

The classical example of a relaxation algorithm, *Jacobi's method*, consists of solving (7) for $u_{j,k}$ giving

$$u_{j,k}^{(n+1)} = \frac{1}{4} \left(u_{j+1,k}^{(n)} + u_{j-1,k}^{(n)} + u_{j,k+1}^{(n)} + u_{j,k-1}^{(n)} \right) - \frac{\Delta^2}{4} \rho_{j,k}. \quad (8)$$

Here, $u_{j,k}^{(n)}$ represents the solution at $u_{j,k}$ after the n^{th} iteration. So each iteration updates grid points using values from the previous iteration. It can be shown that the number of iterations required to converge to a fixed criteria using Jacobi's method is proportional to N^2 .

A minor modification of Jacobi's method, the *Gauss-Seidel method*, uses updated values when they become available as in

$$u_{j,k}^{(n+1)} = \frac{1}{4} \left(u_{j+1,k}^{(n)} + u_{j-1,k}^{(n+1)} + u_{j,k+1}^{(n)} + u_{j,k-1}^{(n+1)} \right) - \frac{\Delta^2}{4} \rho_{j,k}. \quad (9)$$

Gauss-Seidel converges twice as fast as Jacobi, but still needs $O(N^2)$ iterations.

Successive Over-Relaxation is a modification of the Gauss-Seidel method which converges in $O(N)$ iterations. It is based on the observation that the methods discussed above tend to undershoot the solution at each iteration, and therefore converge from one direction only. If, at each iteration, we update the solution further in the "correct" direction, the procedure will converge faster. This is called *over-relaxation*. Taking (7) and replacing the constant coefficients by more general coefficient terms, a residual value can be defined by

$$\xi_{j,k} = a_{j,k}u_{j+1,k} + b_{j,k}u_{j-1,k} + c_{j,k}u_{j,k+1} + d_{j,k}u_{j,k-1} + e_{j,k}u_{j,k} - f_{j,k}. \quad (10)$$

This is basically the left side of (7) minus the right, and is a measure of the error in $u_{j,k}$ at this stage of the computation. Finally, we update the solution according to

$$u_{j,k}^{(n+1)} = u_{j,k}^{(n)} - \omega \frac{\xi_{j,k}}{e_{j,k}} \quad (11)$$

where ω , the relaxation parameter, can take on values between 0 and 2. At $\omega = 1$, (11) reduces to (9). For $1 < \omega < 2$ we are over-relaxing, and (hopefully) converging

faster. The efficiency of SOR is dependent upon a proper choice of ω , which is not generally known initially. Fortunately, ω can be dynamically generated as the algorithm progresses. The basic SOR algorithm proceeds as follows [23]:

```

Input coefficient matrices and boundary value conditions
Guess the solution (generally initialized to zero)
Initialize  $\omega$ 

```

```

While (not done)
  For each grid point  $(j,k)$ 

    calculate residual  $\xi_{j,k}$ 
    update sum of residuals
    calculate  $u_{j,k}^{(n+1)}$ 

```

```

Update  $\omega$ 

```

```

If (sum of residuals < stopping criteria)
  done

```

5.4.2 Parallel Implementation

The parallel version of SOR implemented here is a straightforward modification of the sequential algorithm. Basically, each of p processors is responsible for $\left(\frac{1}{p}\right)^{th}$ of the solution. For example, processor 0 calculates the first $\left(\frac{1}{p}\right)^{th}$ of the rows of the solution, processor 1 calculates the second $\left(\frac{1}{p}\right)^{th}$, etc. One processor is designated the master and performs a superset of the computation done by the slave processors. The master is responsible for initializing the data, distributing the data to the slaves, calculating the global sum-of-residuals using the local sums-of-residuals received from the slaves at each

iteration, returning the global sum-of-residuals to the slaves, and collecting the results at the end of the algorithm. So the parallel implementation proceeds as:

```

If (master)
    input and initialize data
    send data to slaves
else /* slaves */
    receive data from master

While (not done)
    For each grid point  $(j,k)$  on this processor
        calculate residual  $\xi_{j,k}$ 
        update local sum of residuals
        calculate  $u_{j,k}^{(n+1)}$ 

    if (master)
        receive local residual sums from slaves
        calculate total residual sum
        send total residual sum back to slaves
    else /* slaves */
        send local sum of residuals to master
        receive total residual back from master

Update  $\omega$ 

If (sum of residuals < stopping criteria)
    done
else
    exchange boundary rows with nearest neighbor
    processors in preparation for the
    next iteration

If (master)
    receive solution from slaves
else /* slaves */
    send partial solution to master

```

Note from (10) that each processor needs a boundary row of the solution matrix on either side of the rows it is responsible for. At the end of each iteration, these boundary rows must be exchanged with neighboring processors in order for the algorithm to proceed properly.

Chapter 6

Results

6.1 Introduction

The purposes of this chapter are threefold: 1) to discuss possible errors which can lead to inaccuracies in the performance predictions of the simulator, 2) to validate the accuracy and efficiency of the simulator by comparing performance predictions obtained through simulation with those actually observed on the Odyssey system, and 3) to show how the simulator can be used to analyze the performance of either a larger Odyssey system than is available here or a system where some of the Odyssey architectural parameters have been altered. To these ends, the algorithms discussed in Chapter 5 have been run on the Odyssey and the simulator for various numbers of processors and a wide range of problem sizes. The synchronous message-passing facilities of OCC provided the mechanism for interprocessor communication. Statistics on execution time, speedup, prediction error, and simulator efficiency have been collected and are presented in the following sections, along with an analysis of the results.

6.2 Sources of Error

Differences between the performance indices predicted by simulation and those actually obtained on the Odyssey system can be attributed to one or more of the following:

- 1) timing inaccuracies,

- 2) profiling errors,
- 3) cross-profiling block mismatches,
- 4) architecture model deficiencies, and
- 5) unprofilable code whose execution time must be estimated.

6.2.1 Timing

Timing on the Odyssey involves use of the 16-bit memory-mapped hardware timer register which resides on the TMS320C25 chip. This timer is a down-counter that can be initialized to an arbitrary 16-bit value. Once initialized, the timer decrements at each clock tick, eventually generating a hardware timer interrupt when it zeroes. This allows routines which take less than 65536 cycles to be accurately timed. In order to time longer routines, a software managed 32-bit timer has been implemented. Routines to start the software timer, stop the timer, and service timer interrupts are available. At each timer interrupt, the interrupt handler increments the 32-bit value by the elapsed 65536 cycles and reinitializes the hardware timer to its 65K maximum. Also, each timer routine subtracts its own software overhead from the accumulated timer value so that as little error as possible propagates to the timing results. Measurements on code whose execution time is known have shown that the timer as implemented generates no more than 0.1% error when timing sequential sections of code.

Another timing consideration is the observation that timing a parallel routine may alter the way in which processes interact. This is a result of the timer interrupt service routine executing code which does not exist in the actual algorithm and, hence, is not

being simulated. The interrupt service routine takes 208 cycles, in contrast to the 65536-cycle period of the timer register. The resulting possible error is $208/65536 \rightarrow 0.32\%$.

In light of the above, the timing routines appear to be accurate enough so that no more than 0.5% error is introduced by their use. This, as we shall see, is smaller than the errors caused by many other factors and can be effectively ignored.

6.2.2 Profiling

The greatest potential profiling difficulty encountered with respect to the TMS320C25 is the fact that the execution time of an instruction is dependent upon the memory space (on-chip or local) in which its operands reside. Since profiling takes place prior to execution it is impossible to determine (in some cases) how many cycles an instruction will take. This is currently being handled by assigning the total data space of an Odyssey program to local memory, and setting the profiler up so that it uses the corresponding cycle counts for operands in local memory. This allows for almost exact profiling.

The only known error introduced during profiling is for branch instructions. A branch accounts for three cycles if it is taken and two cycles otherwise. We have assumed that branches tend to be taken more often than not, and used the three-cycle value. This will cause the simulated times to be slightly higher than the actual execution time since an extra cycle is added for a branch which is not taken.

