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.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0500
Validation of Rice Parallel Processing Testbed applications

Covington, Richard Glenn, Ph.D.

Rice University, 1989
RICE UNIVERSITY

VALIDATION OF RICE PARALLEL PROCESSING TESTBED APPLICATIONS

by

RICHARD GLENN COVINGTON

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

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

[Signatures]

J. Robert Jump, Professor in
Electrical and Computer Engineering,
Chairman

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

Guy T. Almes, Assistant Professor in
Computer Science

Houston, Texas
December, 1988
Validation of Rice Parallel Processing Testbed Applications
by
Richard Glenn Covington

Abstract

The Rice Parallel Processing Testbed (RPPT) is a collection of software tools for simulating the interaction of parallel programs and parallel architectures. The testbed uses a novel technique called execution-driven simulation, whereby the pseudo-concurrent execution of a parallel algorithm, augmented by profiling code, is used to drive the discrete-event simulation of a parallel architecture. This technique is intermediate between the high accuracy and low computational efficiency of instruction-level simulations and the less accurate but high efficiency statistical distribution-driven simulations, effectively combining attractive features of both of these techniques. The technique provides estimates of overall execution time, as well as more detailed performance indices such as communication vs. computation time, message passing traffic, and processor utilization. The methodology and implementation of the testbed are discussed at length and are compared with recently published related projects. The implementation has been a collective effort involving several people, and the author's contribution to the effort is outlined. Testbed predictions are given for a set of parallel numerical algorithms -- LU decomposition, eigenvalue-eigenvector determination, FFT -- simulated for a hypercube, and the predictions are validated against measurement of actual program execution on an Intel iPSC 16-node hypercube.
Acknowledgments

No words can do justice to my feelings of appreciation toward the people it has been my privilege to work with during my years at Rice. We become a part of all those we meet, and in that fact I someday hope to find consolation for leaving behind friends who cannot be replaced.

First, thanks to the members of my thesis committee. My advisor, Dr. J.R. Jump, has provided years of patient guidance and has taught me how to take a problem and cut through the distracting peripheral issues to see the heart of the matter clearly. Dr. Bart Sinclair has always had his door open for me and has often provided the crucial insight into my problems that I needed to move forward. I also thank Dr. Guy Almes for his uncompromising critical appraisal that has strengthened my presentation of this work.

Second, I’d like to thank to other members of the department. Dr. Sid Burrus and Dr. Don Johnson have offered their thoughtful and appreciated perspective whenever it was sought. Dr. J.D. Wise has been long suffering in his help with my software and hardware emergencies, even those of my own creation. Nora Quiocio, Don Schroeder, Vicki Benson, and Ida Cuellar have made dealing with departmental and university red tape almost painless, always taking a friendly and supportive rather than an adversarial role.

Also, thanks to Lance Hewitt, Kshitij Doshi, Owen Wilson, Geoff Orsak, Sarah Hammann, Sridhar Madala, Steve Leviness, Amy Jurcyk, Ted Hayes, Bill Foundoulis, Phil Arcidi, and Tony Faulise for their friendship, companionship, critical thinking, wit, and conversation.

Finally, my parents, Glenn and Mildred Covington, and sisters, Cammie Blissitte and Dolores Moore, have given me moral support and emotional reserves to draw on when I felt discouraged. When I thought that nobody else cared about my petty problems, my family helped me to keep things in perspective and to push on.
To Dr. Ron Hastings and Wanda Cain,
who years ago first introduced me
to the world of ideas.

...

One has only learnt to get the better of words
For the thing one no longer has to say, or the way in which
One is no longer disposed to say it. And so each venture
Is a new beginning, a raid on the inarticulate
With shabby equipment always deteriorating
In the general mess of imprecision of feeling

...

We shall not cease from exploration
and the end of all our exploring
will be to arrive where we started
and know the place for the first time.

-- T.S. Eliot, "Four Quartets"
# Contents

1 Introduction

1 Motivation 1
2 Overview 2
3 Implementation 3
4 Validation and Scope of the Results 5
5 Organization of Thesis 6

2 Classification of Simulation Techniques and Related Work

1 Simulation Time and Simulation Events 8
2 Levels of Hardware Modeling 9
3 Levels of Software Modeling 11
4 Combined Modeling of Hardware and Software 13
5 Related Work 17
6 The RPPT Methodology 26
7 Comparison of the RPPT with Related Work 27

3 Design and Implementation of the RPPT Environment

1 Introduction 29
2 Overview of Application Building 30
3 Concurrent C 32
4 CSIM 38
5 TPROF 40
6 ASIM 44
7 ASIMP 49
8 Miscellaneous Tools 50
9 The Runtime Environment 51
10 Building an RPPT Application 59
11 Generation of Performance Information 64
12 Personal Contribution 66

4 The TPROF Cycle Count Profiler

1 Introduction 68
2 Description of Standalone Profiling 70
3 Limitations and Possible Enhancements for Standalone Profiling 81
4 Description of Cross Profiling 86
| 5 | Limitations to Cross Profiling  | 97 |
|   | Validation                     | 107|
| 7 | Suggestions for Future Work    | 108|

| 5 | Architecture and Program Models| 109|
|   | Architecture Models            | 109|
|   | Programs                       | 117|

| 6 | Validation                     | 123|
|   | Introduction                   | 123|
|   | Scope of Validation Results    | 124|
|   | General Measurement Methodology| 125|
|   | Results                        | 136|
|   | Discussion                     | 162|

| 7 | Conclusion                     | 164|
|   | Summary                        | 164|
|   | Quality of Predictions         | 164|
|   | Scope of Applicability for     | 168|
|   | Validation Results             | 169|
|   | Enhancements to the RPPT Tools |       |
|   | References                     | 172|
|   | Appendix: Hypercube Program Listings | 176|
CHAPTER ONE

Introduction

1. Motivation

As parallel computer systems become more widely available commercially, it is important to have performance evaluation tools that can deal with the problems inherent in analyzing parallel systems in an accurate yet computationally efficient way. This availability has given impetus to the development of new tools with two important features:

- the sophistication to be able to answer interesting questions about the new systems that are beyond the scope of existing tools, and

- the ability to provide answers accurately and efficiently, with an effective user interface that makes the full power of the tools available to the user.

Interest in tools with such features has given rise to the implementation and validation of the initial version of the Rice Parallel Processing Testbed (RPPT).

The RPPT has been designed to provide special insight into several common performance analysis scenarios, such as:

- the comparative performance of a variety of algorithms and their implementations on a single machine, and

- the performance of a single program across a range of machines.
Both of these scenarios suggest many relevant questions in light of the appearance of many machines with a wide range of architectural designs, each with its own language support tools. For example, what is the right architectural organization for a given algorithm structure? Or, if I design an architecture for a specific program type, how much performance for a general mix of program types am I giving up because of the specialization? Or even more modest questions concerning the relative performance effect of a collection of variants of a single architecture, such as, will I get more performance improvement from an additional processor or a faster bus? For investigations of this sort, the RPPT provides a convenient level playing field for objective comparisons.

2. Overview

The RPPT performance prediction tools and the methodology for using them have been developed in response to a perceived inability of existing tools to strike the optimal balance between accuracy of prediction and efficiency of execution for systems of interest. The testbed is intended to occupy a niche between the high accuracy but expensive memory and cpu requirements of trace-driven and instruction level simulation techniques, and the high execution efficiency but reduced accuracy of statistical workload models. The testbed strives to produce performance predictions that are nearly as accurate as those derived from trace-driven simulations, at an execution cost that is closer to the statistical workload simulation cost.

RPPT uses a relatively new technique called execution-driven simulation. The technique interleaves the pseudo-parallel execution of a parallel program on a uniprocessor with the discrete event simulation of a parallel architecture. The program is
augmented by profiling instructions in a compile time analysis that generate resource usage estimates for the execution of the program as it runs. These estimates in turn drive the architectural simulation.

Because of the high accuracy of its workload characterization, an environment like the RPPT could very well shed light on problems like the "hotspot" controversy. That is, even though worst-case scenario statistical workloads suggest certain performance implications for an architecture, it may well be that such scenarios never arise, or only rarely arise, in real applications, so it would not be cost-effective to commit additional resources to guard against them.

3. Implementation

A typical RPPT application is put together in the following way. First, a parallel algorithm is analyzed by a timing profiler to insert special instructions into the code of the program that will generate estimates of the execution time required by small segments of the program. These estimates are similar to the resource usage statistics found in a distribution-driven simulation, only they are now generated by careful analysis of the program rather than a random number generator. The modified program is then compiled and linked with a module that simulates a parallel architecture, along with libraries and miscellaneous specifications. During execution, the specially inserted instructions produce execution time estimates, which then drive the simulation of the architectural model. Statistics on the performance of the architecture and execution time requirements of the program can be collected during the simulation.
One of the intended uses of the testbed, as mentioned above, is to consider how a single program performs on a variety of architectures. The testbed will be effective in dealing with such a scenario especially when the range of architectures to consider is relatively small. For instance, comparative predictions could be made for a given hypercube organization with 16, 32, or 64 nodes. Once the range of architectures becomes broader, more caution in the interpretation of the results may be necessary. This is because the parallel decomposition of a program whose performance is being studied over a broad range of architectures may have to change in a more than superficial way to take advantage of the changing hardware. At this point, the line that separates the performance effect due to hardware from the effect due to the software interface to the hardware or the new parallel decomposition of the program can become blurred. The conclusions drawn from such comparative studies will need to make allowance for such effects.

Programs for the RPPT are written in a special C-based parallel language that provides the writer with a library of parallel programming primitives. The choice of primitives provided has been influenced by consideration of those provided by several existing systems, such as the Intel iPSC hypercube, and the V distributed operating system. RPPT programs have access, therefore, to a representative and general set of parallel constructs. The possibility always exists that some highly customized primitives for a real program written for some special system may not translate directly into those available in the RPPT environment. With a modest amount of effort, however, versions of the new primitives could be incorporated into the testbed.
4. Validation and Scope of the Results

When introducing a prediction tool such as the RPPT, an important part of the effort is a demonstration of the accuracy of the predictions by validating them against the performance on a real system. The validation process requires both measurement of real system performance and interpretation of simulation performance. Specific methodology for each of these processes must be developed, some of which is independent of the machine or program being studied, and some of which is not. The testbed can provide more automatic collection and processing of general performance data, and at the same time can give the user sufficient direct access to the simulation execution to allow customized collection for more application-specific performance indices.

Validation is clearly limited by the available access to real parallel systems. The work presented here compares predictions to measurements for several algorithms running on an Intel iPSC hypercube. One can conclude that similar accuracy should be possible on a range of loosely-coupled systems. Validation results for other types of systems are currently being prepared by the RPPT group. Systems that will be considered in future work include a 20-node 80386-based Sequent Symmetry, a loosely coupled network of SUN workstations connected via an Ethernet and running the V distributed operating system, and a Texas Instruments Explorer system acting as a front end to an Odyssey board, a system with 4 TMS320C25 DSP's.
5. Organization of Thesis

The goals of this thesis are (1) to present the RPPT in the context of previous and current related work, showing how it offers distinct attractive performance evaluation features in comparison with other related techniques, and (2) to validate its predictions for a subset of realistic program and architecture models. The thesis is organized as follows. In Chapter 2, a taxonomy of simulation and modeling techniques is given, along with a discussion of previous and current related work. Chapters 3 and 4 describe design and implementation issues for the project. Chapter 3 presents an overview of the RPPT environment and discusses its implementation. The author’s specific contributions to the implementation effort are outlined at the end of the section. Instruction timing profilers are described in greater detail in Chapter 4. The architectural and program models used in the validation are described in Chapter 5. Validation results for the profilers and applications are given in Chapter 6. Finally, a summary, conclusions, and discussion are presented in Chapter 7. Program listings for the applications are given in the appendices.
CHAPTER TWO
Classification of Simulation Techniques and Related Work

In order to discuss the relative merits of different simulation techniques, it is convenient to classify each technique by breaking it down into several important constituent techniques, in particular its hardware and software modeling strategies. Any simulation technique can be characterized by the level at which the software and the hardware are modeled, and by how the two models interact. The technique chosen for a particular application is a function of what performance indices are desired, what level of accuracy in the indices is required, and what cost in time and resources can be tolerated. For example, some techniques are appropriate for detailed consideration of functional correctness, others for less detailed studies to determine the approximate quality of performance. The choice is also influenced by the existence of affinities between certain pairs of software-hardware modeling levels; that is, certain hardware modeling strategies work together more naturally with certain other software modeling strategies.

In this chapter, we discuss the most common choices available in modeling level detail for both hardware and software. A taxonomy of simulation techniques is given in Sections 1, 2, and 3. Section 4 describes combinations of software and hardware modeling, including the method employed by the RPPT, called execution-driven simulation. We survey a number of related implementation projects in Section 5, followed by an overview of the RPPT methodology in Section 6, and a comparison of the RPPT with related techniques in Section 7.
1. Simulation Time and Simulation Events

Both simulation time and simulation events for a given model may be termed either continuous or discrete [35]. The distinction dictates many practical details concerning how the simulation is to be implemented. For example, if the state of a system is characterized by a variable whose value may take on a continuous range of values and may change continuously with time, then the simulation model is continuous, i.e., both events and time are continuous. An example of a continuous event system is one modeled by a differential equation, since the value of the dependent variable may change across an arbitrarily small time interval. If, instead, the system state changes by a random, finite (not infinitesimal) amount at random points in time, the simulation is discrete event, continuous time. A discrete event system example is a queuing network, since the lengths of queues change by integral amounts at random points in time. If the value of simulated time can only advance in fixed time increments, the system is discrete time. A synchronous logic circuit, with time advancing at each clock cycle, would be an example of such a system. Otherwise, if simulated time may take on a range of continuous values, the system is continuous time. For continuous models, numerical solution methods such as relaxation are most appropriate, while for a discrete event system, an event queue approach is more useful. Continuous simulations normally approximate the continuous passage of time by breaking time down into a series of small fixed size increments, making the continuous system technique resemble a special case of a discrete system technique.
2. Levels of Hardware Modeling

Hardware modeling can be done at any one of four main levels of detail: (1) circuit level, (2) gate level, (3) register-transfer level, or (4) system level [30].

At the lowest level of abstraction and highest level of detail is circuit-level modeling. Here, the hardware system is reduced to a set of differential equations that relate functions of analog electrical characteristics -- voltage and current, for example -- with respect to time. The entities in this model are those points in the circuit with a distinct value of voltage and current. Equations for the circuit are formulated according to Kirchhoff's laws and others, and are solved by numerical techniques that treat the model as a continuous system. This is an example of a continuous time-continuous event model.

In the context of a digital logic circuit, such a simulation is called for when the performance criteria of interest include detailed switching behavior such as rise and fall time characteristics, or the absence or presence of voltage spikes. The accuracy of the technique is high, but so is the cost in CPU time. The workload is therefore usually limited to a small series of input signal values to the circuit. The amount of circuit detail and small timestep entailed in the approach normally limits the size of the hardware or the number of input current and voltage changes that can be considered.

At the next higher level of abstraction -- gate-level modeling -- the analog behavior of the hardware system is ignored. The system is instead reduced to a collection of logic gates and logic values. The discrete entities in such a model are gates arranged into a logic network, with tokens representing logic values that flow through the network. At
this level, events do not need to be modeled as continuous, so discrete-event techniques become appropriate. The criterion of performance is now the absence or presence of logic races and the logical correctness of the circuit. The technique suffers, however, from limitations on hardware size and workload characterization similar to those experienced in circuit level modeling. The RPPT methodology was not intended for use in circuit level and gate level simulations, so they will not be discussed further.

Register transfer-level modeling takes low-level gates and groups them into higher-level structures such as registers and multiplexers. The criterion of performance can be either one of functional correctness as in the gate-level case, or it can be one of quality of performance.

Finally, in system-level modeling, we find the highest level of abstraction and the lowest level of model detail. Register and multiplexer structures are grouped together to form functional system units such as CPU's or disks. The model treats the functional units as black boxes, whose behaviors are governed by a concise set of parameters. Entities such as CPU's and disks are considered resources and are modeled as servers, while tasks that run on the system and acquire system resources are modeled as customers. The criterion of merit is no longer functional correctness but quality of performance, which is typically measured in terms of queue lengths, queuing times, customer throughputs, and turnaround times. Performance results may be further subdivided into waiting and service time characteristics at each resource. Such queuing-theoretic performance indices make queuing models and discrete event techniques particularly appropriate for this level.
3. Levels of Software Modeling

In a simulation context, software modeling is called workload characterization. Of the possible characterizations, three are of particular interest in the context of the RPPT methodology: (1) service and waiting time distributions, (2) data and address traces, and (3) actual program representations. If a program representation is employed, the process of accounting for simulation effects during program execution may happen at the (1) instruction, or (2) basic block level. A basic block is a segment of assembly language code, delimited by labels and jumps, which will be described in detail in Chapters 3 and 4.

Distribution workloads are most appropriate for working in combination with a system-level hardware model. Resource usage times and waiting times between usages are modeled by service demand and waiting time distributions. The service and waiting time distributions represent information on resource usage behavior, abstracted from many runs of a collection of programs.

In trace workloads, the program is modeled by an instruction trace, if the hardware model includes a CPU, or a data trace, if the model includes a memory subsystem. The traces are generated either from a real program execution, or from a synthetic workload technique. The simulation proceeds by sequentially examining each instruction or datum in the trace and updating the state of the relevant system components accordingly.

Traces may be generated off-line or on-the-fly [18]. Off-line generation means that an instrumented version of a real program or a synthetic trace generator is run to completion and a trace file is generated. The hardware model is then executed by referencing
this file. On-the-fly generation means that the trace generation phase and the hardware simulation phase are interleaved, with the hardware model requesting that new segments of the trace stream be generated only at the time that they are needed. In many cases, a single trace workload may be generated off-line once and then used efficiently for simulation of multiple hardware models. For instance, this is usually the case for the simulation of a sequential program executing on a uniprocessor. In fact, this is one of the characteristics of trace simulations that make them so popular in such contexts. When multiprocessor hardware models are considered, however, there arises the possibility that the global interleaving of instructions executed across all processors in one architecture executing a given program will differ from a similar interleaving for the same program running on a second architecture. This means that an off-line trace for some program running on one architecture may have little relevance for driving a simulation for that program on a second architecture. In such a case, an on-the-fly trace generation is required. In effect, off-line trace generation decouples the trace-generation and hardware simulation phases, while on-the-fly integrates them so that the trace generation phase may be influenced by feedback from the hardware simulation phase.

Program workloads, taken together with the data that will act as input to the program, provide the most exact workload characterization. The level of information present in the program representation usually precludes a circuit level or gate level hardware model for anything more than a short program fragment. The important feature of a program workload is that a compact notation for the complete program plus all the data that the program will use is available to drive the hardware simulation model. A
program workload can be thought of as the limit of an on-the-fly trace generation scheme in which the feedback loop is only one instruction in length. Simulation state may be updated at the instruction level or at the basic block level. Instructions in the program can be either executed directly or emulated in software, or possibly a combination of both, depending on the relationship between the processor being simulated (the target) and the processor on which the simulation is executed (the host).

4. Combined Modeling of Hardware and Software

A complete simulation requires a choice of both hardware and software modeling technique, as well as some points of methodology that indicate how to link the models together. Three common combinations that have the most relevance to the RPPT are (1) distribution-driven, (2) trace-driven, (3) instruction-level, and (4) execution-driven simulations. These are each described in turn in the following sections.

4.1. Distribution-Driven Simulation

In distribution-driven simulation, hardware is modeled at the system level as a network of queues and servers, and the workload by a set of probability distributions. These distributions are typically derived from the execution of some program or a mix of programs which have been instrumented for collection of statistics on resource usage behavior. Hardware structure at a finer level of detail (the gate or register-transfer level), such as CPU or ALU pipeline stages, is typically not modeled directly, although this information may be implicitly reflected in the nature of the distribution(s) being used. Simulation results are generated in the form of performance indices such as queue
lengths, waiting and turnaround times, throughputs, and utilizations. Predictions for queue lengths which prove to be within 10% of observed behavior, and predictions for waiting times within 30%, are considered acceptable for this method [25].

4.2. Trace-Driven Simulation

In a trace-driven simulation, a few components of a single system are modeled by maintaining a small amount of state information. In effect, at least some small part of a complete hardware system is modeled at the register-transfer level -- an instruction pipeline or data cache, for example. The most successful application of this technique has been in the study of how various pipeline and cache organizations affect system performance. For pipeline simulations, the results of interest are instruction throughput and pipeline latency, and for cache simulations, hit or miss ratios as a function of cache size and organization [25]. In most sequential uniprocessor hardware system simulations, instruction and data traces are generated off-line, i.e., the pipeline and cache organization have no effect over what instruction or datum will be issued next. The size of the trace implied by a realistic application makes a trace-driven simulation of a complete application a highly memory- and cpu-intensive endeavor.

4.3. Execution-Driven Simulation

In execution-driven simulation, hardware may be modeled at either the register-transfer or system level, while the workload is a program representation. If the hardware model is at the register-transfer level and the program is examined and emulated one instruction at a time, then the execution-driven simulation resembles a trace-driven simu-
lation. Such a simulation is also called an instruction-level simulation.

A slightly more inaccurate scheme would be to model the hardware at the system level, yet not lose all the performance detail present at the register-transfer level by passing the responsibility for some hardware modeling detail to the software model. This is in principle what the RPPT does. In this scheme, the hardware model is a queuing network, with CPU's, buses, and devices in the interconnection network modeled as resources. The program representation is analyzed statically by a timing profiler that looks at the instructions and memory references in each basic block and and assigns to that block an execution cost in cycles. Flow of control between basic blocks is defined by their delimiting branches and labels. We can think of an assembly language program as a control flow graph with vertices representing basic blocks and edges representing branches. By actually running the program on the input data provided, the branching path through the graph implied by that data can be reproduced. As each basic block is encountered in the process of executing the program, the correct basic block cost is retrieved, and the actual execution of the block's instructions is interleaved with a simulation that accounts for the clock cycles that have elapsed in the model.

Also, direct execution of the workload provides highly significant benefits in the execution time of the simulation. Instruction-level or trace-driven simulations require emulation of the instructions in the workload, rather than direct execution of those instructions on the host processor. For every workload instruction, hundreds of instructions may have to be executed to emulate its effect on the simulated architecture. The ratio of emulation instructions to workload instructions is called the slowdown factor of
the method, and factors of 300 are typical. The RPPT method will generally give rise to slowdown factors of less than 10.

Having the profiler take responsibility for some register-transfer-level modeling is possible, for instance, with the modeling of programs running on an MC 68020-based system, because of the way timing information for instructions is documented [2]. The MC 68020 features an instruction cache and instruction pipeline, both on-chip. Cycle counts for each instruction are derived for three cases: best case, cache case, and worst case. These cycle count values represent the approximate number of cycles that can be assigned to the instruction were it embedded in a highly idealized instruction stream. Best case assumes both cache and pipeline operate perfectly, cache case assumes that the cache operates perfectly but that the pipeline is turned off, and worst case assumes that the cache and pipeline are both turned off. The cycle count execution cost of a program may be estimated according to any one of the three cases. While the timing estimate derived in this way may be less accurate than an exact instruction level simulation of the cache and pipeline, the saving in simulation execution time can be expected to make the technique attractive and competitive with other more accurate and expensive techniques. Not all chips provide such detailed cycle count documentation as the MC 68020, but simpler timing documentation usually reflects simpler on-chip architecture.

Some advantages of the execution-driven approach are therefore

- compactness of workload representation with respect to off-line traces,
- confidence that the workload reflects behavior of a real application running on real data,
- significant performance improvement over emulation techniques, and
- performance prediction accuracy comparable to much more expensive techniques.

5. Related Work

Other projects that employ the notion of execution-driven simulation and that otherwise have particular relevance to the RPPT project are described below. The title of each section indicates the project name, the first author, and the site at which the work was done.

5.1. MPSIM, Axelrod, Lawrence Livermore

MPSIM is a simulation package that runs on the Cray-1 under CTSS (Cray Time Sharing System) [9]. It is divided into MPSIM-1, which simulates the software model by executing a FORTRAN program and generating a trace of Cray-1 instructions, and MPSIM-2, which simulates the hardware model by interpreting the Cray-1 instruction trace in the context of the target architecture model, the S-1 MkIIa. The FORTRAN environment is augmented by an MPSIM-1 library of functions such as fork and join. Results generated include execution speed, speedup, and cache effectiveness. Results are reported for four algorithms.

MPSIM can be characterized as a execution-driven simulation with hardware modeled at the register-transfer level and simulation updates occurring after every instruction. Each Cray-1 instruction generated by MPSIM-1 is first actually executed in order to achieve the correct logical effect of the program. The instruction is then exam-
ined by MPSIM-2 in order to emulate the correct target architecture behavior. Here, special attention is given to ABOX (the arithmetic pipeline) and IBOX (instruction fetch and decode pipeline) effects. Target architecture emulation typically entails large slowdown factors.

In comparison to the RPPT, MPSIM may prove harder to develop into a full-fledged testbed. The MPSIM-2 module for simulating the S-1 hardware is a complex, highly customized program. The program depends on careful comparison of both low (register-transfer) level hardware features of both the Cray-1 and the S-1. Consideration of a new architecture would require a new MPSIM-2 to be written essentially from scratch, and it may be infeasible to do this at all if the target architecture is sufficiently different from the Cray-1. For example, the implementation makes assumptions such as (1) address arithmetic will usually be overlapped with memory references and its performance effect may therefore be ignored, (2) the instruction cache behavior will have little performance impact, and (3) ABOX delays that result in data dependencies between operations may be ignored. In addition, it is assumed that unmodeled features such as conditional branch prediction and the operand queue that partially decouples the ABOX and IBOX always work perfectly. Finally, the modeling of operating system costs is simplistic, with one cycle being charged for each system call (fork, join, etc.). Validation of performance predictions against measurement of programs running on a real system was not possible, since the S-1 was still in the design stage at the time of the research.
5.2. EUCLID, Butler, Renssalaer

EUCLID is a software simulation package that allows for the specification of a multiprocessor architecture and a program to run on that architecture [10]. For a complete application, the user must specify four modules to EUCLID: (1) the architecture file, which describes how components in the architecture (processors, memories, data paths) are linked together, (2) the priority memory access file, which specifies processor instruction execution rates and memory access bandwidths, (3) the user-defined instruction set file, which describes each instruction in the target processor by a Pascal subroutine (each processor in the architecture may have its own instruction set), and (4) the object code file of the program to be executed. The program to be executed must be specified in terms of the language Multisoft, which provides the user with a small set of built-in instructions plus an interface for accessing those instructions that the user has defined in the instruction set file.

In a EUCLID simulation, hardware is modeled at the register-transfer level, and simulation updates occur at each clock cycle. The potential for modeling architecture behavior in an exact cycle-by-cycle way is therefore very high. Target architecture instructions are emulated by the Pascal programs provided in the instruction set file, rather than being directly executed, which clearly implies an enormous slowdown factor. Other drawbacks to the system are lack of HLL support for application programs and a voluminous amount of specification detail for a reasonably complex architecture. Still, if HLL support can be provided, and if the other drawbacks can be tolerated, EUCLID offers a solid foundation on which to build a complete architecture-application testbed.
5.3. Trace Interleaving, Dubois and Briggs, Rice and USC

Trace interleaving is a technique for taking the trace-driven simulation approach, which has worked so well in uniprocessor cases, and adapting it for reliable behavior in a wide range of multiprocessor cases [18]. The method is called trace interleaving because it works by identifying certain global events from each processor's instruction stream and interleaving these in the appropriate order to generate a correct global event trace. Examples of such global events include (1) accesses to shared data, (2) process creation and destruction, and (3) message passing through the network. There are three kinds of trace interleaving permitted -- (1) rank, (2) virtual time, and (3) physical time -- and two kinds of trace generation -- (1) off-line (appropriate only for rank or virtual time interleaving) and (2) on-the-fly.

The simulation environment is divided up into several modules: (1) a trace generator, (2) a trace scheduler, and (3) the architecture simulation module. A library of coroutine support functions (suspend, activate, transfer) is provided to assist in the implementation of the modules. The user must write the architecture simulation from scratch, including such structures as an event queue. The following assumptions are made regarding the computation model. (1) The parallel program is written in terms of explicit parallel constructs, rather than having potential parallelism detected by the compiler. Standard language constructs are provided for both shared (lock, unlock) and distributed (send, receive) memory systems. (2) The hardware cache is large enough to hold the program's working set. If this assumption is violated, the technique must be modified to model instruction fetches. (3) Jobs are self-scheduled, i.e., through programming
language constructs, and scheduling may be either static or dynamic. Completed applications include static and dynamic quicksort and a PDE grid relaxation solution, for a bus-oriented multiprocessor with local cache.

5.4. CARA, Flynn, Stanford

CARA (Compiler-Aided Research into Architectures), also called the Computer Architect's Workbench, is a tool for studying the performance impact of and establishing tradeoffs between such processor features as memory-processor traffic, opcode decoding complexity, number of general purpose registers, and data and instruction cache size and organization [19,34]. In particular it has not been used for predicting the overall cost of program executions in terms of total instruction cycles. It has also been used to draw some interesting comparisons between the performance of RISC and CISC machines. The principal software components are (1) the U-code system, (2) basic block analyzers, (3) cache simulators, and (4) benchmarks. The architecture is specified by a choice of basic block analyzer and values of parameters to analyzer.

The principal steps in the methodology are as follows. (1) The application program is compiled into U-code representation. (2) U-code is run through a basic block analyzer to produce a modified U-code representation, with special trace routines inserted at the start of each basic block. The analyzer also produces a file of basic block tables which describe how the blocks are laid out in memory, specifically, how instructions are allocated to cache lines. A new basic block analyzer must be written in order to introduce a new architectural type into the workbench environment. Presently there are three analyzers available, one for each general processor type. Each analyzer takes parameters
that allow specification of a family of variations within that architectural type. (3) Executable code for the host is generated from the modified U-code, and linked with a cache simulator. (4) The application is executed. (5) Whenever a trace routine is called, the basic block tables are examined to perform a line-by-line simulation of the basic block for each target architecture. A completed run of the workbench produces three types of results: (1) files of the results for each benchmark, (2) averaged results for all benchmarks, and (3) a command file for plotting results.

The workbench is an execution-driven simulator with hardware modeled at the register-transfer level. Simulation state is updated at every basic block. Basic blocks are executed directly in order to implement the program, and are then emulated to determine target architecture behavior.

The current limitations of the system are that (1) it does not handle instruction pipeline effects (implying an inability to provide estimates of overall cycle count costs), and (2) there is no provision for multiprocessor architectures. Work is underway, however, to address both of these limitations.

5.5. PSIMUL, So, IBM

So, et al., describe PSIMUL, developed at IBM T.J. Watson Research Center [38]. The important software modules in the project include the following: (1) VM/SP, a uniprocessor operating system for S/370 systems; (2) VM/EPEX, an environment that allows for the parallel execution of a single VM/SP application; (3) SEMUL, a simulator and trace facility for S/370 systems, that produces (a) a summary of CPU performance
statistics, and (b) a trace of instructions and memory references; and (4) PSIMUL, a parallel extension to SEMUL, that adds some functionality to SEMUL, such as a SYNMK (synchronization mark) call, and provides a general purpose cache simulation model which is driven by the trace generation step. A user's FORTRAN program is parallelized and executed under the control of SEMUL. SEMUL generates tracing and timestamping or synchronization information, which is then used to drive a user-written hardware simulation or cache simulation. Because of VM/EPEX, the trace generation and simulation themselves may be executed in parallel.

5.6. Cray X-MP Simulator, Williams, Los Alamos

Williams, et al., describe a Cray X-MP simulator developed at Los Alamos National Laboratories [42], which is possibly the first implementation of the execution-driven simulation approach. The simulator runs on one processor of a Cray X-MP/24, a two-processor machine, so the simulator itself is not run as a parallel program. The simulator gives speedup estimates for a hypothetical N-processor Cray X-MP. The Cray X-MP is a 1-4 processor shared memory multiprocessor, but simulation results are reported for N = 1, 2, 4, 8, 16, and 32. Predictions are validated against programs running on 2, 3, or 4 processors, and agreement between simulation and measurement is within 1%. Parallel time is the time the last processor in the simulation finishes. Sequential time is estimated by accumulating only time spent in the user code, and not in the multitasking library. Speedup is parallel time over sequential time, and efficiency is speedup over N.

The methodology makes the following assumptions. The host architecture must be a multiprocessor with N small. The target architecture must be a "large N" version of the
destination architecture. Memory and I/O contention must be negligible.

5.7. SIMON, Fujimoto, Berkeley

SIMON [20] is a simulation environment that also combines the discrete event simulation of an architecture with the instrumented pseudo-concurrent execution of a program serving as the simulation workload. SIMON may have been the first project to introduce the idea of self-profiling code to drive the architectural simulation. More recent work that expands on the original project [21,22] includes a provision for profiling code for processors (the targets) other than the one on which the profiled program is to be executed (the host). In this process, a high-level language program is compiled to the target assembly language, instrumented for timing measurement, "decompiled" to a high-level language, and recompiled into host assembly language. The RPPT features a simpler scheme called cross profiling which should be almost as accurate, and which will be described in Chapters 3 and 4.

5.8. Other Projects

SCHEDULE [17], while not a performance analysis package, offers design ideas for the writing of parallel algorithms which may be useful for the RPPT environment. SCHEDULE is a package for presenting a single interface between a user's FORTRAN program and a variety of shared memory architectures. The internals of the SCHEDULE package may change from system to system, but the application FORTRAN program remains the same. One concern in using the RPPT to evaluate the performance of a particular implementation of an algorithm on different architectures is that the efficient
implementation of the algorithm in general depends on the specific features of the architecture. Performance effects must then be attributed to both a changing architecture and a changing application, making the relationship between architecture and program performance harder to infer. Some features of the SCHEDULE interface may provide insight into how to write RPPT application programs in an architecturally independent way.

**SIMPLE/CARE** [15] is a set of simulation tools for the study of expert systems. SIMPLE is an event-based simulation system, and CARE is a computer array emulator that runs on SIMPLE. Architectural components -- a LISP processor, for instance -- may be specified in terms of CARE constructs, producing a discrete-event simulation specification for that component for observing functional correctness and execution time requirements. The workload is an application language program -- a series of queries, for example -- which gets translated into a series of events that affect the state of the simulation. The system is intended for use in prototyping studies for expert system hardware.

**SiGLe** [8], Simulator at Global Level, is a system that allows the user to specify an architecture, a distributed algorithm, and an algorithm mapping scheme. Architectures are envisaged to be networks of processors and buses. Algorithms are specified in any language with standard parallel primitive calls for expressing interprocess communication. The thrust of the work is toward graphical interfaces for architectural model specification and knowledge-based support for load-balancing of user processes on the specified nodes of the architecture. The implementation is not complete.
6. The RPPT Methodology

A detailed explanation of the RPPT implementation is given in Chapters 3 and 4. In order to compare the methodology with the related work, we give a brief sketch of the important steps in the RPPT methodology.

- A parallel program is written in a language we provide called Concurrent C.