6.2.3 Cross-Profiling

In most cases the cross-profiler does a good job of matching basic blocks between compilers. Occasionally, however, the TMS320C25 C compiler generates small blocks

which have no correspondence in 68020 code. The execution of these blocks is not accounted for in the simulation, and therefore leads to slightly low simulated timings. Note that the known positive errors introduced by the profiler are offset, at least in direction, by the possible negative errors generated in cross-profiling.

6.2.4 Modeling

As was discussed in Chapter 4, average times for software overhead in the message-passing routines were inserted into the Odyssey architecture model. The resulting accuracy of the model is dependent upon the complexity of communication between processors. Algorithms whose communication patterns are complex enough that a sender and receiver are approximately equally likely to arrive first at an interaction point will be simulated accurately. Algorithms with very simple communications, say an initial data distribution and a final data collection, will not be simulated as accurately. This is a consequence of the much greater likelihood that the sender (or receiver) will arrive first most of the time. A more accurate method of accounting for software overhead would alleviate this somewhat, but is not currently available.

6.2.5 Unprofilable Code

For code which can not be cross-profiled, estimated execution times must be inserted directly into the simulator at the profiling stage. This is a problem primarily with routines for which C source code is unavailable and whose execution time is data dependent, such as software floating-point. As was discussed earlier, most of the available floating-point routines exhibit relatively fixed execution time, but the important addition and

subtraction operations take widely varying times to execute depending upon the values of their operands. For simulation purposes, an average observed value was used in the profiling step. We shall see that the data-dependent simulation errors which result from an algorithm that uses floating-point can be fairly large. Without the ability to cross-profile these routines there is no satisfactory solution to this problem.

6.3 Validation

Previous RPPT validation work has involved simulations of an Intel iPSC⁴ hypercube and the V-System⁵, a distributed operating system running on Sun workstations [24, 25]. The iPSC study was handicapped by a lack of knowledge of the specifics of the proprietary communication software involved, while the V-System measurements had to be made on a public system with an unknown and varying amount of activity generated by other users. A major motivation of this work was the desire to validate the RPPT simulator against an architecture whose system source code was available and that could be used on an exclusive basis. The Odyssey obviously fulfills these requirements since it may only be accessed in a single-user mode and the Odyssey Concurrent C interprocessor communications software was written here at Rice.

The following sections detail the results obtained by running three very different algorithms on both the Odyssey board and the simulator. Statistics on execution time and speedup serve as the primary performance indices for validation purposes. Also, the

⁴ iPSC is a trademark of Intel Corporation.

⁵ V-System is a trademark of Leland Stanford Junior University.

efficiency of RPPT is measured through a comparison of identical programs executing as both RPPT simulations and basic Concurrent C programs.

6.3.1 Execution Time Results

Figure 6.1 plots total execution time, both real and simulated, of the merge sort algorithm running on various numbers of processors (P) for small data sets (of size N). Figure 6.2 does the same for larger data sets. Figure 6.3 shows simulation error relative to real execution time. A positive error indicates that the simulator predicted a value greater than that actually observed, while a negative error indicates that the simulator's prediction was low. Figures 6.4–6.6 are similar to Figures 6.1–6.3 except the 2-dimensional convolution results are plotted. In this case, a data size of N refers to an $N \times N$ data matrix. Figures 6.7–6.9 show the SOR results run on an $N \times N$ grid.

Profiling and Cross-Profiling Validation Neither merge sort nor 2-d convolution use any floating-point arithmetic so all the code is actually profiled. Consequently, the results of merge sort and 2-d convolution run on a single processor are a fair test of the inherent accuracy of the profiler and cross-profiler.

For merge sort run on a single processor, the errors observed were never more than $\pm 1.0\%$, and this was for data sets ranging from 32 to 16384 words. Errors for 2-d convolution were consistently around -1.7% . Taking the two algorithms as a unit, most of the capabilities of the TMS320C25 instruction set were exercised: merge sort heavily uses comparison operations while 2-d convolution makes much use of the integer arithmetic capabilities of the chip. Also, both algorithms perform a significant amount

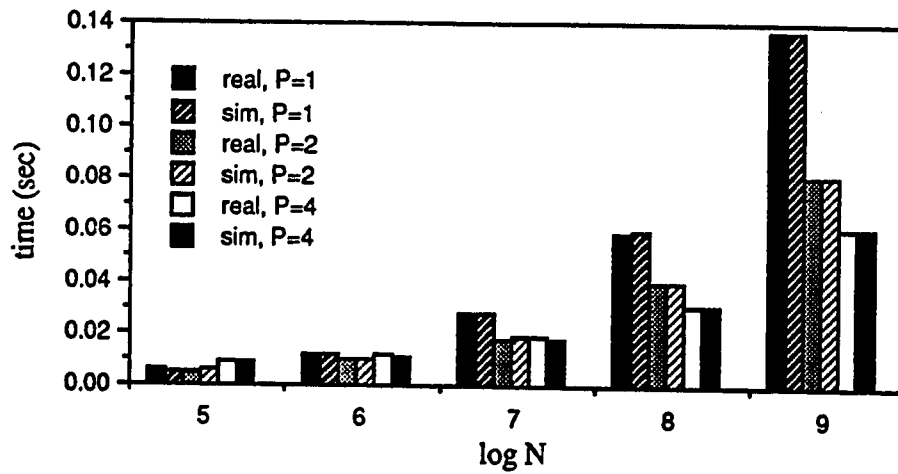


Figure 6.1: Merge Sort, Execution Time.

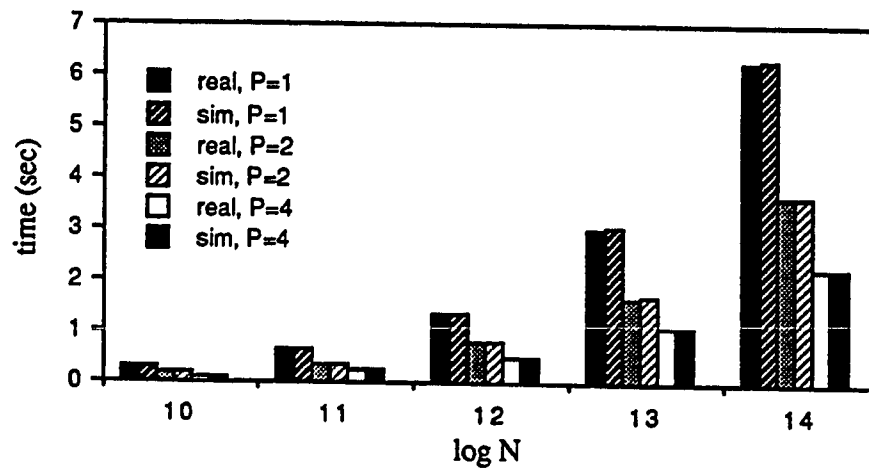


Figure 6.2: Merge Sort, Execution Time.

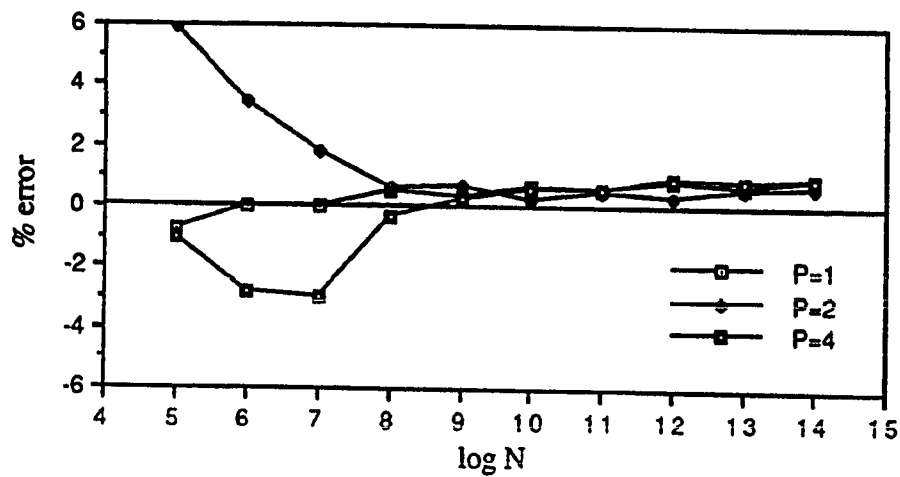


Figure 6.3: Merge Sort, % Error (Exe. Time).

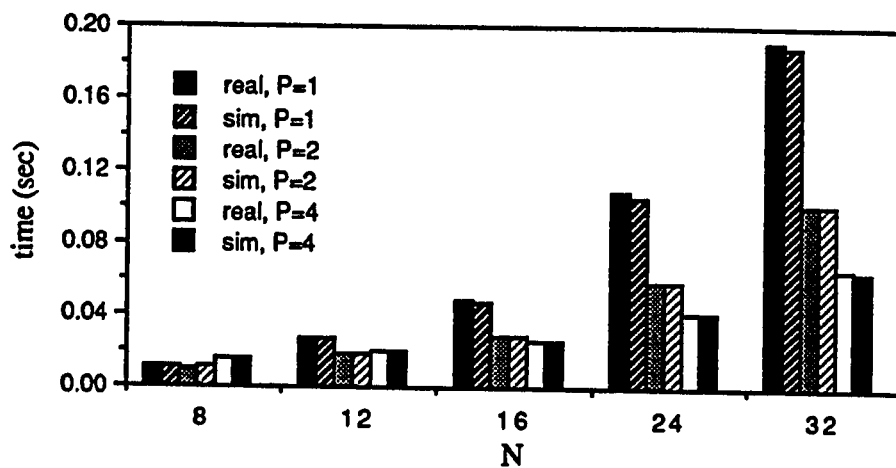


Figure 6.4: 2-D Convolution, Execution Time.

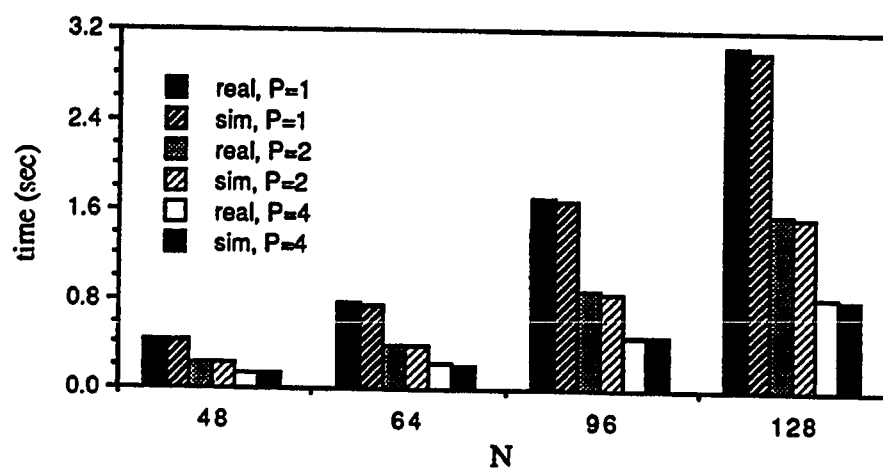


Figure 6.5: 2-D Convolution, Execution Time.

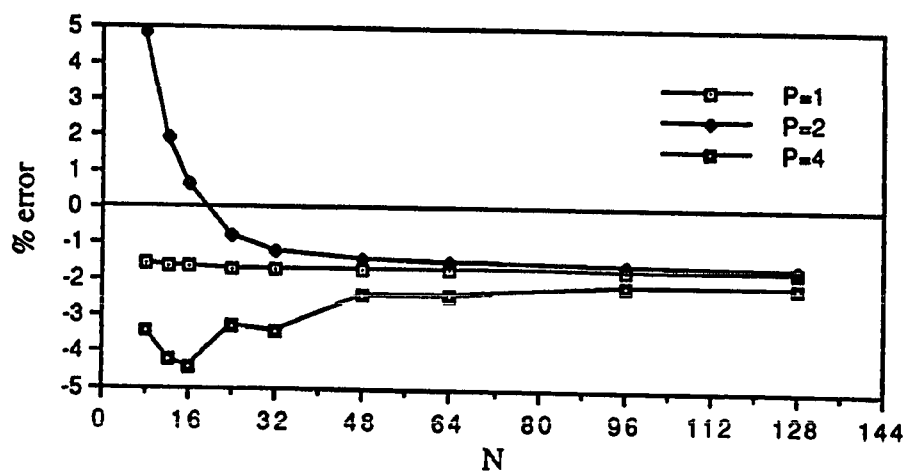


Figure 6.6: 2-D Convolution, % Error (Exe. Time).

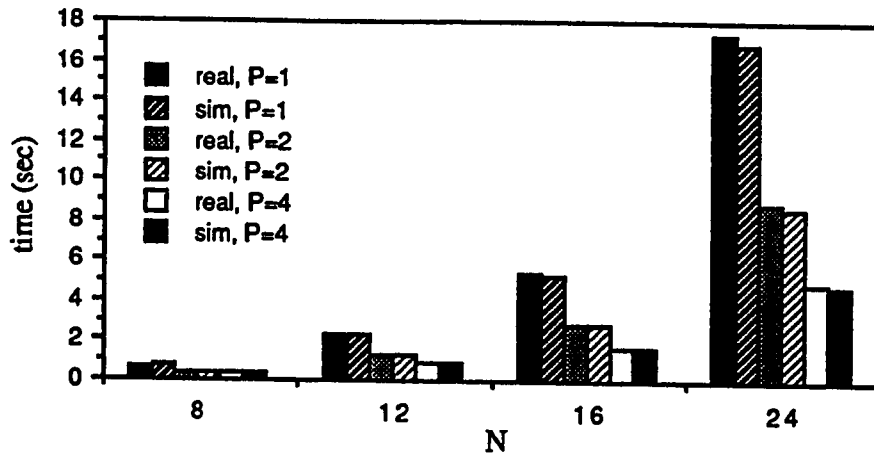


Figure 6.7: SOR, Execution Time.

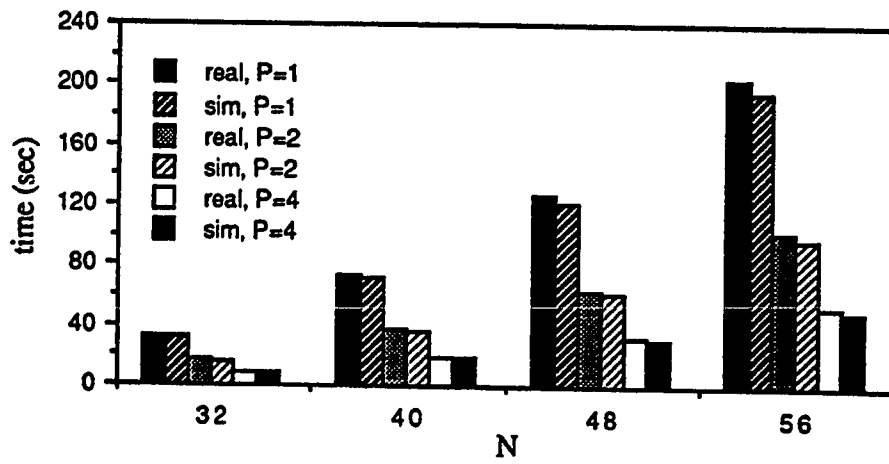


Figure 6.8: SOR, Execution Time.