- The program is modified by several preprocessors, including an instruction profiler that assigns a cycle count cost to each basic block. Depending on the complexity of the profiler used, it is possible to assign costs that are derived from a target architecture representation of the block, rather than a possibly inaccurate count based on the host architecture representation.

- The program is compiled and entered into a program library.

- An architecture specification is written in a discrete event simulation language called CSIM.

- The architecture model is compiled and entered into a library. A user selects a program module and an architecture module, links them together into an executable image, specifies a set of command line arguments, and executes the simulation.

When a basic block in a profiled program is executed, first the code inserted by the timing profiler increments a count of the total number of cycles executed by the number of cycles in the basic block. The basic block is then directly executed. At certain points in the execution, the architecture model simulator is invoked to account for the occurrence of architectural activity. This activity uses the accumulated cycle count to
correctly model the cost of the program’s execution.

Currently, the RPPT testbed is limited to either (1) loosely coupled systems which feature message passing communication and no shared memory, or (2) tightly-coupled shared-memory systems that do not allow shared data to reside in cache. The restriction with respect to data in cache is a function of the current status of timing profilers. A relaxation of this restriction will be possible, at the cost of some of the execution profiling efficiency.

7. Comparison of the RPPT with Related Work

Two characteristics of the RPPT method differentiate the project from the related projects discussed above. First, the method of accounting for program execution cost at the basic block level offers a significant advantage in terms of slowdown factors. Many of the schemes described require the software workload to be simulated at the instruction level, resulting in slowdown factors of several orders of magnitude [38]. The profilers used to generate estimates for execution cost of a program in the RPPT environment generally result in a slowdown of an order of magnitude or less, although modifications to the profiling scheme to take into account cache effects will result in greater slowdowns. SIMON [20] also uses self-profiling code inserted at basic block boundaries, although it may lose some opportunity for optimization by not reducing blocks when possible. The RPPT profilers also provide a novel scheme for incorporating timing information from the target code into the basic blocks of the host code. This scheme, called cross profiling, is described in detail in Chapter 4. SIMON achieves the same effect by a process called decompiling, which was described earlier in Section 5.7. The decompilation
scheme is exact, in the sense that there is no potential for block mismatches, as is the case for cross profiling. Its main drawback, however, is that a new decompiler must be generated from scratch for each new target. In contrast, once the timing analysis modules for a target machine has been prepared in the RPPT cross profiling scheme, little additional work is required, since the block matching phase is target-independent.

Another feature that differentiates the RPPT from other related projects is its significant software support for straightforward creation of user-defined program and architecture modules for ongoing enhancement of the testbed libraries. Programs can be written in Concurrent C which then map easily onto several multiprocessor environments, including the Intel iPSC hypercube, the V system running on a LAN of SUN workstations, the Sequent Symmetry, and the TI Explorer/Odyssey. For many of the projects discussed, each simulation application is carefully coded and instrumented anew, or its range of applicability is restricted to a narrow class of architectures and programs. Interfacing between program and architectural modules in the RPPT has been designed carefully so that arbitrary combinations between modules is straightforward.
CHAPTER THREE

Design and Implementation of
the RPPT Environment

1. Introduction

The RPPT environment has been designed with several performance and useability
goals in mind. First, as mentioned in Chapter 2, with respect to such techniques as
trace-driven or instruction level simulation, the RPPT should result in simulations that
produce accurate predictions, yet make efficient use of time and memory. Second, to
fulfill its function as a testbed, the organization should allow modular specification of all
principal model components, and these components should comprise a wide spectrum of
application program and architecture types. Third, the implementation of the simulation
software support should itself be kept modular to allow selective refinement and upgrad-
ing of individual simulation components (e.g., replacement of one timing profiler with a
newer, more accurate one). Some of the attractive features of such an implementation
are that program and architecture specifications can be specified and compiled independ-
dently, making it possible to quickly link modules into a complete application. Also,
components such as the parallel algorithm and the architectural simulation may be exe-
cuted in isolation, for purposes of testing and debugging, before being linked together
into a complete RPPT application.

This chapter describes the implementation of the RPPT environment, with attention
to how performance and design goals are achieved, and how the various layers of
software interact. Section 2 gives an overview of how RPPT tools are used to build an application simulation. The next few sections, 3 through 8, summarize the features of each of the RPPT's principal constituent software packages. Once the principal tools have been described, Section 9 gives a more detailed description of how applications are specified. Section 10 describes how at runtime the different software tools are layered together to support the execution of a complete application, and Section 11 explains how performance information is generated and utilized. Finally, in Section 12, the author's personal contribution to the RPPT design and implementation is outlined.

2. Overview of Application Building

As outlined in Chapter 2, an RPPT application requires specification of a parallel algorithm, a simulation model of an architecture, and other miscellaneous specifications such as information describing mapping of objects (processes, global data) to specific processors, and descriptions of software overhead to be charged against system calls on the simulated architecture. The RPPT provides software packages to aid in this specification. Additional software is provided for instruction time profiling, for window-based user interfaces used for application setup and application results display, and for runtime support of the RPPT environment. A detailed block diagram is given in Figure 3.1. As seen in this figure, an application program is written in Concurrent C, and the source code passes through several steps for preprocessing and compilation, producing an object module that is loaded into a library of program modules. (Note that a single machine-independent program may be profiled by multiple timing profilers, depending on the machine for which timing information is desired.) Similarly, an architectural
Figure 3.1
RPPT block diagram
simulation specification, written in CSIM, is preprocessed and compiled into an object module loaded into a library of architecture modules. To put together an RPPT application, the user (not necessarily the same person as the program or architectural module writer) interacts with a Suntools-based program "simtool". This program prompts the user for information necessary to link up and execute a complete application. During execution, there are a series of Suntools-based programs that filter statistical and debugging information from the running program and present the user with a more structured, high-level view of the application's dynamic behavior. In the next few sections, each major RPPT tool is described in turn.

3. Concurrent C

Concurrent C (CC) is a parallel programming language based on the C language [31]. It serves as the specification language for all parallel programs in the RPPT, and provides process control features for the runtime support of RPPT applications. Not a new language defined by its own compiler, it is rather a collection of predefined data-types and a library of subroutines for operating on these data types, plus other subroutines for runtime support of the CC environment, all of which work together within a standard C language environment. Both user and system source code is compiled and linked by the C compiler to create a CC executable image. CC began as an outgrowth of CSIM 1.0 (described below) by taking the independent process control features of CSIM and organizing them into a separate package. It was then augmented by new code for such features as message passing. The important features of CC are lightweight process control (create, activate, suspend, fork, join), message passing (send, receive), and
mutual exclusion via semaphores (p, v).

CC programs run either in standalone mode or in RPPT mode. Standalone mode means that the program is run with no RPPT overhead such as architectural model interaction or execution timing profiling. This mode is appropriate for merely demonstrating the program's functional correctness, rather than drawing conclusions about the program's quality of performance on some underlying architecture. In RPPT mode, the program execution drives the simulation of an architectural model. In this mode, functional correctness is assumed to have already been established, and quality of performance is the focus of attention.

CC provides a distinction between logical nodes and physical nodes. A CC program is written with respect to an general architecture characterized by an arbitrary number of abstract processors, which are called logical nodes. Each CC process is associated upon creation with some logical node. A physical node, in contrast, corresponds to a physical processor in the simulation model of a specific architecture. As part of the simulation specification process, logical nodes are mapped to physical nodes. Logical nodes are therefore purely conveniences of notation that facilitate the specification of a CC program independently of the details of a specific architectural model. A correspondence or mapping between logical and physical nodes can then be established separately from the CC program proper. For a standalone CC program, the identity of physical nodes is irrelevant, since there are no performance effects to be considered for such a program. In contrast, when the program is run as part of an RPPT application, i.e., in RPPT mode, the program will now interact with the architecture specification. In this
case, the assignment will have an impact on the model's performance.

For example, the process creation subroutine "Create()" expects a node ID as a parameter, which indicates the logical node to which the new process is assigned. If the program should ideally schedule some arbitrarily large group of processes each to their own node, then the program may assume that there are an arbitrarily large number of logical nodes and assign each process to its own logical node. When the RPPT application is linked together, the user may specify the desired mapping, such as a wraparound assignment: \( p_i = l_i \mod N \), where \( l_i \) is the value of the \( i \)th logical node, \( p_i \) is the value of the \( i \)th physical node, and \( N \) is the number of physical nodes. When the RPPT application is executed, the CC environment dynamically translates references to logical nodes into the specified physical nodes.

The principal components of the runtime CC environment are a driver process and a set of lists for ready to run processes. A process in CC is similar to a thread in C++ [39] or a lightweight process in some recent Unix implementations [3]. As currently implemented on uniprocessor systems, it supports only pseudo-concurrency, but extending the package to provide true concurrency in a shared-memory multiprocessing environment, such as the Sequent Symmetry, should be straightforward. The main process, defined by the C entry point subroutine "main()", is provided by the CC library. This process performs runtime initialization, including creation of the driver process, and then transfers control to the driver process. The driver is responsible for maintaining ready lists of ready-to-run process descriptors, created either by the system or by the CC client. There is one ready list maintained for each physical node in the underlying architecture, plus
one extra list called the real or main ready list. In a standalone CC application, all processes normally use only the main ready list. The specification of a CC program must include the definition of a subroutine named "UserMain()". A process descriptor associated with "UserMain()" is automatically placed on the ready list during initialization.

CC processes may become conditionally and implicitly suspended as a result of a call to "ReceiveMessage()" or "P()" for some semaphore, or they may suspend themselves explicitly by a call to "Suspend()". At a suspension point, control is passed from the suspending process back to the driver, which then selects the next process to run from some ready list and passes control to the chosen process. This activity continues until the driver finds no ready processes, at which point the driver returns control to the main process, and the execution terminates. The current implementation of the CC process control support provides only a coroutine-like transfer policy for passing the single thread of control from one active process to the next. That is, processes run until either an implicit or explicit call to "Suspend()" is encountered. For purposes of verifying the logical correctness of a CC standalone program in the absence of simulated time, such a process control mechanism is acceptable. In an RPPT simulation, however, this policy can have a dramatic effect on the amount of synchronization time experienced by some processes. This problem will be addressed again in the discussion on validation in Chapter 6. A CC implementation that provided true scheduling of active processes rather than just coroutine transfers would also provide the ability to model important features found in real systems such as interrupts. The current process control implementation also makes it possible to write a program that deadlocks. This is not necessarily a CC shortcoming, but
merely something that the user should keep in mind when writing and debugging his programs.

Although CC resembles and has in fact been influenced by the parallel programming primitives provided by the Intel iPSC hypercube message passing and process control libraries, some differences between CC programs and hypercube programs should be kept in mind when writing RPPT application programs.

- Process Creation

Since processes in CC are lightweight, their creation entails little overhead, so applications may spawn them off liberally. In addition, every call to the ASIM library subroutine "AsimSendMessage()" spawns off one process for messages between processes on different physical nodes. On the iPSC hypercube, as on most similar systems, processes are heavyweight items, and creating new ones is expensive. The system also limits the number of simultaneously active processes per node to around 10 or 20. Therefore, if a CC program is being written with the ultimate intent of running a similar version on the iPSC hypercube or most any other real system, process creation should be used with restraint. A typical arrangement is to create a single driver process for each node, and have that process coordinate all activity for that node from within that one process.

- Scope of Variables and Address Spaces

In CC, local variables defined inside the body of some procedure are private to each instance of that procedure. Global variables are not private and are seen by all instances of all procedures. This is because a CC application runs in a single address space. A procedure that runs on a node of the iPSC hypercube, in contrast, runs in an address
space that is private to that node. Private copies of global variables defined in the source code are replicated on each node and are not kept coherent with one another. This is consistent with the fact that the system is loosely coupled, with no shared memory. All communication takes place via message passing. When writing CC programs, usage of global variables for on-the-fly exchange of information between processes should be avoided.

An example CC program to start up two processes that send exchange messages and terminate is as follows. The entry point routine "UserMain()" looks like this.

```c
UserMain() {
    PROCDESC *pd1,*pd2;

    pd1 = CreateChild(0,a,"%d",id);
    pd2 = CreateChild(1,b,"%d%u",id,pd1);
    Activate(pd1);
    Activate(pd2);
    Join();
}
```

The processes "a()" and "b()" look like this.
a(id)
int id; {
    int lp,tp,dp;
    char buf[80];
    ReceiveMessage(BLOCK,buf,1,&lp,&tp,&dp);
}

b(id,pd)
int id;
PROCDESC *pd; {
    char buf[80];
    SendMessage(pd,sizeof(buf),buf,1);
}

CC programs are written with explicit parallelism of relatively coarse granularity. In this example, the main process creates two child processes and then activates them. It then synchronizes by waiting until they have both terminated. Process "a" receives a message from process "b", blocking its execution if necessary until the message is available. Process "b" correspondingly sends to process "a".

4. CSIM

CSIM [11,12] is a discrete-event process-interaction simulation package that runs as a software layer on top of CC. Like CC, it is a collection of predefined datatypes and subroutines that augment both the C language and CC. CSIM starts with the functionality of the CC environment and adds simulation time, controlled by an event list, and process synchronization control, supported by data structures such as state variables, conditions, and semaphores. The event list allows the CSIM application to observe the pas-
sage of simulated time, and the state variable and condition features provide the application with a more general synchronization capability than is available in CC alone. CSIM also facilitates the collection and display of statistics generated by the execution of a simulation application, although these features will ultimately be moved entirely a new RPPT support program called STATTOOL, discussed in Section 8.

CSIM allows for the specification of a simulation application in terms of independent processes. A CSIM process is implemented as a CC process with a small amount of additional CSIM-specific descriptor information. Although the CC driver process is responsible for taking process descriptors off the various ready lists and transferring control to them, the CSIM runtime environment still requires its own driver which mediates between processes on the event list and those on the ready lists.

As an example architectural simulation specification, consider a simple bus. In CSIM, a bus can be modeled by either a SEMAPHORE or QUEUE data structure. A SEMAPHORE is a lower-level, more unstructured construct, but it allows the application writer more freedom in specifying such features as custom queuing disciplines. Consider the code necessary to allow a CSIM process to acquire the resource "bus" for "service-time" time units. For a SEMAPHORE model of the bus, the CSIM code looks like this.
SEMAPHORE bus;

proc() {
    ...
    WaitSemaphore(&bus);
    DelayProcess(servicetime);
    SignalSemaphore(&bus);
    ...
}

Here, the bus is modeled as a resource and the workload or program is modeled as a customer. The customer acquires the resource, delays for a given amount of simulated time, then releases the resource. For a QUEUE, the code is even simpler.

QUEUE {fifo} bus;

proc() {
    WaitQueue(&bus,servicetime);
}

In this example, the resource is acquired, held, and released, all within one procedure call. More complex models, such as multistage interconnection networks, are also straightforwardly specified in CSIM.

5. TPROF

TPROF is a dynamic instruction cycle count profiler for assembly language programs. Application programs running in RPPT simulation are analyzed by one of a collection of TPROF profilers which have been developed for the testbed. Profilers insert code into the program that provides the architecture model during simulation execution with estimates of the real execution time that the program would require on the real system. This section presents an overview of the profiler. Chapter 4 discusses the design
and implementation of the profilers in detail. Validation results for several profiler implementations are included in Chapter 6.

5.1. TPROF vs. BB

The TPROF design is based on the BB (basic block) instruction count profiler, developed at Bell Labs [40]. TPROF differs from BB, however, in significant ways because of differences in intended usage. BB is concerned, not with absolute execution time estimation, but rather generation of cheap dynamic instruction mixes. TPROF, in contrast, attempts to accurately estimate the execution time of each basic block in the program. The BB profiler views a program to be profiled as a collection of code segments called basic blocks. A basic block is a series of consecutive assembly language instructions that are delimited by labels and branches. BB analysis takes place in several steps. At compile time, BB

- generates an assembly language representation of the C program,
- performs a basic block analysis of the assembly language, and
- inserts counting instructions at the top of each basic block.

At runtime, as each counting instruction is executed, an internal table for dynamic counts is incremented. As the program terminates, the table of counts is written to an external file. In a post-mortem analysis step, BB combines the table of counts with symbol table information collected from the original C program and creates a new file which contains the program, with each line prepended by a count of how many times the line was executed during the program run.
TPROF also takes a C program, generates an assembly language file, and performs basic block analysis. Instead of assigning a unit count to each instruction, however, as is done in BB, TPROF breaks down each instruction into an opcode and operands to arrive at an accurate cycle count for the instruction, then sums up cycle count totals for each basic block. Before inserting counting instructions, TPROF performs a basic block reduction step by examining the complete adjacency graph of the basic blocks and coalescing certain basic blocks into single aggregate blocks. This process is described in more detail in Chapter 4.

5.2. Standalone vs. Cross Profiling

Another novel TPROF feature is the ability to do cross profiling. This is a generalization of the ordinary profiler in which a C program is profiled to determine the cycle count cost of each basic block according to the assembly language representation of the program for one machine (the target), while the execution of the profiled code is carried out in the assembly language of a second machine (the destination). Cross profiling is necessary in the RPPT because of the direct program execution implied by the execution-driven simulation technique. Since instructions are executed directly in this technique, instead of being emulated, the use of an ordinary profiling scheme would limit the processors which could be considered in the simulated architecture to be almost identical to the one on which the simulation happened to be executing.

Cross profiling consists of several steps. First, the C program is "marked". Marks are special C source code statements inserted into the original source code that help to bind segments of assembly language -- from which machine-specific timing information
is derived -- to its parent C source code construct -- which is a machine-independent representation of the algorithm. Second, the marked source program is manipulated and compiled to generate an assembly language listing for both the host and target machines. In this step, the markers are converted into comments embedded in the assembly language. Third, the two marked assembly language listings (one for the host and one for the target) are analyzed by TPROF, which performs basic block analysis, timing analysis (target only), and block reduction on the files, counting cycles in each block in the target file, and identifying line numbers for the beginnings of blocks in the host file. Fourth, a merge step matches up blocks between the two marked assembly language files. This is done by matching marks across the two blocks. For example, a given mark $X$ will fall into basic block $i$ in the host listing and basic block $j$ in the target listing. The match step infers that host block $i$ and target block $j$ correspond to one another, and a match is recorded for the two blocks. Finally, after all possible matches between host and target blocks are established, counting instructions are inserted at the head of basic blocks in the host code, with counts derived from the equivalent target block. Mismatches can occur, and a detailed discussion of how these situations are handled is given in Chapter 4.

There is much about the structure of each TPROF implementation that generalizes from one version to the next. Still, a given implementation is intrinsically tailored to the specifics of a given instruction set. A major component of the effort required to bring a new architecture into the testbed is involved with the development and validation of an accurate TPROF profiler for that system's instruction set. Therefore, future work on TPROF implementations for the RPPT should give attention to making these profilers
more structured, modular, and easier to use and interpret.

6. ASIM

The ASIM (Architectural Simulation) library is a collection of modified versions of most of the important procedures in the CC library, such as "SendMessage()" and "Create()", that are called by CC programs. For each such procedure, the ASIM library features a corresponding version: "AsimSendMessage()" and "AsimCreate()", for example.

There are two important reasons for providing these modified calls. First, a standalone CC program runs as a collection of CC processes, while RPPT applications run as a collection of CSIM processes. The two process types are implemented with different process descriptors, so CC procedures are designed to manipulate CC process descriptors, and ASIM procedures to manipulate CSIM process descriptors.

The second reason for the modifications found in the ASIM library is the need to model simulation and timing activity that is not present in a standalone CC program. This activity includes

- the performance impact of interprocessor communication,
- the elapsed execution time as estimated by the self-profiling instructions inserted into the program by TPROF, and
- the software overhead penalties experienced by the real program running on the real system during calls to operating system primitives.
All of these simulation effects are modeled by introducing subroutine calls into the ASIM procedures. The three effects described above are modeled by calls to "UserSend()", "TimeAccount()", and "SoftwareAccount()", respectively. The modified versions of the procedures are essentially copies of the corresponding CC procedure, with these special subroutine calls added. Information exchange and software overhead effects are described later in this section. Discussion of execution time modeling is deferred until Section 11.

6.1. Modeling Information Exchange

Whenever an implied data transfer takes place in the simulation, the simulation effect of the transfer is handled by a call to "UserSend()". For example, message passing in CC programs written to resemble iPSC hypercube applications is handled by calls to the procedures "SendMessage()" and "ReceiveMessage()". When the Concurrent C program is run through the ASIMP preprocessor in preparation for RPPT execution, calls to "SendMessage()" and "ReceiveMessage()" are translated into calls to "AsimSendMessage()" and "AsimReceiveMessage()". "UserSend()" is called from within "AsimSendMessage()" at that point at which the effect of the architecture on the execution of the call must be simulated.

Figure 3.2 gives a sketch of the code required for "AsimSendMessage()". In the case of "SendMessage()", it is not enough to correctly model simulation of the information transfer by inserting a call to "UserSend()". A small rearrangement of the code is also necessary to correctly model the interaction of the message passing with the underlying architecture. The problem is that a "UserSend()" must remain active while the
SendMessage() {
    buffer message
    place in process queue
    (normal send)
}

AsimSendMessage() {
    TimeAccount();
    if (src != dst)    Fork(  AuxSendMessage  );
    else normal send;
    SoftwareAccount();
}

AuxSendMessage() {
    UserSend();
    normal send;
}

SoftwareAccount(src,dst,bytes) {
    DelayProcess(k1*bytes+k2);
}

Figure 3.2
difference between CC and ASIM calls
message traverses whatever interconnection device the architecture provides, terminating or returning to the caller only after the message arrives at its destination. However, a process sending a message from one node to another should not be delayed for the entire duration of the message delivery. In an architectural model based on a multistage interconnection network, the sending process would experience an unreasonable delay before it could continue execution. For this reason, simulation of a remote send requires that the sending process fork off an auxiliary process to actually deliver the message through the architecture, allowing the sending process to continue execution, even though the message may not arrive at the destination for some time. The auxiliary process then takes responsibility for simulating the movement of the message through the network. Once "UserSend()" returns, a signal is generated for any process waiting on a receive of that message that it may activate and continue execution.

For a program running on a bus architecture, a call to "AsimSendMessage()" first correctly implements the logic of the "SendMessage()" call, then forks off an auxiliary process that calls "UserSend()". For such an architecture, "UserSend()" might look like this:

```c
QUEUE bus;
int bandwidth = 1.25e6

UserSend(src,dst,bytes)
int src,dst,bytes; {
    WaitQueue(&bus,(double)bytes/bandwidth);
}
```
That is, the auxiliary process requesting the "SendMessage()" waits in the bus queue with a request for \(\text{bytes} / \text{bandwidth}\) seconds of service time. For a more complex interconnection, such as a multistage delta network, "UserSend()" can call into action an arbitrarily complex simulation process. The time that a process spends waiting on a bus queue is not charged against a physical node, because presumably a process hung on a bus wait would get swapped out and another process on that node would get a chance to run. Therefore, bus wait time should not get tallied up as processor idle time, although the auxiliary process itself is logically delayed until the transaction is complete.

6.2. Modeling Software Overheads

In order to correctly account for the performance effect of software overhead in subroutines such as "SendMessage()" and "ReceiveMessage()", the ASIM subroutines make accounting calls which cause simulation time delays to be charged against the appropriate simulated processor. Since this overhead is likely to vary widely from architecture to architecture, the writer of an architectural model must define software overhead delay characteristics as part of the architectural specification. If simulation of a hypothetical architecture is desired, then the RPPT scheme should work well, giving the user the flexibility he needs to model a wide range of overhead delays, using whatever values seem "reasonable" or "realistic". If validation of simulation results against performance of the program on an existing architecture is desired, the only means currently available for inferring the value of these characteristics is through empirical observation of the real system for simple programs, followed by a check that the characteristics arrived at in this way remain valid over a range of different programs. In the absence of detailed low-
level (often proprietary) information on the implementation of the operating system for some real architecture to be modeled, only by a careful design of simple communication experiments on the architecture can the operating system behavior be captured. Validation of simulation predictions against measured execution times is discussed in Chapter 6.

One potential modification of the ASIM library that would prove useful in later RPPT work would be to rewrite the routines so that they are more isolated from the CC environment. Currently, since the ASIM library routines are only modified copies of the CC routines, they share many data structures. This arrangement normally causes no problems, because the only use of these data structures is being made by the RPPT application program. Occasionally, however, the CC support environment layer at the bottom of the RPPT environment uses data structures in a way that conflicts with the usage of that structure by the ASIM library. The solution would be to rewrite ASIM as a separate isolated layer of software that shares no data structures with CC. Although some duplication of effort for the solution to common problems might result, this would be outweighed by the increased ease of debugging a program built out of more modular, cleanly separated software layers.

7. ASIMP

ASIMP is a lex-based scanner that converts calls to CC subroutines into calls to the corresponding ASIM subroutines. This translation is necessary to provide an automated conversion of a CC program which can be run as a standalone CC application to an RPPT application program which will run in RPPT mode, in tandem with an architectural
simulation. The nature of the difference between CC and ASIM subroutines was described earlier in Section 6.

By providing an automatic filter such as ASIMP, the testbed maintains the convenience of allowing the use of the same CC program in both a CC mode and an RPPT mode. For instance, an RPPT application program is written with calls to CC subroutines SendMessage(), ReceiveMessage(), and the like. For purposes of testing some aspects of the program's logical correctness, it can be run as a standalone CC program. Later, when architectural information is added and the combination is run as an RPPT application, the program is processed by ASIMP, which lexically scans for calls to the CC subroutines and replaces them with calls to AsimSendMessage(), AsimReceiveMessage(), and the like. The program can now be profiled, compiled, and linked with an architecture module and run as a complete RPPT application.

Once the RPPT environment is expanded to handle tightly-coupled, shared-memory system models, the need will arise to distinguish between references to local and remote data, or to cacheable and non-cacheable data, or to shared and private data. In this case, ASIMP can also modify references to such data so that they are correctly simulated in the RPPT version of the program.

8. Miscellaneous Tools

A number of window-based user interfaces have been developed for use as front-end and back-end support in RPPT applications. SIMTOOL [14] helps the user to specify module and runtime parameter selection during specification of an application. It
creates makefiles and csh script files for building and executing the desired application with the requested runtime parameters. PLOTTOOL [27] performs a static post-mortem plotting of numeric data generated by a simulation run. DPlot [26], in contrast, is a dynamic back-end plotting package. It monitors a stream of performance data from the simulation and displays the data in the form of dynamically updated time histories or histograms. TRAPP [32] provides a structured replay of a simulation execution by displaying trace information in a collection of windows. Each independent simulation process is assigned its own window in which timestamped trace output from that process is displayed. In this way, TRAPP allows the user to more easily view the structure of an application's process interaction. Finally, STATTOOL [29], accepts a stream of raw performance data from an RPPT application, processes this information into higher-level user-oriented performance indices, and then pipes this information to DPlot for final display. Information generated by STATTOOL is in the form of means, standard deviations and histograms computed from the raw data, as well as smoothed and windowed versions of the input data stream. Currently, the responsibility for statistical computation is scattered throughout the software, including the CC, CSIM, and ASIM libraries. Ultimately, all libraries will be modified to output only raw data, passing the responsibility for all statistical manipulation to STATTOOL. Refer to the block diagram in Figure 3.1 above for an illustration of how these tools fit into the overall RPPT environment.

9. The Runtime Environment

The RPPT runtime environment can best be thought of as a collection of interacting levels of software. In order to take a collection of standalone programs and combine
them into an RPPT application, the application is built out of several layers of software, each acting as a client of the services provided by the layer immediately beneath it. At the bottom lies Concurrent C, which provides independent process control services. On top of this lies the CSIM layer, which enforces simulation time constraints, and simulates architectural activity. Finally, the ASIM layer runs as the top layer, providing the program execution activity needed to drive the underlying architecture layer.

One advantage of this organization is that software components at different layers can be developed and debugged independently, without the additional complexity of interaction. For example, CC programs or CSIM simulations may be run in either standalone or RPPT mode. A standalone application is one that is meant to serve as a parallel program with no architectural specification, in the case of CC, or as a simulation model with no program specification, in the case of CSIM. An RPPT application, in contrast, means that a CC program specification and a CSIM architecture specification will be combined and run as a single simulation. In this section, we describe how a layer interacts with the layer immediately beneath it, treating each layer as a client that accesses the services provided by the underlying layer.

9.1. A Client View of CC

At the bottom of the software layers lies the CC environment. The static linkage between CC client and CC environment is illustrated in Figure 3.3. The CC runtime library provides the definition of the C entry point "main()", which initializes the CC environment, creates a process descriptor for the CC driver, then transfers the thread of control to the driver process. Whenever a process suspends or terminates, it hands its
```c
main() {
    InitConcurrentC();
    Create( ConcDriver );
    Transfer( ConcDriver );
}
```

```c
ConcDriver() {
    Fork( UserMain );
    while (ready list not empty) {
        desc = getnextdesc();
        Transfer( desc );
    }
}
```

```c
UserMain() {
    ...
}
```

**Figure 3.3**

client view of Concurrent C
thread of control back to the process that transferred control to it originally. This controlling process is "ConcDriver()", which then makes a decision as to which process to pass control to next.

The CC client (either a standalone program or CSIM) provides a subroutine named "UserMain()". This is the subroutine that the client sees as its entry point. Inside the driver, "UserMain()" is forked off as an independent process and is put on the ready list. The first time "ConcDriver()" passes through its "while" loop, it finds "UserMain()" as the first process eligible for execution at the head of the ready list. When "UserMain()" runs, it makes its calls to "Create()" and "Activate()" that instantiate the initial user-defined processes and moves them onto the ready list. In this way, the client program is bootstrapped into the CC environment. "ConcDriver()" repeatedly passes through its "while" loop until no processes are found on the ready list, then terminates.

9.2. A Client View of CSIM

The next layer of software in the application is the CSIM environment. The static linkage provided by CSIM to the CC environment, as well as the static linkage expected by CSIM from its client, is illustrated in Figure 3.4. As a CC client, CSIM provides a "UserMain()" definition. "UserMain()" initializes the CSIM environment, then creates and activates a special process called "CsimDriver()". The purpose of "CsimDriver()" is to move processes between the CSIM event list and the CC ready list. An outline of the process code is given in Figure 3.5. "CsimDriver()" repeatedly executes a "while" loop, in which it moves all CSIM processes at the head of the event list onto the CC ready list, after which it suspends. The last currently active CSIM process to suspend or terminate
UserMain() {
    CsimInitialize();
    Create( CsimDriver );
    Activate( CsimDriver );
    UserInitialize();
    Join();
    UserFinalize();
}

template for entry and cleanup routines for CSIM client

UserInitialize() {
    ...
}

UserFinalize() {
    ...
}

definition provided by client

Figure 3.4

client view of CSIM
CsimDriver() {
    while (event list not empty) {
        for (each pd at head of list) {
            Activate(pd);
        }
        Suspend();
    }
}

CsimSuspend() {
    CsimApcount--;
    if (CsimApcount==0)
        Activate(CsimDriver);
    Suspend();
}

CsimActivate(pd)
PROCDESC *pd; {
    CsimApcount++;
    Activate(pd);
}

Figure 3.5
the CSIM driver
is then charged with reactivating "CsimDriver()". For this purpose, "CsimDriver()"
maintains a count of all active CSIM processes in a variable called "CsimApcount"
(CSIM active process count). All suspending CSIM processes decrement the count, and
all activating processes increment it, as illustrated by the definition of the subroutines
"CsimSuspend()" and "CsimActivate()" shown in the figure.

After "CsimDriver()" has been activated, "UserMain()" calls "UserInitialize()".
This is the entry point procedure into the CSIM environment seen by the CSIM client.
"UserInitialize()" makes the necessary calls to "ForkProcess()" and "ScheduleProcess()"
that instantiate the initial CSIM processes desired by the client. "UserMain()" then exe-
cutes a "Join()" which forces a wait until "CsimDriver()" has terminated.

9.3. ASIM as a CSIM Client

At the final layer of an RPPT application, the ASIM environment runs as a CSIM
client. The linkage for this layer is shown in Figure 3.6. As a CSIM client, the ASIM
layer provides definitions for "UserInitialize()" and UserFinalize(). For the RPPT
environment, a naming conflict that must be resolved at this level exists between CSIM
and the application program, both of which have been written to run as CC clients. The
solution is that CSIM remains the nominal CC client, providing a definition for the
"UserSend()" subroutine. The ASIMP preprocessor then renames the "UserMain()"
defined by the application program to "AsimUserMain()". This renamed process is then
scheduled in the "UserInitialize()" entry point provided by the ASIM layer.
UserInitialize() {
    ArchInit();
    desc = AsimCreate( AsimUserMain );
    SchedProcess(desc);
}

UserFinalize() {
    ArchFinalize();
}

ArchInit() { ... }  
ArchFinalize() { ... }  
UserSend(src,dst,bytes) { ... }

UserMain() { ... }  
ASIMP
AsimUserMain() { ... }

Figure 3.4
ASIM as a special CSIM client
Another naming conflict that arises at this level is between the ASIM layer and the architectural model, both of which have been written to run as CSIM clients. The solution here is that the ASIM layer remains the nominal CSIM client. The architectural module, rather than providing definitions for "UserInitialize()" and "UserFinalize()", as it would if it were a standalone CSIM application, instead provides definitions for "ArchInit()", "ArchFinalize()", and "UserSend()". In a sense, the architecture module can be thought of as a fourth layer running as an ASIM client.

It should be emphasized that much of the layering and resolution of naming conflicts described here is done automatically by a collection of environment- and application-building makefiles. The user must merely provide program and architecture specifications which contain the necessary subroutine definitions, place these specifications, along with the necessary makefiles, into the correct location in the RPPT directory hierarchy. This process is outlined in the next section.

10. Building an RPPT Application

The last section explained how the different software layers of an RPPT application are connected. This section presents detail on how each application layer is written, compiled, and loaded into the appropriate library as a testbed module, and how modules are linked together to create the final executable application simulation. Detailed suggestions for how to lay out CC programs to be run in the RPPT environment so as to minimize problems with timing the simulation are given in Chapter 6, and considerations for writing the program so as to streamline the profiling process are given in Chapter 4 on the TPROF timing profiler. Figure 3.7 illustrates the physical layout of the RPPT
directory structure. Composition of the "proglib", "archlib", and "examples" directories will be described below with reference to this figure.

10.1. Specifying an Algorithm

Each algorithm in the testbed occupies its own subdirectory of directory "proglib". Each subdirectory must contain the following files

- a program makefile
- source code, including a definition of the entry point procedure "UserMain()"
- information for "simtool" describing runtime command line parameters and default values

As mentioned above, the program initially can be developed and debugged in stand-alone CC mode, with no architecture model complication. Once it is ready for interaction with the architecture, an RPPT program application makefile must be created. The makefile must be organized according to one of various template makefiles to be found in directory "proglib". The available templates include "makefile.cross.hyp", "makefile.cross.seq", "makefile.cross.v", and "makefile.cross.320", depending on the timing profiler desired. The easiest way to build the makefile is to copy the desired template from the "proglib" directory and edit it to correctly reflect the names of the application's source code files. When the makefile is run, it produces an object file for later linking with an architecture module.
home = /mu1/sim

bin  concur  csim

proglib

eigen  lu  fft

prog makefile
prog source
simtool info
object module

archlib

hyper  ether  bus

arch makefile
arch source
simtool info
object module

texampless

ex1  ex2...
ex1

example makefile
nodespecs
acctfuncs
executable

Figure 3.7
RPPT file structure
10.2. Specifying an Architecture

In a fashion similar to the algorithm specification process, each architecture module has its own subdirectory in directory "archlib". This subdirectory should contain

- an architecture module makefile
- the architecture module source code including a CSIM definitions file, a CSIM process names file, and code defining "ArchInit()", "ArchFinalize()", and "UserSend()" procedures
- "simtool" information on runtime command line parameters

The makefile should be built by copying a template file "ModelMakefile" from the "archlib" directory and editing it to refer to the correct source code file names. When the makefile is run, it produces an an architecture object module for later linking into a complete RPPT application.