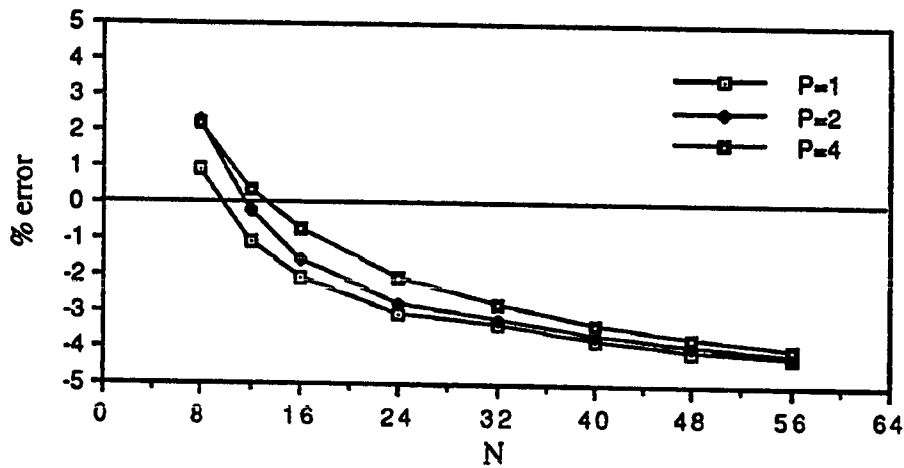


Figure 6.9: SOR, % Error (Exe. Time).

of function calling, parameter passing, and operand address generation (both global and local). Overall, the errors observed ranged from -1.73% to $+0.96\%$. It seems reasonable, then, to expect errors no more than about $\pm 2\%$ for any algorithm which can be completely profiled. This is quite accurate and probably sufficiently precise for most purposes.

Unprofiled Code Validation The SOR algorithm implemented here spends most of its execution time inside the software floating-point routines. As was discussed previously, problems with determining reasonably accurate average execution times for addition and subtraction were encountered. Looking at the one-processor SOR results, errors of up to -4.2% were observed, and the trend was towards even larger errors for larger data sets. Without the ability to cross-profile the software floating-point routines, these errors cannot be eliminated. However, they may be used as a base from which to estimate the magnitude of simulation error attributable to other factors.

Architecture Model Validation While the uniprocessor results measure simulation error solely resulting from inaccuracies propagated through the profiling steps, multiprocessor results serve as measures of the Odyssey architecture modeling deficiencies. From the error plots shown in Figures 6.3, 6.6, and 6.9, the important observations to make are the following:

- 1) As the problem size increases, the influence of the architecture model decreases. This occurs because for a fixed number of processors, computational requirements increase more rapidly with problem size than the corresponding increase in interprocessor communication. The communication needs of merge sort, for instance, consist of

the initial data distribution and final data collection: an $O(N)$ operation. The computational aspects of merge sort, on the other hand, are $O(N \log_2 N)$ for fixed P . 2-d convolution and SOR exhibit similar relative increases in computational requirements with increasing problem size. Hence, as the data size is expanded, simulation errors asymptotically approach the one-processor baseline.

- 2) For smaller problem sizes where interprocessor communication overhead accounts for a significant fraction of execution time, errors of up to 6% were observed. These errors were evident only in the very smallest cases, and improved markedly with increasing problem size.
- 3) For all three algorithms, larger errors were evident in the two-processor runs than with four processors. This is an effect related to the software overhead modeling deficiencies discussed previously. The two-processor cases were affected more because communication patterns were simpler. For instance, the data distribution step in all three algorithms is structured such that the sender (master) always initially arrives second, causing an overestimation of software overhead and a corresponding positive simulation error. This effect was more dramatic in the two-processor cases because the fixed error involved was “amortized” over only two message passing events as opposed to four.
- 4) Overall, the accuracy of architecture modeling is fairly high. Significant errors become apparent only for very small problem sizes. None of the algorithms discussed would ever be run for such small data sets as was done here. 2-d convolution, for example, would most likely use as large an image matrix as could fit in the memory

of the Odyssey processors. For larger, realistic data sizes, architecture modeling generated errors of less than 3%. This is accuracy on the order of the basic profiling steps, and therefore sufficient for most purposes.

6.3.2 Speedup Results

An ultimate purpose of RPPT is to determine the efficacy with which a concurrent algorithm is matched to either an imaginary parallel architecture or an actual one whose physical parameters have been altered. Generally, the usefulness of a parallel algorithm is measured by the number of processors it may be run on while still exhibiting near linear *speedup*. For this purpose, accurate speedup predictions are probably of more value than absolute execution time results.

Speedup for N processors is defined here as the ratio of uniprocessor execution time to N processor execution time for a fixed data size, so speedup may be derived directly from the execution time data given in the previous section. A speedup of S means that the N -processor execution was S -times faster than the single processor execution. Speedups may be less than one (i.e., slowdown) in which case the algorithm actually runs faster on one processor than on N processors. For the three algorithms discussed here the smallest data sizes were chosen to be those such that a speedup greater than one was observed in the two-processor runs. These are the smallest data sizes for which any speedup can be demonstrated.

Speedup results are plotted in Figures 6.10–6.15 for the two- and four-processor runs. Figure 6.10 shows speedup, both real and predicted, for merge sort running on two and four processors. Figure 6.11 plots the percent error of the prediction relative to

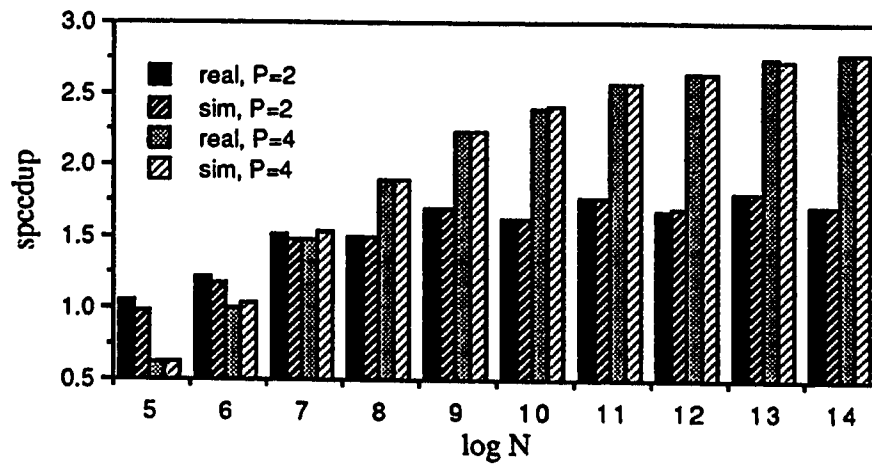


Figure 6.10: Merge Sort, Speedup.

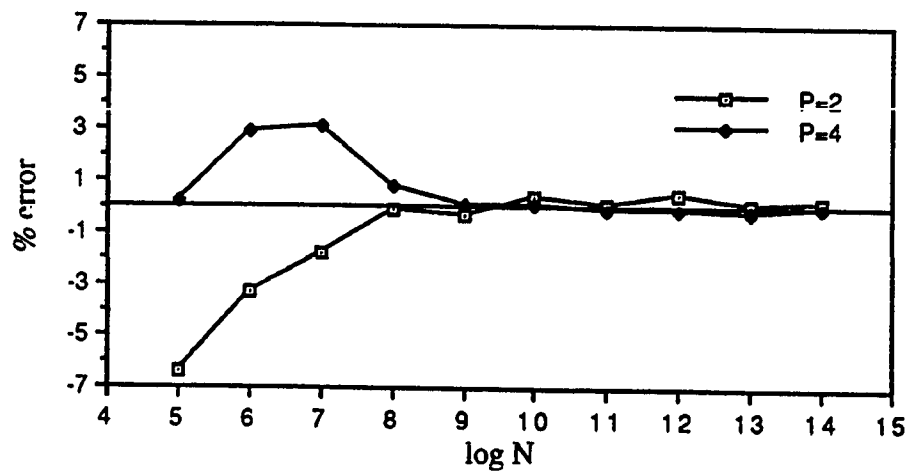


Figure 6.11: Merge Sort, % Error (Speedup).