10.3. Linking the Modules

A complete RPPT application is assembled in a newly created subdirectory in directory "examples". This directory should contain

- an example makefile
- a "nodespecs" file, containing definitions of the number of logical nodes, physical nodes, the mapping between them, and the number and size of the CC process stacks.
an "acctfunes" file, providing functions that compute and return the amount of software overhead that the user wishes to be charged against the execution of the ASIM primitives on the simulated system.

The makefile should be a copy of the file "ModelMakefile" in directory "examples", edited to indicate the desired choice of architecture and program modules. When the makefile is run, it produces an executable simulation. If desired, the SIMTOOL application may be called inside this directory to build the correct makefile. SIMTOOL will also build a shell script file to run the application once it has been built by the makefile. The shell script is described in the next section.

10.4. Running the Application

SIMTOOL produces a csh script file that runs the executable simulation with the correct runtime command line arguments. Program results, as well as simulation timing and statistical information and debugging traces are then generated, which can be accessed via back-end tools such as TRAPP, STATTOOL, PLOTTOOL, and DPLLOT. The executable simulation is by default named "sim", and expects command line arguments in a standard format. Command line arguments to the architectural module are given in a list starting with a "-a" flag, and arguments to the program module in a list starting with a "-p" flag. The lists continue until the end of the line or the next flag. The flags are stripped away before the arguments are actually passed to the modules, so the user's code may access the argument vector as if only the arguments themselves were given on the command line. The subroutines "ArchInit()" for the architecture and "User-Main()" for the program should be written to access the argument vector just like
"main()" in a standard C program.

11. Generation of Performance Information

Now that we have seen how applications are built and how the software layers are interfaced, it remains to explain how performance information is generated by the profiled program and used to drive the architectural simulation. The principal contributions to the estimates for simulation time are computation times from the instruction profilers and communication times and software overheads from the architecture simulation specification.

11.1. Computation Time

The source code for a program application is assembled, analyzed by one of the TPROF profilers, modified by self profiling instructions, and compiled into an object module, as described above. In order to see how execution time estimates generated by the profiler are passed to the architectural simulation, consider the schematic representation of a generic algorithm given in Figure 3.8. An algorithm can be divided into a series of local computation intervals punctuated by process interaction points. These points are defined by a call to any ASIM procedure that requires communication or information exchange between processors. For RPPT algorithms running on the iPSC hypercube architectural model, the principal interaction points are "AsimSendMessage()" and "AsimReceiveMessage()". Profiling code inserted by TPROF accumulates cycle counts in a global variable as the program’s flow of control moves from block to block. At process interaction points, an interaction prologue or accounting procedure is called to allow
GenericProcess() {

    block 1
    block 2
    ...

    send or recv
    process interaction point

    block i
    block i+1
    ...

}
simulated time to elapse on a simulated processor by the amount implied by the accumulated cycle count. The accumulated count is reset, the processor’s ready list is blocked, a call to DelayProcess for the cycle count value is made, and the list is unblocked. During the resource allocation and delay, no other process wishing to execute on the node may acquire the node. In this way, the program’s execution time demand on the processor is simulated.

11.2. Communication Time

Communication time can be broken up into three main components: software overhead, transmission time, and synchronization time. Synchronization time is a result of the interleaving of suspends and activates and is not directly under the control of simulation parameters, although it is affected by relative rates of execution and physical channel bandwidths. Transmission time and software overhead are modeled as described above in Section 6 on ASIM.

12. Personal Contribution

The design and implementation of the RPPT environment, as outlined in this chapter, has been a group effort, to which the author has contributed as follows:

- Participated in the design and performed the implementation of the original CSIM (CSIM 1.0), from which the independent process control code was taken to create Concurrent C.
- Redesigned CSIM 1.0 into Concurrent C and CSIM 2.0.
• Implemented the new CSIM (CSIM 2.0), a rewrite of 1.0 to run on top of Concurrent C.

• Provided the ideas for the RPPT program-architecture interface -- UserSend, RemoteRead, RemoteWrite, the logical node-physical node distinction, physical node ready queue control.

• Supervised the implementation and validation of the standalone 68020 version of TPROF, and performed the design, implementation, and validation of the standalone 80286 profiler and 80286-68020 cross profiler, as well as cross profilers for V-UNIX and 32020-68020.

• Adapted LU decomposition and eigenvalue-eigenvector programs for execution on the hypercube and for simulation within the RPPT environment.

• Designed and supervised implementation of SIMTOOL, PLOTTOOL, and DPLLOT interfaces, and contributed ideas to the design of TRAPP and STATTOOL.
CHAPTER FOUR
The TPROF Cycle Count Profiler

1. Introduction

An important component of the RPPT environment which contributes to its power of prediction and economy of performance is the family of TPROF timing profilers. The profilers are the workload generator-generator for the RPPT simulations. That is, the profilers take a C program and generate a profiled program. The profiled program, when executed, generates the workload that drives the architectural simulation. This technique replaces such traditional techniques as statistical or trace-driven workloads described in Chapter 2.

In order to use the execution of a single program to drive multiple architectural simulations, the program is written in Concurrent C, with as few machine-dependent characteristics as possible. This program can then be profiled by any of the existing TPROF implementations, depending on the architecture for which timing estimates are desired. The profiler, in tandem with existing compiler tools, introduces the machine specific information necessary to generate execution time estimates for the program, making the program suitable for combination with the corresponding architectural model in a complete simulation.

Two styles of profiling are used in RPPT. Standalone profiling derives its timing estimates from information that is specific to the processor on which the profiling takes
place. The profiled code is then executed directly on this same processor. In certain circumstances, however, it is desirable to execute the profiled program on one processor (the host), yet allow the timing information be derived from a second processor (the target). This scheme, called cross profiling, is useful when program execution on the second processor may be expensive or inconvenient. In particular, cross profiling can be used to derive profiling information in situations involving

- a fast machine for which cpu time is expensive,
- a simple processor for which download access from a mainframe is inconvenient, or
- a need to run the profiled code on a specific machine as part of a larger programming environment which cannot be easily moved to any machine for which profiling information is desired.

In the case of the RPPT, it is the last scenario which gives rise to the need for cross profiling.

When compared with a standalone profiler for the same target processor, agreement between predictions is shown to be consistently within one percent. Validation results have been produced by several standalone and cross profilers developed at Rice. These include

- standalone profiler for C on 68020-68881 systems running SUN Unix 3.4 release 3.5 [1,2].
- standalone profiler for C on 80286-80287 systems Intel Xenix [7].
• standalone profiler for C on 320C25 systems [4].

• cross profiler, 68020 SUN Unix C host, 80286 Xenix/Microsoft C target.

• cross profiler, 68020 SUN Unix C host, 68020 V Unix C target.

• cross profiler, 68020 SUN Unix C host, TI 32020 C target.

These results will be discussed in detail in Chapter 6.

The remainder of this chapter is organized as follows. In Section 2, the ordinary standalone profiling scheme is described, and its limitations are discussed in Section 3. Section 4 describes the cross profiling scheme, with limitations given in Section 5. Finally, Section 6 offers a discussion and directions for future work.

2. Description of Standalone Profiling

Dynamic instruction count profiling was chosen for RPPT studies as a relatively low overhead method for obtaining detailed information on program execution time behavior. One of the first implementations of the technique is the bb (basic block) profiler [40]. The goal of the method is to capture as much information as possible on program behavior in a compile-time analysis, and then to introduce at code segment boundaries profiling instructions that will accumulate the information produced by the compile-time analysis when the profiled program executes. The code produced by such a process is sometimes described as self-profiling. An effective implementation of such a scheme produces a profiled program that executes in only slightly more time than the original unprofiled program. The extra instructions or time required to perform the profiling will be referred to as profile overhead.
In order to keep profile overhead low, the number of self-profiling instructions introduced into the program should be kept to a minimum. Yet the profiling instructions collectively must capture the execution behavior of the whole program. The optimal way of satisfying both requirements is to introduce profiling instructions at basic block boundaries at the assembly language level. Formally, a basic block is a straight line segment of assembly language code with the property that, during execution, each time any instruction in the block is executed exactly once, all other instructions in the block are also executed exactly once. In practical terms, a basic block begins with

- the first instruction in a file,
- a labeled instruction, or
- an instruction immediately following a branch.

Blocks end with

- the last instruction in a file,
- an instruction immediately before a label, or
- a branch.

Profiling analysis requires three steps.

- The program is broken down into a set of basic blocks.
- A timing analysis then assigns a weight or cost to each block.
- Finally, a new profile instruction is inserted into the top of each block.
The software modules required for this process are illustrated in the block diagram in Figure 4.1.

The newly inserted instruction becomes a part of the block and at runtime is executed once with each execution of the block, performing a dynamic profile update for the block according to the block's assigned weight. In this way, a minimal number of profiling instructions introduced at compile-time capture the complete execution behavior of the program at run time, insofar as the execution time of the instructions can be inferred from a compile-time analysis. The technique is called standalone in order to distinguish it from cross profiling, which will be discussed in Section 3.

Figure 4.2 shows a profiling example. Here, a "for" construct in C is assembled and analyzed into basic blocks. The basic block structure of the code can be seen more clearly in the graph representation, in which blocks of code are represented as vertices and control flow as directed edges. Once the basic blocks have been identified and assigned a cost, an instruction that increments a counter named "timer" is inserted at the top of each block.

2.1. Algorithm

This section describes the detailed steps in the standalone algorithm (see the block diagram in Figure 4.1).

2.1.1. Generation of Assembly File

This step is performed by the compiler, for example, by the "-S" option on the Unix C compiler. The C source is converted to an assembly language listing. Care should be
Figure 4.1
block diagram for standalone profiling
for ( a ; b ; c ) { d }

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a;</td>
<td>block 1</td>
</tr>
<tr>
<td>L1: b;</td>
<td>block 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>d;</td>
<td>block 3</td>
</tr>
<tr>
<td>c;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>L2: ...</td>
<td>block 4</td>
</tr>
</tbody>
</table>

```
add timer, cost1
a;
L1: add timer, cost2
b;
jeq L2;
add timer, cost3
d;
c;
jmp L1;
L2: add timer, cost4
```

**Figure 4.2**

basic block analysis and
insertion of profiling code
taken to ensure that this file corresponds to the executable version for which timing results are generated. That is, the compiler should be run with exactly the same options with respect to floating point options, level of code optimization, and memory model size.

2.1.2. Basic Block Identification (First Pass)

A forward reference problem forces the analysis to make two passes of the listing. In the first pass, the listing is scanned and written back to a temporary file, basic block boundaries are detected, and a place marker is written to the temporary file at the top of each basic block. Labels and branches are observed in this pass and a control flow graph for the file is created. The second pass is described below in Section 2.1.5.

2.1.3. Timing Analysis

Timing analysis is not actually a separate sequential step, but occurs simultaneously with the block identification step. Once a basic block boundary is detected, cycle count accumulation for the new basic block begins. In the absence of coprocessor interaction, this process is straightforward. First, instructions from the assembly listing are analyzed one at a time by breaking them down into opcode and operands. Second, operands are classified according to addressing mode. Then, a table lookup which is keyed on opcode and operand types provides a cycle count estimate for the execution time of the instruction. The table lookup may also take into account considerations such as bus traffic, processor state, and memory speed. Counts are accumulated until the next basic block boundary is detected, and the value is stored to be used in the second pass when basic block
markers are replaced by profiling instructions.

Timing analysis is more complex if the profiler must calculate overlap between main processor and math coprocessor activity. For example, in the case of programs compiled for the 80286-80287, code for the math coprocessor (80287) is generated in line with main processor code, yet at run time significant execution overlap between the chips is possible. Our current approach is to have the profiler calculate overlap according to a simplified model of the protocol governing the interaction between the chips as follows. First, the coprocessor (CP) waits for the main processor (MP) to issue a floating point (FP) instruction. Then, once the FP instruction is issued, the MP and CP may execute in parallel. There can be, however, only one FP instruction outstanding at any one time. The MP may therefore execute in parallel from the issue of one FP instruction up to the next FP instruction or the next FWAIT instruction. An FWAIT is generated whenever a data dependency arises and the MP must idle until the result of the currently outstanding FP instruction is available. An example assembly language fragment with inline 80286 and 80287 code is given in Figure 4.3.
<table>
<thead>
<tr>
<th>line</th>
<th>count</th>
<th>cum</th>
<th>opcode</th>
<th>operands</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>69</td>
<td>1076</td>
<td>fld</td>
<td>QWORD PTR [bp-58]</td>
<td>;s</td>
</tr>
<tr>
<td>2</td>
<td>177</td>
<td>1145</td>
<td>fmul</td>
<td>QWORD PTR [bp-76]</td>
<td>;t</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1145</td>
<td>mov</td>
<td>bx, [bp-68]</td>
<td>;i</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1145</td>
<td>shl</td>
<td>bx, 2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>1145</td>
<td>les</td>
<td>si, [bp+20]</td>
<td>;z</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>1145</td>
<td>les</td>
<td>si, es: [bx][si]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1145</td>
<td>mov</td>
<td>bx, [bp-80]</td>
<td>;j</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>1145</td>
<td>shl</td>
<td>bx, 3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>69</td>
<td>1322</td>
<td>fld</td>
<td>QWORD PTR es: [bx+8][si]</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>177</td>
<td>1391</td>
<td>fmul</td>
<td>QWORD PTR [bp-16]</td>
<td>;c</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1568</td>
<td>fsub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>115</td>
<td>1668</td>
<td>fstop</td>
<td>QWORD PTR [bp-104]</td>
<td>;temp</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>1783</td>
<td>fwait</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 4.3:**
80286-80287 assembly code fragment and timing analysis

For each line in the table, the count column indicates the execution time estimate for the instruction in cycles. The cumulative column indicates the total number of cycles in the basic block so far. This column is not incremented until the overlap is determined. One such overlap calculation begins at line 2 and runs through line 8. At line 2, a floating point instruction of cost 177 cycles is issued. This instruction can run on the 80286 parallel with the the main processor instructions on lines 3 through 8. Since these instructions only have a total cycle count of $6+5+22+22+6+5=66$ cycles, and since the next floating point instruction gets issued at line 9, then the execution time of the overlapped instruction streams is $\max(177,66)=177$, and the block time is incremented by 177 to account for lines 2 through 8.
2.1.4. Basic Block Reduction

After the control flow graph for the program has been generated, it may often be reducible. Reduction is a process that coalesces certain blocks in the graph without changing the graph’s control flow. The motivations for block reduction are

- reduction of profile overhead in both the standalone and cross profilers, and
- improved basic block matching in the cross profiler.

Block reduction has the effect of increasing the average length of a basic block, resulting in lower profile overhead, since one profiling instruction is inserted for each block. It also tends to prune away certain nonessential blocks that complicate the block matching process for the cross profilers.

The opportunity for reduction can occur when the compiler generates a label but never uses it as the target of any branch. In this case, during basic block analysis, a new block is started when the label is seen, but after analysis is complete, it is determined that no branch ever uses that label as a target. This can happen if a compiler automatically generates labels for particular points in the code for some control flow structure. For instance, C features loop control statements such as "break" or "continue" within the body of "for" and "while" constructs. No matter where or how many times in the "for" or "while" body a "break" or "continue" is invoked, the constructs act as goto’s with implicit targets, with the target located in a fixed position with respect to the other code in the for loop. A given compiler may choose to generate labels at these fixed positions to act as "break" or "continue" targets, without checking to see if a "break" or "continue" is actually invoked. If the constructs are in fact never invoked, the label proves unnecessary,
and the block reduction step detects and coalesces any resulting extra blocks.

Another opportunity for block reduction occurs when an unconditional branch is made from one block to another, and the second block is not branched to by any other block, and never entered by control falling through from the previous sequential block. In this case, the second block is a simple logical extension of the original block. A compiler may produce such a situation when control flow constructs omit optional sections, but the compiler generates a skeleton for the complete structure anyway. For instance, an "if-then" without a matching "else" on the 80286 Xenix/Microsoft C compiler creates such a basic block arrangement.

Both of these opportunities for basic block reduction manifest themselves in the program graph as a chain of blocks each with one predecessor and one successor. The reduction algorithm proceeds in general by taking the graph representation of the basic block structure, detecting all occurrences of the single-predecessor, single-successor relation, and combining each distinct chain of such blocks into a single reduced block. Formally, this operation is known as an adjacent vertex homomorphism.

For purposes of correct operation of the profiling process, reduction is optional, but it is still recommended for the advantages outlined above. An example of an unreduced graph and its corresponding reduced version are given in Figure 4.4. The reduction step proceeds by locating all chains of blocks that start with a block with zero or two or more predecessors, that proceed to only one successor, and that end with a block with zero or two or more successors, then coalescing the chain into a single block.
Figure 4.4
example basic block reduction
2.1.5. Profile Instruction Insertion (Second Pass)

After basic block detection, timing, and reduction is done, the temporary file with place markers is scanned. Markers at the top of each reduced basic block are replaced by profiling instructions which increment a counter by the timing weight assigned to that block. Care must be taken when inserting profile code to not affect condition codes, and to not interfere with code that is involved in environment setup. For example, the UNIX C compiler for the 68020, at the beginning of each procedure, generates a prologue segment which sets up the stack environment for the procedure. In this instance, rather than uncritically inserting profiling code into the top of each block, the code must follow the stack setup segment.