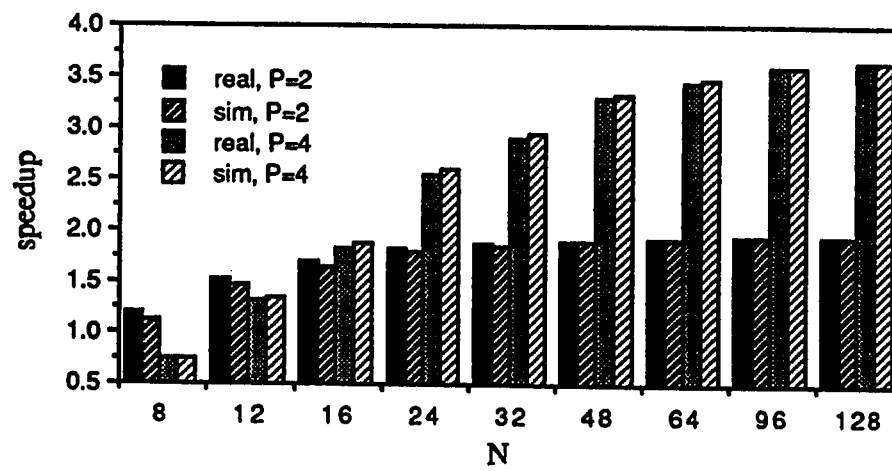


Figure 6.12: 2-D Convolution, Speedup.

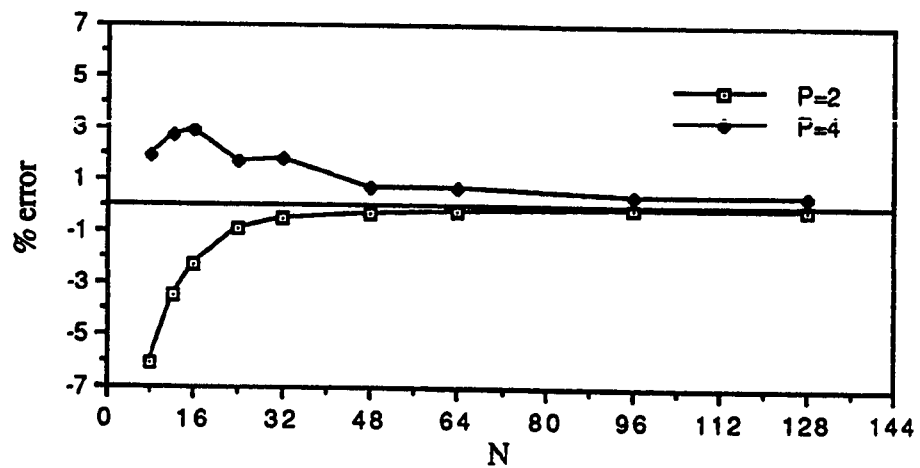


Figure 6.13: 2-D Convolution, % Error (Speedup).

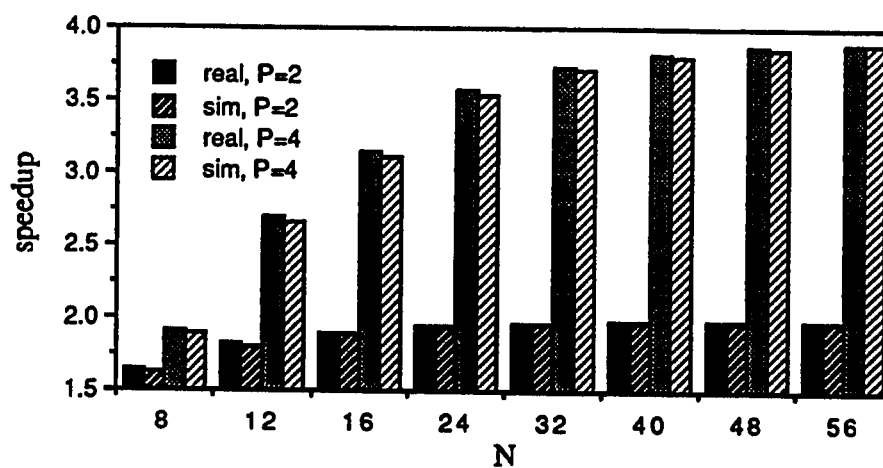


Figure 6.14: SOR, Speedup.

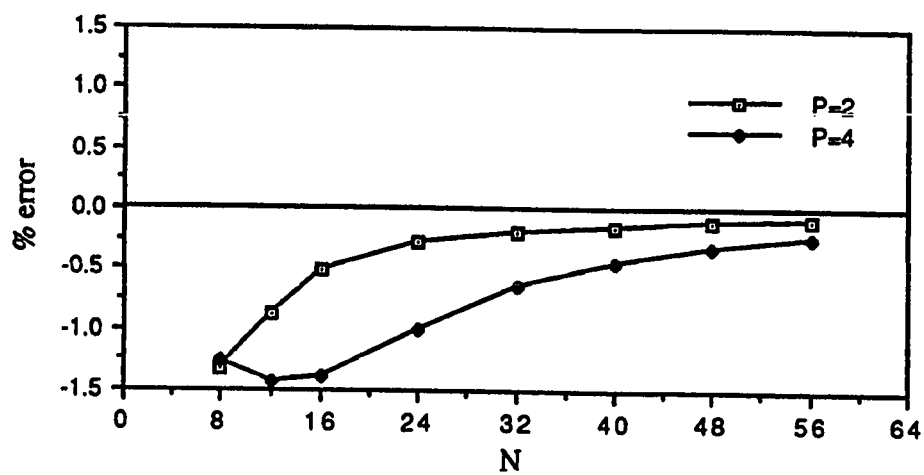


Figure 6.15: SOR, % Error (Speedup).

that actually observed. Positive errors indicate a prediction which is too large, negative errors the reverse. Figures 6.12–6.13 and 6.14–6.15 give corresponding results for 2-d convolution and SOR respectively.

For the smaller problem sizes, the magnitude of prediction error generally followed those for execution time. However, the errors in speedup prediction exhibited an interesting characteristic: as data size is increased, speedup prediction errors appear to asymptotically approach zero. Simulation of SOR, for instance, gave execution time errors near 4% for larger data sizes, but speedup errors under 0.5%. In fact, SOR consistently predicted speedup with greater accuracy than either of the other algorithms—just the reverse of the execution time response.

These curious results come about because of a fundamental difference in the makeup of execution time and speedup indices. Execution time is an absolute statistic whose value is dependent on a single index. Speedup, on the other hand, is a ratio of two distinct values which are not independent. As an example, take the extreme case of an algorithm which generates simulated execution times exactly double those for the real system. The execution time errors would consistently be +100% but speedup predictions would be correct because the linear errors in execution time (really just a scale factor) would cancel in the speedup ratio.

This type of response is most evident in an algorithm like SOR where, for a given problem size, the large data-dependent errors arising from floating-point profiling inaccuracies caused virtually identical execution time errors over different numbers of processors. These proportional errors tended to cancel in the speedup predictions and,

hence, gave much more accurate results. The same characteristic is evident in 2-d convolution, where 2–3% errors in execution time become negligible when speedup is calculated. Consequently, it appears that simulation errors which are mostly dependent upon problem size, as opposed to degree of parallelism, do not affect speedup predictions to as great a degree as execution time predictions.

6.3.3 Simulation Efficiency

The need for efficient simulation was a major motivation for the development of the RPPT. There are many ways the efficiency of simulation may be defined. For our purposes, the most reasonable measurement of efficiency is the ratio of the RPPT execution time to Concurrent C execution time. Note that by the RPPT execution time we refer to the actual time required to run the simulation, not the predicted execution time of the algorithm being simulated.

The execution of a Concurrent C program contains little overhead beyond that of a normal sequential program. Interprocessor communication overhead is mostly comprised of inexpensive transfers of blocks of data. In contrast, an RPPT simulation must not only perform everything done by Concurrent C, but must also increment a counter at each basic block, maintain simulation time through the discrete-event handling facilities of CSIM, account for the hardware and software aspects of the architecture model, and collect various performance statistics. Thus, most of the simulation overhead in an RPPT application is due to factors not accounted for by Concurrent C. Because of this we may use the execution time of a Concurrent C program as a baseline from which to measure the efficiency of the same program executing as an RPPT application.

Simulation overhead for merge sort, 2-d convolution, and SOR is plotted in Figures 6.16, 6.17, and 6.18, respectively. An overhead of X signifies that the RPPT simulation took X -times longer to run than the simple Concurrent C execution of the same program. Overheads for the smaller problem sizes of merge sort and 2-d convolution are not available because the timing mechanism was insufficiently precise to give valid results. In any case, for these sizes the execution time was less than 0.3 sec., and consequently overhead here is not of great importance. The data as shown exhibits 2 general responses:

- 1) As data size increases, simulation overhead decreases. This is an effect of the aforementioned relative increase in computational needs (and corresponding decrease in communications) for larger problem sizes.
- 2) As the number of processors is increased, simulation overhead also increases. This is mostly a result of additional processors generating more message-passing events.

The smallest overheads were associated with SOR, primarily because of the great amount of floating-point calculation required. Merge sort and 2-d convolution exhibited somewhat greater overheads. Overall, the simulation overhead appears relatively low. For the uniprocessor cases, overheads ranged from 1 to 5. With four processors the largest overhead observed was 12. This is in contrast to overheads of 100 or more for standard instruction-driven simulations, even when simulating only one processor.

6.4 Prediction

Most of this thesis has dealt with validation of the RPPT Odyssey simulation. We have seen that for the available four-processor system, the simulator's predictions are

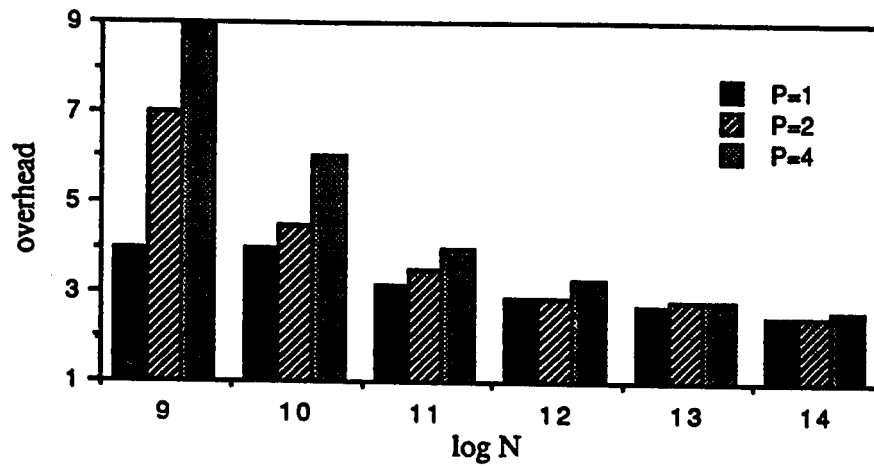


Figure 6.16: Merge Sort, Simulation Overhead.

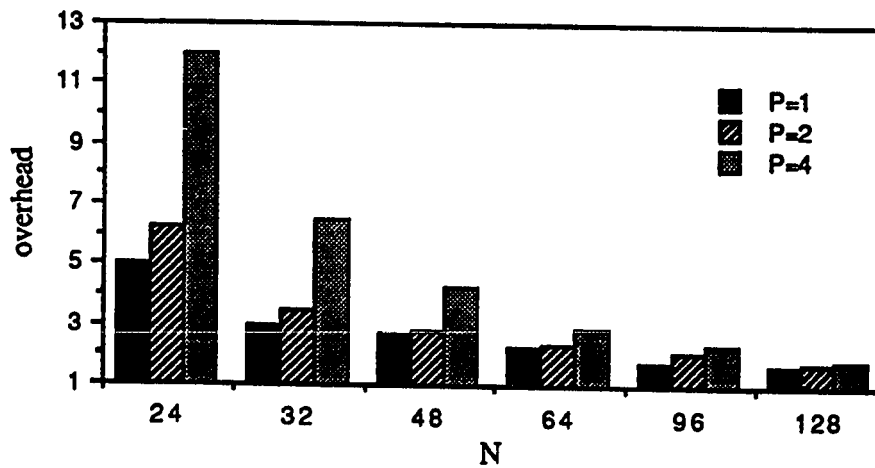


Figure 6.17: 2-D Convolution, Simulation Overhead.

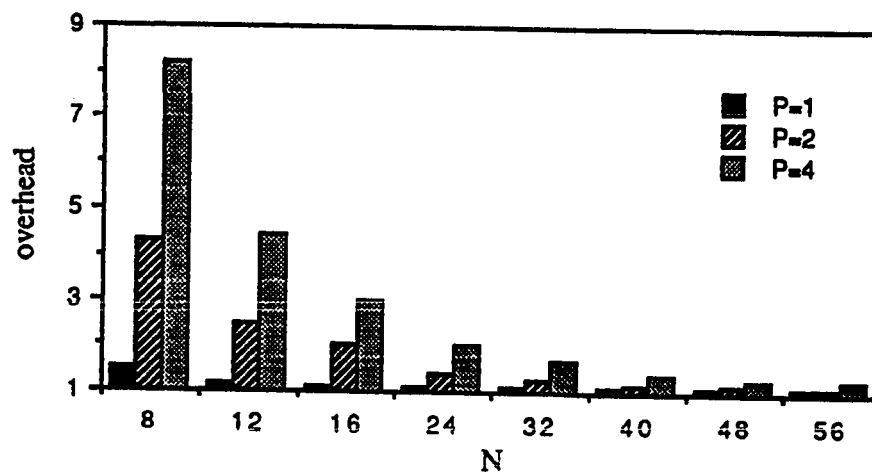


Figure 6.18: SOR, Simulation Overhead.

quite accurate over a large range of problem sizes for several dissimilar algorithms. An important application of the RPPT is, of course, to explore the performance of both larger Odyssey configurations (which are not currently available) and Odyssey architectures whose hardware/software parameters have been modified.

In this section, we use the simulator to predict Odyssey performance up to the 32-processor level. We also show the effect of some extreme modifications to the Odyssey architecture model. Of the three algorithms used here, 2-d convolution is most typical of the kind of application which would be run on the Odyssey in actual practice. We show here the results of 2-d convolution operating on two disparate data sizes.

Figures 6.19–6.21 plot results for a 32x32 data size, while Figures 6.22–6.24 are for a 128x128 input data matrix. In all cases, the number of processors (P) was varied from 1 to 32. Results are given for four architectural configurations as indicated in the figures. These configurations consist of the following.

- 1) “total”: the actual architecture model which was used for all the results given earlier in this chapter.
- 2) “no xmit”: like “total” except the transmission time of a message over the SP bus interconnection network occurs instantaneously in simulation time.
- 3) “no overhead”: like “total” except all software overhead in the model is set to zero.
- 4) “only sync”: a combination of “no xmit” and “no overhead” where, in effect, only synchronization overhead is encountered when processors communicate.

These modifications are very extreme and not physically realizable, but are nevertheless instructive. For example, “no xmit” shows the maximum possible performance gain that

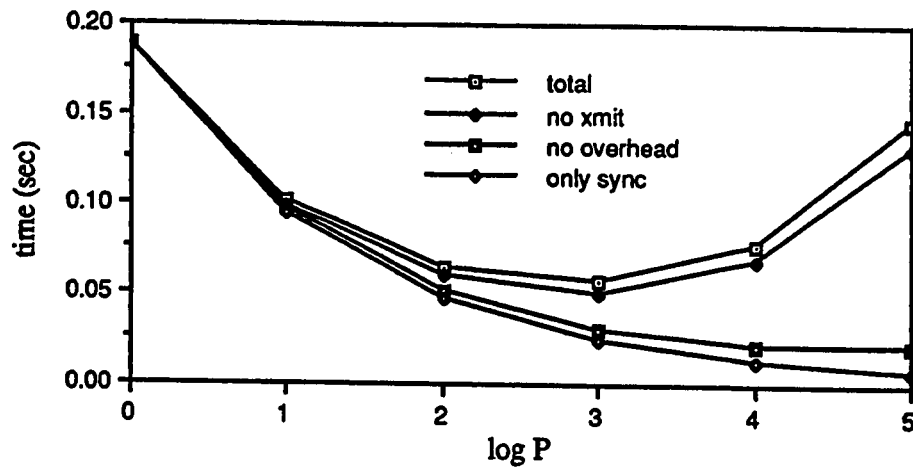


Figure 6.19: 2-D Convolution, 32x32 Matrix, Predicted Execution Time.