2.1.6. Compilation

The instruction insertion step produces a modified assembly language listing of the original program. The modified listing can now be compiled into a self-profiling executable image.

3. Limitations and Possible Enhancements for Standalone Profiling

We have seen the overall profiler scheme in the previous section, and shall now consider in detail some of the problems that prevent the profiler's timing predictions from attaining 100% accuracy.
3.1. Pipeline and Cache Effects

The timing analysis employed by the profiler assumes that each instruction can be assigned an exact time representing the precise number of cycles that the processor is involved in executing that instruction. In reality, for a processor with an instruction pipeline, or even one with just a prefetch buffer, overlap between multiple instructions in various states of execution makes it difficult to assign any one execution cycle to a single instruction. In effect, exact cycle count timing analysis is possible only through a careful cycle by cycle simulation that keeps up with the state of the whole pipeline. Our scheme, however, attempts to avoid the prohibitively costly execution time of the exact scheme, while not sacrificing significant accuracy.

Timing analysis is implemented as a table-driven lookup of timing costs for each instruction, which may require a scan and parse of instructions to break them up into opcodes and operands. Cost information is supplied by the manufacturer's documentation for the processor. For a complex chip such as the 68020, the documentation attempts to capture pipeline and cache effects by quoting several timing costs for each instruction, one cost for each of several somewhat idealized execution conditions. Each 68020 instruction is assigned a best, cache, and worst case execution cycle count. Best case means both instruction pipeline and data cache are assumed to work perfectly, cache case means the cache works perfectly and the pipeline is turned off, and worst case means the cache and pipeline are both turned off. For a given program, the timing analysis process must choose which case, or which weighted average of the possible cases, best represents the typical program execution conditions. Empirical results
derived from observation of program execution on local 68020-based systems suggests that typical behavior is slightly worse than cache case behavior. For any arbitrary program, pipeline and cache effectiveness will fall at some characteristic point between best- and worst-case values. For example, choosing individual instruction values that are 20 percent of the distance from cache to worst case values produces profiling estimates that are in good agreement with measured times, although this choice is somewhat arbitrary.

For processors such as the 68020 or 80286, which feature on-chip instruction pipelining, it is possible to partially correct at compile time for some effects that will not be observable until run time. The profiler estimate can be improved by allowing the profiling instructions to take advantage at run time of program state information to dynamically adjust the weighting actually used for some block. For instance, conditional branch statements have different timing costs depending on whether or not the branch was taken. A successful conditional branch is more expensive on a processor with an instruction pipeline than an unsuccessful one because it forces a pipeline flush. The profiler can compensate for this effect by adding profiling code that sets a status flag when branches are not taken. This flag can then be examined, and if a branch has been taken, a pipeline flush penalty cycle count can be accounted for. The cost of the added accuracy of such a scheme is the extra overhead per basic block required by the status setting and checking instructions.

Profilers in general play an important role in the modeling methodology of the testbed. The sophistication of the profiler determines the level of detail for which program interaction with the architecture may be modeled. The profilers described here are
adequate for systems, either without cache, or with cache but disallowing shared data. Profilers to take into account performance effects of cache coherency enforcement are being currently being developed. Such profiling requires a modification to the simple and efficient profiling scheme outlined as part of the general execution-driven methodology. For cache profiling, simulation of the cache model can still be constrained to happen only at basic block boundaries, allowing some performance improvement over a traditional instruction-level simulation. For each basic block, however, a trace of all memory references for that block must be generated, and the effect of the references on the cache must be simulated. This extra simulation detail will give rise to a significant increase in profiling overhead, and a degradation of simulation execution time efficiency. Cache profiling is currently limited to models that do not allow shared data to be cached. Degradation of simulation performance will inevitably grow worse when this restriction is lifted.

The feature of such simulations that ultimately bring about this performance degradation is the granularity of global process interaction points. The frequency of interaction points dictates how frequently global simulation time gets updated. For loosely-coupled multiprocessor systems without shared memory or cache, in which processes communicate via relatively infrequent message passing, the only process interaction points are the message passing points. For a cache model with shared memory in cache, every reference to a shared data item becomes a process interaction point.
3.2. Data Dependent Instruction Times

Since timing analysis must be performed at compile time, instructions such as
shifts, microprogrammed I/O instructions, and others may introduce inaccuracies into the
analysis, since the time required by these instructions at run time depends on the value of
the operands, quantities which are not available during the compile time analysis of the
program. At compile time, an instruction time that represents operands whose values are
in some sense "typical" or "average" must be used. To more accurately predict the cost
of such instructions, the profiled code inserted at basic block boundaries could be
modified so that rather than incrementing a counter by a compile-time constant, the cost
of the block could be calculated dynamically. The code could analyze the operands and
calculate an appropriate cycle count. The cost of such improved accuracy would be the
increased profiling overhead.

3.3. Calls to Unprofiled Subroutines

A program that calls subroutines, such as those from the math library, which have
not been profiled must independently account for the cycles expended inside such sub-
routines. Typically, if a copy of the source code for these subroutines cannot be obtained
for profiling, the cost can be estimated from direct timing measurements of simple loop
programs that repeatedly call the subroutine. From such experiments, an average cost
per call value can then be estimated, assuming that the cost is not parameter value depen-
dent. Validation problems of this sort are discussed in more detail in Chapter 6.
4. Description of Cross Profiling

The motivation for cross profiling stems from the attractiveness of specifying the implementation of an algorithm to be studied in the testbed in a relatively machine-independent way. In this way, a single algorithm representation can be written for consideration with respect to a variety of architectures, and the responsibility for incorporating machine-dependent timing information can be relegated to the profilers. The profilers, therefore, act as architectural-information filters, which take a machine-independent program and merge with it all necessary architecture-specific timing information. This is critical for the RPPT environment since it features many targets (iPSC hypercube, Sequent Symmetry, TI Odyssey) and one host (Sun 3). An underlying assumption for this approach is that timing information derived for the program as it would look on one machine (the target) can be incorporated in a systematic way into the program representation on a second machine (the host). In particular, the cross profiling technique we use assumes that a block of code can be identified at the source level and that a correspondence exists between different assembly language representations for that block for different machines. Even if a single HLL source code program exhibits different basic block representations for different compilers, we assume that a nearly one-to-one correspondence still exists between blocks of the different representations. Therefore, if a compiler-independent basic block representation of a program exists, then timing analysis can be performed on the target assembly language representation (TALR), and profiling instructions inserted into the blocks of the DALR (destination or host assembly language representation) will carry the weights derived from the corresponding
block in the TALR.

In practice, the correspondence between the DALR and TALR is not perfect because of idiosyncrasies in each of the two compilers. For most commonly used HLL constructs, however, the differences do not result in substantive differences in profile estimates, and can therefore usually be ignored, or the discrepancies can be patched by a small set of *ad hoc* corrections. A detailed description of correspondence problems is given below in Section 5.2.

The cross profiling algorithm requires a series of analysis steps to be performed on both the TALR and DALR. These steps are similar to the steps required for standalone profiling, although no timing analysis is performed on the DALR. Also, additional initial and final steps are required, initial steps to mark the source code so that basic blocks in the assembly language can be anchored to a particular construct in the HLL source code, and final steps to merge the results of the analysis of the TALR and DALR. The detailed steps of the algorithm are given in the next section.

4.1. Cross Profiling Algorithm

This section gives a detailed description of each step in the cross profiling scheme. A block diagram of the process is given in Figure 4.5.

4.1.1. Source Level Marking

In this step, the HLL source program is augmented by markers that identify what source level construct a particular code segment is associated with. We will call these block markers, which have no relation to the place markers used to mark the location
Figure 4.5
block diagram for cross profiling
for insertion of profile instructions by the standalone profiler. In a later processing step, the block markers will be converted to comments embedded in the assembly language. The markers fall into some basic block and identify the block by binding it to the source construct indicated by the form or name of the markers. Identical markers will appear in blocks in each of the two assembly language representations, establishing a correspondence between the blocks.

The C source code is passed through a lex-yacc parser, which generates the program's parse tree, which is then written back out with block markers inserted at certain control flow points. The names of these subroutines reflect the significance of their insertion location, e.g., "FB();" for "for begin", "WT();" for "while test", and so on. The intent is that each basic block in the resulting assembly language file contains at least one block marker function for purposes of unique block identification and later matching of blocks between the DALR and TALR.

Implementation of the block marker insertion process can take many forms. If a compiler has an "asm" feature -- a mechanism for inserting assembly language code inline into the C source code -- then the markers may be passed through directly to the assembly language level. In a C program, however, there are often certain segments of code -- e.g., code that appears in "for" control expressions -- that cannot be tagged with an "asm" construct. Even worse, some C compilers do not even provide an "asm" feature. In such cases, it is necessary to introduce superfluous code into the program solely to pass marking information through to the assembly language level, then filter out the code generated from the final listing, while retaining the marking information that it
provides. The exact scheme used to accomplish the marking function is irrelevant, so long as it correctly passes markers into the assembly language code. An example of unmarked and marked C source with the corresponding assembly language versions are given in Figure 4.6.a through 4.6.d.

4.1.2. Generation of Assembly File (TALR and DALR)

When the marked source is assembled, the marks can result in

- direct "asm" insertions into the assembly language,
- calls to subroutines with indicative names, such as "jbsr_FT" or "jbsr_WT", or
- assignment of suggestive string constants to some pointer variable, such as "xxx = (char *)"FT";".

Care must be taken to ensure that marking code does not cause extraneous new code to be generated in an unpredictable way. This is a problem for small chips like the 32020, for which there is little machine state, and the compiler is making very careful use of all available registers. Marking code can perturb the global machine state, causing much resetting of register values. This extra code can in turn perturb the execution behavior of the program. This code does not result directly from the assembly of some item of marker code, and is therefore difficult to filter out directly from the assembly file. This can be considered as another manifestation of a probe effect, in which the observation of system performance affects that performance.
for (l = m; l >= 1; l--) {
    tst1 += fabs(d[l]);
    if (*((iter) > maxit) {
        tst2 = tst1;
        if (*info == 0) {
            *info = m;
        }
    }
}

Figure 4.6.a
unmarked C source code

for (xxx_FL_003(), l=m;
     xxx_FT_003(), l >= 1;
     xxx_FR_003(), l-- ) {
    xxx_FBB_003();
    tst1 += fabs(d[l]);
    if (xxx_IT_000(), *((iter) > maxit) {
        xxx_IBB_000();
        tst2 = tst1;
        if (xxx_IT_001(), *info == 0) {
            xxx_IBB_001();
            *info = m;
            xxx_IBE_001();
        }
        xxx_IBE_000();
    }
    xxx_FBE_003();
}

Figure 4.6.b
marked C source code
movl a6@(-0x64),a6@(-0x60)
L189:
cmpl #0x1,a6@(-0x60)
jit L188
...
fmoved sp@+,fp0
fadd a6@(-0x40),fp0
fmoved fp0,a6@(-0x40)
...
jle L193
movl a6@(-0x38),a6@(-0x40)
movl a6@(-0x34),a6@(-0x3c)
movl a6@(-0x24),a0
tstl a0@
jne L196

Figure 4.6.c
unmarked assembly language

| ~ FI 003
movl a6@(-0x64),a6@(-0x60)
L189:
| ~ FT 003
cmpl #0x1,a6@(-0x60)
jit L188
| ~ FBB 003
...
fmoved sp@+,fp0
fadd a6@(-0x40),fp0
fmoved fp0,a6@(-0x40)
| ~ IT 000
...
jle L193

Figure 4.6.d
marked assembly language
4.1.3. Marker Code to Marker Comment Conversion (TALR and DALR)

If an "asm" call is not used to introduce markers, then the side effect code that gets generated must be filtered out, while leaving behind the location identification information. This step is a simple lexical translation of marker subroutine calls or string assignments into assembly language comments. For example, "jbsr _FT()" gets translated into "; FT".

4.1.4. Basic Block Identification (TALR and DALR)

This step is similar to the corresponding step for the standalone profiler, except that the markers introduced must be examined and a record kept of which block they belong to.

4.1.5. Timing Analysis (TALR only)

No timing analysis is performed on the host code. Line numbers indicating the start of a basic block are maintained for the DALR, to be used when profiling instructions are inserted later. For purposes of validating the predictions of the cross profiler, it is important that the timing analysis step for the cross profiler be identical to the corresponding step in the standalone profiler.

4.1.6. Basic Block Reduction (TALR and DALR)

In this step, reduction as described for the standalone profiler is performed separately on both the TALR and DALR. In the cross profiling case, the reduction step serves not just to reduce profiling overhead, but also simplifies the task of block matching
between the TALR and DALR. Reduction helps the analysis to arrive at a canonical basic block representation for the algorithm, filtering out the quirks of individual compilers. It is also more likely that every reduced basic block will have at least one block marker which can be used for matching purposes in the merge step.

4.1.7. Merge: Match TALR and DALR Basic Blocks

In this step, a given marker is searched for in both the TALR and DALR. If the marker belongs to TALR block i and DALR block j, then blocks i and j are assumed to correspond. This is the step in which the principal source of error for the cross profiling scheme is introduced. The basic block structure according to two different compilers is unlikely to be identical, even after block reduction. Constructs that commonly give rise to the most problems are "case" statements, "if"s with no matching "else", complex expressions for loop tests, and expressions involving data types that are not the same size on the host and target (e.g., "int" on the 68020 (4 bytes) and 80286 (2 bytes)). These problems are discussed in more detail in the next section.

The merge step is also a convenient point to introduce enhancements such as time slicing, which acts as a partial solution for synchronization problems that insinuate themselves into the simulation via the ASIM library routines. These problems arise because of the default Concurrent C process scheduling protocol of not relinquishing execution control until an explicit suspension is encountered. In ASIM routines, certain execution interleavings of user processes calling these routines can produce situations in which inordinately long synchronization delays may be experienced by some user process. For example, one ASIM problem arose because one user process ran without relinquishing
execution control for a significant time, blocking the execution of another ready to run process which was trying to initiate a message pass. Another process, blocked on a receive for the message, experienced an unrealistically long delay. The simulation results did not match the observed hypercube behavior because timeslicing by the real system prevented such an accumulation of synchronization delay. Introduction of timeslicing into the simulation brought the simulation results into agreement with observed hypercube behavior.

Timeslicing is implemented at the merge step by introducing additional code at each basic block boundary in addition to the profiling code already inserted. The new code keeps track of the value of a variable that indicates how many cycles have dynamically elapsed. The value is incremented by the number of cycles for the block just about to execute, and if the total is greater than the timeslice, the accumulated value is set to zero and the process relinquishes execution control and moves to the tail of the ready list. This effect is provided by the CC procedure "Nice()". Figure 4.7 depicts the code inserted at basic block boundaries to implement time slicing. Note that "AsimNice()" makes calls to accounting procedures as described in Chapter 3. One of the side effects of the account procedure is to reset Xinstrs, so that the measurement of the next timeslice interval can begin. The desired value of the timeslice is specified in the "specs" file, which is provided when setting up an application. A description of this file was given in Chapter 3, Section 10.3.
block i:
    Xinstrs += cost(i);
    if (Xinstrs>TimeSlice) AsimNice();
    ...
    ...

block i+1:
    Xinstrs += cost(i+1)
    if (Xinstrs>TimeSlice) AsimNice();
    ...
    ...

Figure 4.7

diagram for timeslicing code insertion
4.1.8. Profile Instruction Insertion (DALR only)

If a correspondence between TALR block i and DALR block j has been established, DALR block j can be augmented with a profiling instruction with weight derived from TALR block i.

5. Limitations to Cross Profiling

5.1. Target Compiler Access

The system must have access to a compiler on the target system, or a cross compiler on the host system.

5.2. Basic Block Mismatches

Differences in the basic block structure generated by the target and host compilers inevitably occur, producing block mismatches. Ideally, any given HLL control flow construct can be considered to give rise to a unique basic block structure in some idealized assembly language. This will be true for most HLL constructs, and for any but the most exotic of instruction sets, because there is a relatively strict interpretation attached to a given HLL construct, and control flow in most assembly languages is implemented totally by conditional and unconditional branches. Therefore, there can only be a small number of fundamentally distinct assembly language realizations of a single construct. Still, compiler quirks do occasionally give rise to mismatches between the TALR and DALR. Compiler optimizations are particularly likely to introduce code into the ALR’s that gives rise to fundamental mismatches. Some important representative problems are as follows.
5.2.1. Case Statements

Case statements cause mismatch problems between ALR’s if one compiler attempts to optimize the target label selection by use of a PC-relative jump instruction when the range between the minimum and maximum label value is small. The SUN C compiler optimizes in such situations by building a PC-relative jump table, using the case label as an index into the table. Figure 4.8.a illustrates the basic block structure generated by the SUN C compiler, using a jump table, for a generic case construct. Figure 4.8.b, in contrast, shows how the case statement is treated on systems such as the iPSC hypercube that do not use the PC-relative jump. The construct here is simply a long chain of if-then-else tests and jumps.