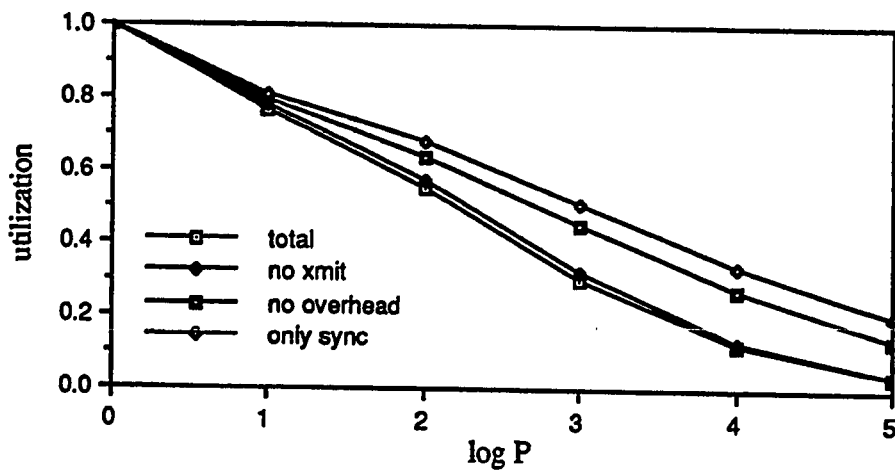


Figure 6.20: 2-D Convolution, 32x32 Matrix, Predicted Utilization.

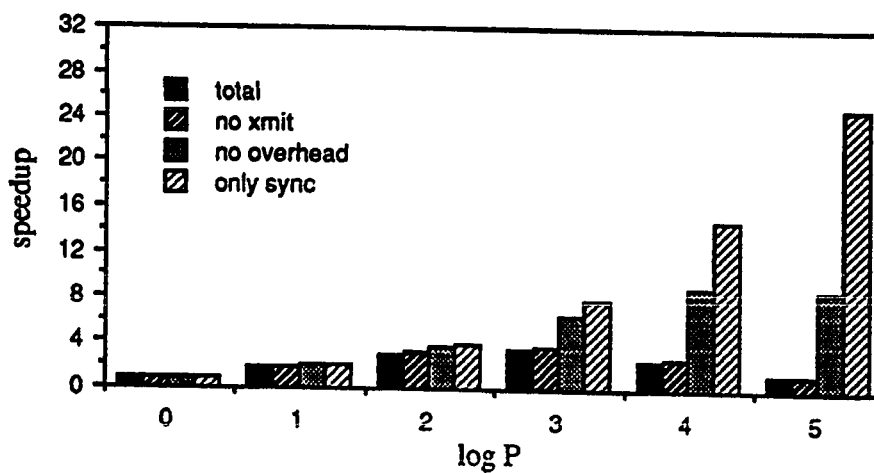


Figure 6.21: 2-D Convolution, 32x32 Matrix, Predicted Speedup.

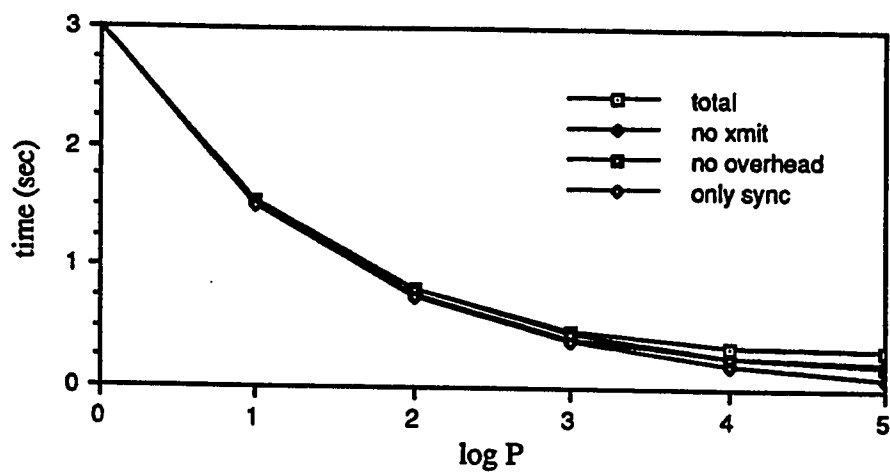


Figure 6.22: 2-D Convolution, 128x128 Matrix, Predicted Execution Time.

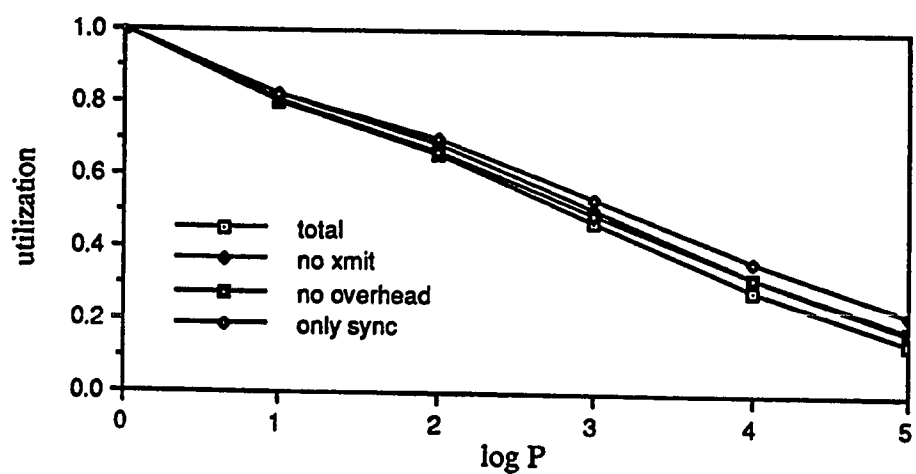


Figure 6.23: 2-D Convolution, 128x128 Matrix, Predicted Utilization.

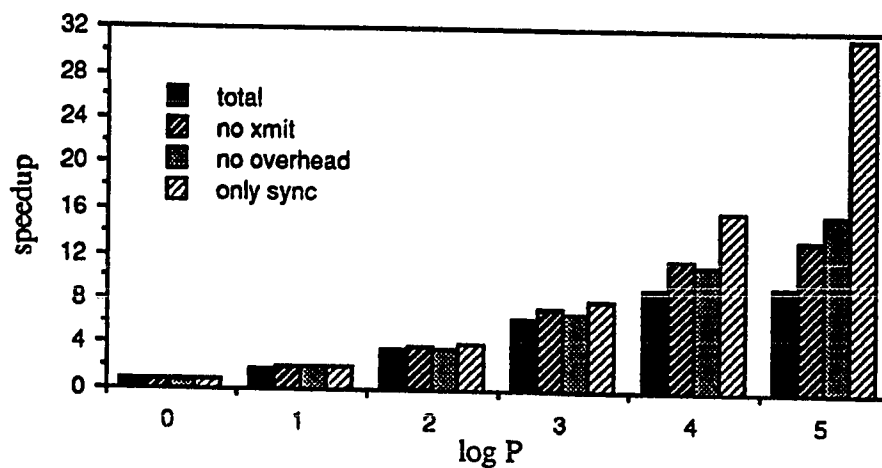


Figure 6.24: 2-D Convolution, 128x128 Matrix, Predicted Speedup.

can be realized by increasing the speed of the SP bus. Similarly, “no overhead” details the maximum gain achievable by writing the message-passing routines more efficiently.

For both problem sizes, the first figure given shows the total execution time as a function of the number of processors P . The next plots processor utilization as a function of P . We define utilization here as the fraction of total execution time attributable to operations necessary for convolution, as opposed to operations needed for synchronization and communication. The final figure shows speedup in each case.

For all practical purposes, performance of the real system (“total”) and the system with an infinitely fast bus (“no xmit”) operating on the small 32×32 data matrix peaks at the four-processor level. Maximum speedups of only about 3.5 were obtained for these cases. The elimination of software overhead, however, lead to a speedup of 9 for 16 processors. Apparently, the small data size used here makes software overhead the dominating factor, and virtually eliminates the influence of bus speed on overall performance.

Results for the more realistic 128×128 size were quite different. Here the use of longer messages made bus speed somewhat more important than software overhead through the 16 processor level, after which software overhead becomes larger. In any case, the performance of the “total” system is not much worse than either of the next two configurations. For 16 processors, “total” speedup was 9, “no xmit” speedup 11, and “no overhead” speedup 10. Speedup was virtually linear for the final case consisting of only synchronization overhead.

Of course it is unrealistic to eliminate either bus transmission or software overhead

completely. In the case of the bus, though, the total elimination of transmission time did not result in a dramatic increase in performance, and this was true even for the large problem size. Software overhead was generally more important, but only for the small data size necessitating small messages. In reality, software overhead can probably be reduced more easily than a hardware-dependent parameter such as transmission time. Assembly-coding the communication routines, for example, might decrease software overhead by a factor of two or more. This could significantly improve the performance of the system on small problems. As was noted before, however, problems this small would not typically be run in practice.

Chapter 7

Conclusions

In this thesis we have attempted to validate the accuracy and efficiency of the RPPT execution-driven approach to simulation. Validation of the RPPT involved the implementation and execution of three algorithms over a wide range of problem sizes on the RPPT as well as on the Odyssey system itself. Quantitative differences between predicted and actual performance indices served as measures of the validity of this approach. We have also attempted to judge the usefulness, from both an efficiency as well as a flexibility standpoint, of the high-level OCC parallel primitives used to program the Odyssey.

7.1 Summary of Results

The important quantitative results of this thesis, discussed in the previous chapters, are summarized below.

- 1) Simulation of sequential code running on a single processor is accurate to $\pm 2\%$ of actual execution time.
- 2) Predictions of execution time and speedup of parallel algorithms operating on problems of non-trivial size are accurate to $\pm 5\%$, and are generally much better for algorithms that are well matched to the coarse-grained requirements of the Odyssey architecture.

- 3) Unprofiled routines such as software floating-point can cause somewhat larger errors in execution time predictions. However, speedup prediction errors seem less affected by the relatively fixed influence of these routines, and are usually much more accurate.
- 4) Execution-driven simulation is efficient considering its flexibility and accuracy. Simulations of four processors, for example, exhibited slowdowns no greater than twelve relative to the corresponding simple Concurrent C executions.
- 5) The limited analysis of Section 6.4 indicates that the overall performance of the Odyssey system is improved only to a minor extent by increases in SP bus transmission speed. The performance enhancements resulting from decreased software overhead can be of greater consequence, but only for very small problem sizes.
- 6) The high-level Odyssey Concurrent C communications software makes efficient use of the Odyssey architecture for the larger problems which would typically be run on the system.

In addition to measurable quantities like efficiency and accuracy, qualitative factors such as flexibility and ease-of-use are also important considerations. For instance, simulations of new architectures are quickly and easily developed using the execution-driven approach to simulation as implemented here. The simulation of a new parallel system primarily requires the writing of an architecture model and a profiler (if one does not already exist for the processors in the system). All other RPPT system software is common to every simulation and need not be modified or rewritten. Also, since algorithms share the common form of a Concurrent C program, any algorithm may be simulated in conjunction with any existing architecture model. Finally, the RPPT

simulates the execution of a parallel algorithm by actually running that algorithm on the simulation host. Hence, once a simulation of a system has been developed, implementing new algorithms to be run and/or simulated is simply a matter of writing the necessary program for the target architecture.

7.2 Future Work

Several possibilities exist for extensions to this work. First, and probably foremost, is validation of the Odyssey simulator using additional processors. This would require access to another Odyssey board, which is not currently available but may be in the near future. It seems likely that validating the simulator against eight (or more) processors will uncover deficiencies in architecture modeling that are not significant at the four-processor level. A related consideration is the possibility of a different method of accounting for software overhead. This could eliminate the prediction errors arising from the use of average values in the simulator, but would require a minor rewrite of the ASIM procedures.

Another possibility is repeating the validation work discussed in Chapter 6 using the asynchronous message-passing routines. Rewriting the algorithms to use the *SendMessage()*/*ReceiveMessage()* primitives would not be difficult, and the validation results should serve to reinforce the accuracy of the simulation methodology. Also, the internal structure of the asynchronous routines is such that accurate (i.e., fixed) values for software overhead can be used, in contrast to the average values necessitated by the structure of the synchronous utilities. Finally, the Odyssey architecture analysis of

Section 6.4 should be extended to more fully document the performance implications of modifications to the various hardware and software components of the system.

References

- [1] W. Gass, R. Tarrant, and G. Doddington, "A Parallel Signal Processor System," in *Proceedings ICASSP*, (Tokyo), pp. 2887–2890, 1986.
- [2] Texas Instruments Incorporated, *Explorer Odyssey System User's Guide*, October 1987. Publication number (2537256-0001A).
- [3] Texas Instruments Incorporated, *Second-Generation TMS320 User's Guide*, 1987.
- [4] Texas Instruments Incorporated, *Digital Signal Processing Applications with the TMS320 Family*, 1986.
- [5] Texas Instruments Incorporated, *TMS320C25 C Compiler Reference Guide*, March 1988. Publication number (1604909-9706).
- [6] Texas Instruments Incorporated, *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide*, December 1987. Publication number (1604908-9705).
- [7] Institute of Electrical and Electronic Engineers, *NuBus—a Simple 32-Bit Backplane Bus*, 1986. P1196 Specification, Draft 2.0.
- [8] P. Penz and R. Wiggins, "Digital Signal Processor Accelerators for Neural Network Simulations," in *Proceedings of the AIP Conference on Neural Networks for Computing*, 1987.
- [9] M. D. Ingram, W. J. Foundoulis, J. R. Jump, and J. B. Sinclair, *Odyssey Concurrent C User's Manual*. Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, December 1988.

- [10] D. Mannering, *Notes on C Programming for the TMS320C25 Odyssey Board*. Speech and Image Understanding Laboratory, TI Computer Science Center, Texas Instruments Incorporated, April 1988.
- [11] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Santa Fe, NM), pp. 4–11, May 1988.
- [12] R. G. Covington, S. Dwarkadas, J. R. Jump, G. Lauderdale, S. Madala, and J. B. Sinclair, "The Efficient Simulation of Parallel Computer Systems," Tech. Rep. TR 8904, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, March 1989.
- [13] R. G. Covington, *Validation of Rice Parallel Processing Testbed Applications*. PhD thesis, Rice University, Houston TX, December 1988.
- [14] S. Madala, "Concurrent C User's Manual," Tech. Rep. TR 8701, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, January 1987.
- [15] R. G. Covington and J. R. Jump, "CSIM User's Manual," Tech. Rep. TR 8501, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, July 1985. Revised February 1986.
- [16] R. G. Covington and J. R. Jump, "CSIM 2.0 User's Manual," Tech. Rep. TR 8712, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, October 1987.
- [17] R. G. Covington, J. R. Jump, and J. B. Sinclair, "Cross-Profiling as an Efficient

- Technique in Simulating Parallel Computer Systems,” Tech. Rep. TR 8903, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, January 1989.
- [18] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*. Rockville, MD: Computer Science Press, 1984.
 - [19] E. M. Reingold and W. J. Hansen, *Data Structures*. Boston, MA: Little, Brown and Company, 1983.
 - [20] R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Reading, MA: Addison-Wesley, 1977.
 - [21] J. F. Botha and G. F. Pinder, *Fundamental Concepts in the Numerical Solution of Differential Equations*. New York: John Wiley & Sons, 1983.
 - [22] G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*. Englewood Cliffs, New Jersey: Prentice-Hall, 1977.
 - [23] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*. Cambridge: Cambridge University Press, 1988.
 - [24] S. C. Ingels, “Validation of the Rice Parallel Programming Testbed with Sorting Programs,” Master’s thesis, Rice University, Houston, TX, May 1989.
 - [25] V. Mehta, “Performance Prediction of Fast Fourier Transform Algorithms on Loosely Coupled Multiprocessors,” Master’s thesis, Rice University, Houston, TX, May 1988.