Given that blocks do not match perfectly for case statements, there are several options to repair or minimize the damage. First, we could simply disallow case statements from programs to be cross profiled. This is rather restrictive, so we may instead try to force the SUN compiler to not use the pc-relative jump. We can do this by adding a case label with a large value that will never be selected by the real program. This forces the difference between the largest and smallest label value to be so large, requiring a prohibitively long jump table, and the compiler will generate instead a chain of if-then-else comparisons. Another option is to insert counts into each case block that reflect the cost of the number of if-then-else comparisons traversed in arriving at the block. This last scheme is accurate, but it will result in some target blocks -- those blocks corresponding to the chain of if-then-else tests -- that appear to have no host match, even though their cycle count effect will be included within the body of the case label code.
Figure 4.8.a

case statement basic blocks

optimized
Figure 4.8.b

case statement basic blocks
unoptimized
5.2.2. Redundant Jumps

"If-then-else" statements with a missing "else" clause can cause a block mismatch by the introduction of unnecessary single-jump blocks, as shown in Figure 4.9. In the case of an Intel iPSC hypercute target and a SUN 3 host, an extra block exists in the TALR that gets executed on the target, yet there is no corresponding host block to which to assign the target block's execution cycle count. It may be possible to use the C parse and mark step in the profile process to introduce a program transformation, i.e., have the parser "fill in" the missing else clause, possibly adding a null operation, to produce a better match between blocks. Currently, this mismatch is simply ignored since the resulting error is almost always negligible, as will be seen in Chapter 6.

5.2.3. Expression Evaluation

Block matching problems can also occur, even when the target and host compilers produce identical block structures, if there exist blocks that fail to receive some identifying mark during the parse-and-mark step. In this case, the merge step has no way of inferring matches between blocks even though the correspondence between them may be exact. This failure can happen for even relatively simple logical predicates, such as that shown at the top of Figure 4.10. This expression results in the basic block structure shown to the left of the figure. Currently, no attempt is made to exhaustively mark each expression and subexpression, so such constructs can result in unmarked -- and therefore unmatchable -- blocks. The block marker currently works better for simpler structures such as that one on the right, where the block for the logical predicate requires only a single block, which is guaranteed to possess an identifying mark.
if (test) { if-clause }

Figure 4.9
basic blocks for else-less if clause
if ((i && j) || (k ^ l)) { ... } else { ... }

Figure 4.10
expression evaluation with and without
basic block structure
5.2.4. Long Loop Control Variables

Long loop control variables can cause block mismatches if one instruction set can perform test or compare operations on 32 bits in one instruction and another must compare or test 16 bytes at the time. This is the case for the 80286-68020 profiler operating on programs with long int control variables. In Figure 4.11, we see the difference between the basic block structure for a "for" construct with a control variable of type "long int". The structure on the left is the 80286 version, and the version on the right is the 68020 version. The discrepancy arises because the 80286 must perform its test of the value of the 4-byte long int 2 bytes at the time, increasing the complexity of the block structure of the loop control code. The most effective remedy for this mismatch is to introduce a program transformation that forces the 68020 block structure to more closely fit the 80286 structure.

5.2.5. Other Systematic Basic Block Differences

For the case of the 32020-68020 cross profiler, a matching problem of a different kind arises. The 32020 C compiler replicates the test-block code at the bottom of the body block for for-loops and while-loops. This creates a difficulty for merging because the mark for the test block now appears twice in the TALR, complicating the determination of a unique match between corresponding markers in the TALR and DALR. A comparison between block structures for a "for" loop on the 32020 and the 68020 is given in Figure 4.12.
for (i = 0; i < n; i++) {...

long i;

for init

cmp(h(i), h(n))

body; i++

exit

int i;

for init

cmp(i, n)

body; i++

exit

Figure 4.11

long and int control variables in 80286 for construct
host (68020):
cycles (n) =
c1 + (n+1)*c2 + n*c3

target (32020)
cycles (n) =
d1 + n*d2

Figure 4.12
basic block structure and cycle counts
"for" loop
68020 vs. TI 32020
Finding an acceptable correspondence between target and host blocks requires consideration of the problem from another perspective. That is, there exists a more fundamental basis on which to assign a correspondence between blocks than the markers that they contain. Our ultimate objective is to find a correspondence that causes the cross profiled 68020 program to generate the same cycle count as the standalone profiled 32020 program. In the figure, each block is labeled by its cycle count, and its logical function is indicated by the text to the side of the node. First we write analytic expressions for cycle counts in the two cases. If a "for" loop is executed \( n \) times on the 68020, the number of cycles is \( c_1 + (n+1)c_2 + n \ c_3 \), and for the 32020, it is \( d_1 + n \ d_2 \). By making the assignment \( c_1 = d_1, c_2 = 0, c_3 = d_2 \), we can insure that the 68020 cross-profiled code calculates the same number of cycles as the 32020 standalone-profiled code.

6. Validation

Two independent validation schemes are necessary to validate the execution time predictions of the cross profiler. First, the standalone profiler for the target processor must be validated by comparison with execution time measurement of the unprofiled program running on the target machine. This measurement is normally performed by a hardware clock and software that ensures that only system time for the process is measured.

The cross profiler is then validated by comparison with the prediction of a standalone profiler for the target machine. Validation methodology and results are given in detail in Chapter 6.
7. Suggestions for Future Work

Standalone and cross profiling schemes have proven to be important tools for accurate workload generation in RPPT applications. Some of the limitations of the schemes are fundamental, having to do with inherent bounds on the accuracy of compile-time estimates of run-time instruction execution costs. Certain problems with cross profiling, however, could be minimized by developing better error condition handling and better user interfaces for the tool. For instance, the merge step of the cross profiler generates a list of diagnostics that indicate those blocks in either the target or host program that fail to match. Other similar error conditions, or unusual situations that the user should be warned about, are also reported in this file. Currently, this diagnostic output is at a relatively low logical level, requiring some familiarity with the software for correct interpretation. If an unmatched basic block is found, a warning message indicates the line number in the assembly file where the block begins. Since basic block mismatch errors are usually characteristic of certain C source code stylistic issues, the tool would be more usable if it were able to translate the assembly language location of the error back to the location in the source code that generated the problem.

It may even be preferable to introduce some fixes to these stylistic problems by automatically implementing some syntactic transformations during the C source code parsing, before compilation and block analysis. For instance, "if" statements with no matching else could be transformed into "if" statements with empty "else" clauses.
CHAPTER FIVE
Architecture and Program Models

The RPPT will ultimately be a collection of interchangeable architecture and program modules. Over the past two years, it has been expanded to incorporate around a half dozen architecture models and a dozen algorithms. A significant limitation on validation work up to this point has been the number of parallel systems on which to run real programs and against which to compare predictions. This limitation is less acute for example programs, although the programs to date have tended to be numerical algorithms such as linear algebra applications, with few non-numerical algorithms such as graph-theoretic ones.

The validation work described in this thesis concentrates on a subset of the available architectures and algorithms and will be presented in detail in Chapter 6. This chapter gives a description of the architecture and program modules considered.

1. Architecture Models

The RPPT architectural models are currently limited to tightly coupled designs, although work is under way to add designs such as the Sequent Symmetry into the testbed. Currently available models include a bus-oriented multiprocessor, an Ethernet-based system, and a hypercube organization.

The process of architectural model specification in RPPT requires only a description of the interconnection between processors in the model. Internal details of the specific
processor architecture -- e.g., pipeline, cache, numeric coprocessor interface, software overheads -- are not specified in the architectural model itself. The performance effect of these details is accounted for by the timing profilers or accounting functions. Currently, the testbed requires all processors in an architecture to be identical, except for certain parameters such as clock rate. This restriction could be relaxed, however, with some relatively minor enhancements to the testbed. Details for the specification process and code layout are given in Chapter 3. All validation results presented here are derived from studies of applications running on the hypercube, and a discussion of the real system and how it is modeled are given next.

1.1. Intel iPSC Hypercube

The local computing environment at Rice includes a 16-node Intel iPSC hypercube. The hypercube architecture is defined by an interconnection relation for linking \( m = 2^n \) nodes together, where \( n \) is called the dimension of the cube. Let the \( m \) nodes in an \( n \)-dimensional cube be numbered from 0 to \( m - 1 \). Then node \( i \) is connected to each node \( j \) for which the binary representations of \( i \) and \( j \) differ in exactly one bit. In an \( n \)-dimensional cube, each node is connected by a direct physical link to exactly \( n \) other nodes. If messages are sent between processors running on the nodes of the cube, we can characterize the distance the message must travel by speaking of the number of "hops" or links it must traverse from node \( i \) to \( j \). In a cube of \( m = 2^n \) nodes, the maximum number of hops separating any two nodes is equal to \( n \), the dimension of the cube. Therefore, the maximum communication latency grows only as \( O(\log m) \) in the number of nodes in the system. The hypercube therefore achieves an effective balance between cost and
communication latency, avoiding both the high cost of a full connection such as a crossbar and the high maximum latency of a bus or mesh connected system. A schematic of the hypercube interconnection is given in Figure 5.1.

The following description and figure for the iPSC node architecture are taken from [5]. The iPSC system comprises a manager processor and a hypercube of 16 nodes. The manager is a 80286-80287-based system that executes the XENIX timesharing operating system. Each node in the hypercube consists of an 80286-80287 main processor-math coprocessor pair, each clocked at 8 MHz, a 512 Kbyte, dual port, zero wait state memory, an expansion memory slot with a 512 kbyte 1 wait state memory, a processor bus, an I/O bus, and five 82586 LAN controller message coprocessors (four to build the cube, one for the the global channel to the manager). The 80286 has access to one of the memory ports via the processor bus, and all message coprocessors access the other memory port via the common I/O bus. The architecture of a single node is shown in Figure 5.2.

When a packet arrives over some channel to a message coprocessor, the coprocessor interrupts the main processor and waits to be given a pointer to a memory location for the incoming packet. Acknowledgement is made on a point-to-point per-packet basis. There is no overall acknowledgment for completely received messages. Once the pointer is available, the main processor may proceed in parallel with the coprocessor's moving the packet into memory. Simultaneous packet traffic in and out of a single node is theoretically limited by the bandwidth of the bus that connects the message coprocessors to main memory. There is no contention for channels, since channels are a direct link between exactly two nodes.
Figure 5.1

Hypercube Interconnection Structure
Figure 5.2
Intel iPSC node architecture
On top of the node hardware runs the Intel Node Operating System (NOS). This is a minimal system responsible for message passing and process control. The NOS maintains a single user system allowing a limited number of active user processes at one time. The NOS grants each active process 50 msec timesliced access to the cpu. In addition to timeslice interrupts, the cpu may be interrupted by the message coprocessors for service of message passing activities. The NOS is responsible for translating requests for message sends and receives between processes into point-to-point packet transmission across the requisite number of links. Message length can vary from 0 to 16 kbytes, with long messages broken up into the necessary number of packets of maximum length 1 kbyte. The packetization algorithm divides multi-packet messages into approximately equal-sized packets.

For programs running on one of the nodes, the user program interface to the message passing system provides send and receive primitives, which have the following syntax and semantics, as described in [5,6].

```c
sendw(ci,type,buf,len,node,pid)
int    ci,type;
char   *buf;
int    len,node,pid;
```

```c
recvw(ci,type,buf,len,cnt,node,pid)
int    ci,type;
char   *buf;
int    len,*cnt,*node,*pid;
```

The parameter "ci" is a channel id number, "type" is a user-defined message type, "buf"
points to the message being sent or received, "len" is the size of the buffer, "cnt" is the number of bytes actually received, and "node" and "pid" identify the sender or receiver of the message. The semantics of "sendw()" is that once control is returned from the subroutine call to the calling program, the message pointed to by "buf" has been buffered by the system, and that the calling program's copy may then be safely modified. In particular, return of control to the caller implies nothing about the ultimate reception of the message by the receiver. In a loosely coupled system, it is useful to relax the traditional meaning of synchronous message passing, due to the large latencies involved. For "recvw()", return of control means that "buf" points to a valid message of the requested type.

Each node maintains an independent clock, which is synchronized once with other node clocks during cube initialization. The clocks are readable by a node program via the function "clock()", with a resolution of 5 msec. The clock resolution level has implications for validation of simulation predictions, since the accuracy with which programs may be timed is limited by this resolution. Validation issues are explored in more detail in Chapter 6.

All nodes are connected via a 10 Mbit/sec global channel to a manager, an 80286-80287 based system running the XENIX operating system in a timeshared, multi-user mode. There is no direct I/O between the node and any I/O device connected to the manager. Messages may be logged to a special "log" file over the global channel to main memory on the manager via a call to the "syslog()" primitive. For programs running on the manager, a modified set of message passing primitives are provided:
sendmsg(ci,type,buf,len,node,pid)
int ci,type;
char *buf;
int len,node,pid;

recvmsg(ci,type,buf,len,cnt,node,pid)
int ci;
int *type;
char *buf;
int len,*cnt,*node,*pid;

The manager primitives are almost identical to the corresponding node primitives. The only difference is that the type parameter for "recvmsg()" is a return value, not a passed parameter. When the manager performs a receive, it must receive the next message regardless of type, which may only be determined after it is received.

Both manager and node programs are written, compiled, and linked on the manager. Manager processes are then executed by the XENIX OS; node programs are loaded by XENIX onto a specific node and execution is started. An application may either be loaded and run on the cube directly from the command line of the XENIX shell on the manager, or it may be written as a cooperating set of processes that run on both the cube and the manager. In this case, a manager process starts executing and dynamically loads processes onto the cube under program control via the "load()" system call.

The CSIM model of the hypercube architecture used in the RPPT implements a data-link layer simulation for message passing. The model provides a definition of the UserSend procedure (see Section 6.1 of Chapter 3). For each node, queues are maintained for incoming packets, outgoing packets, incoming messages, and outgoing messages. A manager process runs on each node to route packets through the queues.
correctly. Incoming messages are dequeued and packetized, and each packet is placed into an outgoing packet queue. Packets to be forwarded are merely moved from an incoming packet queue to the correct outgoing packet queue.

2. Programs

This section describes the programs used for the validation studies presented here. They include two linear algebra algorithms -- LU, an LU decomposition program, and EIGEN, an eigenproblem solver -- and an FFT implementation. The programs are all numerical in nature, yet they still span a wide range of behaviors. LU and FFT are single precision floating point algorithms, while EIGEN uses double precision. LU and FFT are direct solution algorithms (i.e., the number of steps required is a deterministic function of the size of the input), while EIGEN features some iterative subparts (i.e., the number of steps required involves a dynamic convergence criterion check). Some steps of the algorithms are purely computational (i.e., require no communication between nodes), while others have small computational loads and frequent communication requirements. Most communication in all the algorithms is nearest-neighbor only, except for the "psytre" step of the EIGEN program. Program layout is similar from application to application, and is discussed in more detail in Chapter 6.

2.1. LU Decomposition

The LU program was developed for the Intel iPSC Hypercube at Oak Ridge [23]. This program solves the linear system represented by the matrix equation
\[ A x = b \]

where \( A \) is a square \( n \) -by- \( n \) dense real matrix, and \( x \) and \( b \) are real column vectors of length \( n \). LU decomposition is a formulation in matrix theoretic terms of the familiar Gaussian elimination procedure for solving systems of linear equations. The algorithm proceeds by factoring (decomposing) \( A \) into matrix factors \( L \) and \( U \), and then solving for \( x \) via forward and back substitution. That is,

\[ A = LU, \]

where \( L \) is lower triangular with diagonal elements unity, and \( U \) is upper triangular. Forward substitution is then used to solve the system

\[ Ly = b \]

for \( y \), and back substitution solves the system

\[ Ux = y \]

for \( x \). The program follows a common practice in numerical linear algebra algorithms of overwriting the input matrix \( A \) with the two factors \( L \) and \( U \). The two factors are accommodated within the original storage for \( A \) by not explicitly storing the diagonal elements of \( L \), since these are understood to be unity. Discussion of the algorithm will refer to the decomposition as the "lu" step, and the back solve as the "bk" step.

The algorithm features partial pivoting for numerical stability and load balancing steps for enhanced hypercube performance. In standard Gaussian elimination, at the \( i \)-th iteration, rows \( i + 1 \) through \( n \) are modified by subtracting off some multiple of row \( i \) such that the \( i \)-th element of the row equals zero. For row \( j \), the multiple is \( a_{ji}/a_{ii} \). The element \( a_{ii} \) is called the pivot for this iteration. For matrices with elements whose values vary over a wide numerical range, division by a relatively small pivot can introduce
numerical instability. For this reason, partial pivoting is introduced, whereby at iteration $i$, rows $i$ through $n$ are examined, and the row with the largest value in the $i$-th position is exchanged with the $i$-th row. The row with the pivot element is called the pivot row.

The algorithm provides two strategies for distributing data across the nodes. In the "rsrp" (row storage row pointers) version, which is considered here, the rows of matrix $A$ and the right-hand-side vector $b$ are distributed to processors according to a wrap mapping, i.e.,

$$A[i][*] (\text{row } i) \rightarrow \text{node } i \mod p$$

where $p$ is the number of processors. Nodes $0$ through $(n \mod p)$ receive $(\left\lfloor \frac{n}{p} \right\rfloor + 1)$ rows, and nodes $(n \mod p)$ through $(p - 1)$ receive $(\left\lfloor \frac{n}{p} \right\rfloor)$ rows.

For a matrix of dimension $n$, the decomposition phase requires $n-1$ elimination or reduction steps. At step $i$, pivot determination proceeds by each node first determining a local candidate, followed by a global communication step to determine the global candidate. The node possessing the pivot row then broadcasts it to all other nodes. When the pivot row is received, each node reduces its participating rows (rows $i+1$ through $n$) by subtracting off the appropriate multiple of the pivot row. The decomposition phase completes after $n-1$ iterations of this process.

Once a row is identified as a pivot row in some step of the decomposition, that row no longer participates in the decomposition. This means that if one node finds itself with a disproportionate number of the rows destined to become pivot rows in early steps of the decomposition, once that node's rows have all become pivot rows, that node will then sit
idle for the rest of the decomposition step, degrading the algorithm's effective degree of realizable parallelism. In order to prevent this, the algorithm enforces a form of load balancing for pivot rows. Counts of the pivot rows on each processor are maintained, and whenever a new pivot row is chosen, a check against the counts for that node is made. If there is an uneven distribution of pivot rows across processors, two nodes will swap rows to balance the counts. In this way, all nodes retain approximately equal numbers of participating rows at each step of the decomposition, allowing each node to remain active throughout the computation.

The algorithm is divided into a manager process that runs on the manager and a node process, one instance of which runs on each available node. The manager process creates the input matrix $A$ and vector $b$ by either reading them in from a disk file or generating them from random number draws, loads each node with a node process, divides the matrix up and sends columns to nodes as messages, and waits to receive the solution vector $x$. Source listings for the program can be found in the appendix, and a discussion of the LU decomposition process can be found in [16].

2.2. Eigenvalue-Eigenvector Calculation

The eigenproblem code is a translation into C of several subroutines from the EIS-CUBE package, a parallel version of the EISPACK FORTRAN library for the Intel iPSC hypercube [36]. The code takes as input a real dense symmetric matrix of dimension $n$, and produces, if possible, all the $n$ eigenvalues and corresponding eigenvectors. The three principal steps in the algorithm are (1) a parallel symmetric tridiagonal reduction (subroutine "psytre()"), (2) tridiagonal bisection (subroutine "tridib()"), and (3) a parallel
symmetric QR iteration (subroutine "psytrr()"). These three steps will be referred to by their subroutine names. A discussion of the eigenproblem can be found in [24] or [37].

The algorithm starts with the manager distributing columns of the matrix to the nodes in a wrap fashion. In the psytre step, each node operates on its $n$ -by- $m$ column slice of the matrix and reduces it to a tridiagonal form by a series of $n-1$ Householder transformations. Each transformation requires each node to perform a vector gather and broadcast. Each node computes a subset of the elements of the diagonal and off-diagonal. A gather operation is then performed to give each node a copy of the complete diagonals. The tridiagonalization can be described by the matrix relation

$$H = Z A Z'$$

where $A$ is the original real symmetric matrix, $H$ is the tridiagonal result, and $Z$ is the unitary but not symmetric transformation matrix. The "psytre" step computes $Z$ as a series of matrix products, one product per iteration.

$$Z = P_{n-2} P_{n-3} \cdots P_1$$

Each $P_i$ is an elementary Hermetian matrix that effects a Householder's transformation for column $i$. Each $P_i$ is unitary and symmetric.

In the "tridib" step, the $n$ eigenvalues of the tridiagonal matrix are estimated. Each node starts with the complete diagonals and computes $n/p$ of the eigenvalues independently from the computation on the other nodes. Another gather operation is then performed to give each node a copy of all eigenvalues.

In the "psytrr" step, eigenvector values are determined. This step starts with the diagonals of the tridiagonal and the Householder transformation matrices computed in
the "psytre" step, and the eigenvalues computed in the "tridib" step. It then performs an implicit QR iteration on the tridiagonal matrix. This step initially reintroduces some nonzero elements into the matrix for elements not on the main three diagonals. The iteration continues until the values of these offdiagonal elements are again negligible. Each node produces its contribution to the eigenvector calculation independently from the other nodes, so no communication is required.

2.3. Radix 2 FFT

The radix 2 FFT is an algorithm for calculating the discrete Fourier transform \( H(f) \) of some function \( h(t) \). The discrete Fourier transform is defined by the relation

\[
H(f_n) = H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i kn/N}
\]

In general, \( h(t) \) is a complex-valued function of time \( t \), and \( H(f) \) is a complex-valued function of frequency \( f \). The input to the algorithm is a set of \( n \) complex numbers \( h(t_n) \), the value of the function \( f \) at \( n \) evenly spaced times \( t \),

\[ t_n = \Delta n, \quad n = 0, 1, 2, \cdots N-1. \]

The output is \( n \) values of the transform

\[ H(f_n), f_n = \frac{n}{\Delta N}, \quad n = 0, 1, 2, \cdots N-1. \]

A general description of the algorithm may be found in [37], and more details on the RPPT application implementation are in [33].
CHAPTER SIX

Validation

1. Introduction

The RPPT environment has been designed to support investigations into the execution behavior of a wide range of programs and architectures, both real and hypothetical. One typical scenario that meshes well with the intended use of the testbed is to take a specific program and consider its performance on a series of closely related architectural variants. It is likely that only some (or even none) of these architectural variants exist. As a first step toward making such predictions plausible, we must establish how well the RPPT predictions fit the empirical observation of real applications running on real systems. For example, suppose the performance for a program on a system to which we have access (such as the 16-node hypercube here at Rice) is known, and it is desired to know how much performance improvement could be expected for that same program running on a 32 or 64 node version, to which we do not have access. The testbed can easily generate such performance indices, but in order to establish a level of confidence in these results, predictions for the 16-node machine should first be compared with the observed performance of the real program on the real system. If we can show that the RPPT prediction matches well with observation for the existing system, we can be confident that the predictions for the hypothetical (or at least unavailable) architectural variants are meaningful. We can then say that this particular RPPT application, this combination of program and architectural models, has been validated against
observation of the real system. Validation of the testbed as a whole will be only as successful as the validation of individual models.

As mentioned in Chapter 1, an important goal of this thesis has been to present results that partially fulfill the larger goal of the complete testbed validation. This chapter presents validation results for a subset of the testbed tools and application models. First, individual tools -- timing profilers and architecture simulators -- are validated. Second, complete RPPT applications are run and compared with actual program executions, the predictions and observations being broken down, whenever possible, into constituent contributions from profilers and simulators. Such an approach will minimize the potential for compensating errors, or if they are present, will help to gauge the magnitude of the effect. The results will establish criteria for closeness of fit between prediction and observation, combined with explanations to account for discrepancies.

The chapter starts with a discussion of the scope of applicability of the results in Section 2, followed by a description of measurement methodology in Section 3, with results presented in Section 3, and concluding with a discussion of results in Section 4.

2. Scope of Validation Results

Strictly speaking, the scope of any collection of validation results extends only to systems with characteristics that are similar to those systems against whose execution behavior predictions have been compared and reported. The results in this thesis, for example, focus on a collection of matrix-theoretic algorithms running on a 16-node hypercube, with some discussion included for an FFT algorithm, and preliminary results
for modeling programs running on the TI 32020. The scope of the results may be extended, with caution, to any similar loosely coupled system running "similar" programs.

In the next section, the measurement methodology used for generating the necessary hypercube validation results will be discussed. Measurement methodology is of necessity in part specific to a particular machine or class of machines. For example, the execution time for program running on a loosely coupled multiprocessor system whose independent processes communicate via message passing may be characterized by a certain categorization such as total time, computation time, and communication or synchronization time. Some of the quantities and techniques for their measurement may apply only to the specific class of systems for which it was designed. In contrast, a different class such as a tightly-coupled shared-memory system would require new categories such as cache coherence time (i.e., time spent by the system in maintaining cache coherence). Nevertheless, the specific nature of the measurement methodology required for one validation study should not be seen as an indication of some lack of general applicability for testbed results. The testbed remains suitable for study of a wide class of architectures and programs which are merely outside the scope of the investigations reported here.

3. General Measurement Methodology

This section lays some groundwork of a systematic scheme for measuring the execution behavior of programs running on the hypercube. This methodology governs how to make timing measurements on the real system and how to interpret simulation data so
that the performance indices in each case are suitable for comparison with each other. We also discuss problems that limit the accuracy of measurements, or that lead to ambiguity of interpretation.

3.1. Instruction Profilers

The goal of the timing profilers is to capture in detail the contribution to overall program execution time due to instruction execution by the cpu's of the system under consideration. This contribution will be referred to as computation time, and it includes execution of all instructions derived from the user's application program. The profiler is responsible for taking into account such execution time effects as instruction pipelining, data caches, memory access latency, and floating-point coprocessor interaction. We will exclude some system code that gets executed on the cpu from the computation time category, reclassifying it as communication time, described below in Section 3.2. This is necessary because the system code in question is not available for profiling, and is difficult to isolate for simple timing measurement via the system's hardware clock, as can be done for some other system code. Measurement of a segment of code by reading a hardware clock before and after its execution and taking the difference in the readings will be called time profiling.

Code which is executed on the real system but which cannot be instruction profiled may still need to be time profiled. This can be the case for procedures from the math library, for instance, "sqrt()" or "log()". In some of the example applications presented here, source code was obtained and in effect made part of the application program. The system subroutine then gets profiled similarly to other application code. Time profiling is
used if access to the source code is not possible. Once a subroutine is time profiled, the observed time is then entered into a table of subroutine times to be used by the profiler. When the profiler encounters a call to this subroutine in a program, the execution time estimate for the current basic block is incremented by the tabulated time. The profiler in effect treats the whole subroutine call as one instruction. On the hypercube, code of the following form is usually sufficient for time profiling.

```c
extern long clock();
long start,stop;
int i,lim;

...

start = clock();
for (i=0;i<lim;i++) subr(...);
stop = clock();

time = (stop-start)/lim;
```

Depending on the complexity of the subroutine being time profiled, from 100 to 10,000 subroutine calls may be required for an accurate estimate of the cost per call. Even for long observation intervals, problems can still arise. Time profiling is vulnerable to clock resolution problems and data dependent execution times. It may be necessary to execute the subroutine over a range of input parameters to get a more representative average value. Even then, an average value may not describe the subroutine execution time adequately, especially if there is a large data-dependent execution time variation.
3.2. Message Passing and Architecture Simulation

In addition to time that can be charged to the execution of the instructions of the user's program on the cpu, time also elapses during information exchanges between nodes via message sends and receives. We will call this quantity communication time, which we break down as follows.

- **software overhead** -- execution of instructions in send and receive message passing primitives
- **synchronization time** -- idle process time spent waiting on a message to arrive
- **transmission time** -- time required to send a stream of message bits across a physical channel

Transmission time is not directly charged to a cpu in the architecture, and is in fact overlapped with synchronization or execution time on other cpu's, but its duration can affect the synchronization delays experienced at other cpu's.

Communication time in an RPPT simulation is modeled by specifying software overhead and transmission times so as to match the empirically observed execution time behavior of the real system. Synchronization time, in contrast, is not directly under the control of simulation parameters, but is instead determined by the simulation time ordering of process suspensions and activations. Transmission time is specified by the simulation description of the architecture interconnection, which is provided in the "User-Send()" procedure (see Chapter 3). Software overheads are established for important procedures, such as the message passing primitives, by direct observation of the execu-
tion of simple message passing application programs on the system being studied. Here again, we must substitute observed times for profiling estimates for code which is executed by the cpu but for which source code is not available for profiling.

3.3. Complete RPPT Applications

3.3.1. General Program Layout

Application programs for the hypercube generally conform to a common layout or physical organization. First, typical programs feature a single process, called the manager, running on the cube manager system. This process provides the user and disk I/O interface for the application. It reads in data, loads processes onto the nodes, sends startup messages and data, and waits for results. Second, the applications feature a single process, called a node process or node driver, running on each of the available nodes. The driver waits for a startup message and data from the manager, synchronizes with other nodes, calls subroutines to operate on data or exchange messages with other cube nodes, sends final results back to the manager, and terminates. This arrangement can be described in high level C-like code as follows.
cube manager: main() {
    read command line parameters;
    allocate storage for data;
    for (i=0;i<procs;i++) {
        load(i,"driver");
        send(i,startup);
        send(i,data);
    }
    for (i=0;i<procs;i++) {
        receive(results);
    }
    print out results;
}

node driver: main() {
    receive(startup);
    allocate space for data;
    receive(data);
    synchronize();
    ...
    computation and communication between node drivers;
    ...
    send(mgr,results);
    exit();
}

In the driver process, it is helpful to have the node processes synchronize among themselves before measurement of execution time begins. Otherwise, the measurement on
some nodes will be biased by the delay incurred by the serial loading and activation of
node processes onto the nodes by the manager. That is, processes loaded earlier will
experience much greater delays than than processes loaded later.

As a matter of general C coding style, RPPT application programs are written in a
way so as to minimize block mismatches in the cross profiling step (see Sections 4 and 5
Chapter 4). In practice, this means replacing case statements with the equivalent "if-
then-else" construct, trying to avoid "if-then" constructs without a matching "else",
replacing the conditional operator "?:" with the equivalent "if-then-else", and avoiding
excessively complex expressions that exhibit internal basic block structure of their own.
The rationale for such transformations is (1) to minimize the use of constructs that are
treated differently by different compilers, and (2) to ensure that all basic blocks get block
markers for matching purposes.

3.3.2. Timing or Measurement of Hypercube Applications

Although the underlying profiling and simulation methodology of the RPPT
environment is generally applicable to a wide range of architectural and program types,
timing or measurement methodology must be tailored to the characteristics of the specific
system being studied. The fact that an application is intended for execution on a loosely
coupled system such as the hypercube defines the performance indices we are interested
in measuring on the real system and predicting in the simulated system. In particular, for
hypercube applications, the principal figure of interest is the application's total execution
time, which will be broken down into computation time and communication time, as
described above in the section on general measurement methodology.
The cost of a hypercube application that adheres to the general program layout described above will be the execution time, as measured on some node, from the synchronization point after the startup message until the end of the node driver. The point at which node processes start their timing is significant because of the possibility of serialization time effects when node processes are downloaded to the cube and activated. If the manager process serially loads processes onto each node, the global time of activation of each node driver will be significantly staggered because of the time expense of the load operation. If timing measurements on the node are started immediately, nodes loaded first will execute ahead, up to the first synchronization point, where they will block, waiting for later processes to catch up. Such an effect will cause extra time, which we can call serialization time, to be charged to the process total time and synchronization time, although not affecting the process computation time. Since process downloading is not modeled by the testbed, this extra time will not appear in the simulation estimate.

This same serialization effect occurs to a lesser degree even if execution time measurement is not started until after the node drivers synchronize between themselves after activation. Depending on the synchronization scheme used, one node might be singled out to bear more of the cost of the synchronization step, getting a late start on the algorithm with respect to the other drivers. Still, such an error is much less than the analogous error if no synchronization step is used. Accuracy of determining a representative execution cost can be improved by (1) taking averages over all nodes, and (2) averaging this average over multiple independent runs of the program. The first averaging helps neutralize whatever small serialization effect remains. The second averages over any
small statistical fluctuation that may be experienced in the system hardware state.

For the algorithms presented here, total time is measured individually for each major subpart of the node driver execution. For example, the eigenvector-eigenvalue application node driver has the following layout.

driver(...) {

    ...
    start = clock();
    psytre(...);
    psyttETIME = clock() - start;
    synchronize(...);

    start = clock();
    tridib(...);
    tridibtETIME = clock() - start;
    synchronize(...);

    start = clock();
    psyttqr(...);
    psyttqrTIME = clock() - start;
    ...

}  

Again, this arrangement is to avoid timing those parts of the program involved in such activities as data structure initialization or data I/O, program fragments which have only a superficial contribution to the computational behavior of the algorithm, but which would be difficult to model in the simulation.

Once total time is determined, we must decompose it into computation and communication components. That part of the total time which is due to communication is found in an independent measurement, and the difference between total and communica-
tion time is assumed to be computation time, with the exception of a small probe effect
generated by frequent calls to clock to measure the necessary time subintervals. For
applications presented here, all communication between processes is achieved through
sends and receives. Furthermore, these are the only subroutines called in which time that
is not modeled by the timing profilers elapses. Therefore, if we can measure the cumula-
tive time spent by some node program inside send and receive, then all remaining time
can be attributable to code which in the simulation is profiled. This is done by accumu-
lating over many short intervals the time spent inside send and receive. For example, a
code fragment with communication such as this:

```plaintext
...  
cdim = 1;  
while (cdim<np) {
    if (me<cdim) {
        send(...);  
    }  
    else if (me<2*cdim) {
        receive(...);  
    
...  
```

is augmented by interval timing code to this:
cdim = 1;
while (cdim<np) {
    if (me<cdim) {
        start = clock();
        send(...);
        commtime += clock() - start;
    }
    else if (me<2*cdim) {
        start = clock();
        receive(...);
        commtime += clock() - start;
    }
}

Then, for each algorithm or subalgorithm being timed, a total, communication, and computation time (= total - communication) are collected.

The number of clock reads can slightly perturb the results for programs of short duration. The number of clock calls can be counted, then an extra factor can be subtracted from the total time when calculating the computation time. That is, instead of using

\[
\text{computation} = \text{total} - \text{communication};
\]

we use

\[
\text{computation} = \text{total} - \text{communication} - (\text{clock reads})*(\text{clock time});
\]

The time for a single call to clock() can be time profiled as for other subroutines which cannot be instruction profiled.

start = clock();
for (i=0; i<1000; i++) clock();
time = (clock() - start)/1000;
The estimation of cumulative communication time by summing up over many such small intervals is also vulnerable to error due to lack of sufficient clock resolution. The hypercube "clock()" function returns values rounded off to the nearest 5 msec. Some programs have been run for which the cumulative value of the communication time over hundreds or thousands of individual intervals amounts to only tens of milliseconds. For such programs, there is sufficient uncertainty of measurement of individual intervals on the real system that it becomes difficult to say anything meaningful about the quality of the simulation predictions.

4. Results

4.1. Accuracy of TPROF Timing Profilers

The TPROF timing profilers, described in Chapter 4, have been developed and validated on both simple contrived programs and on complete real applications. Dynamic cycle count profilers suffer from certain intrinsic problems that limit the accuracy of their predictions, such as

- instructions whose execution times are a function of their operand values, such as shifts and microprogrammed character manipulation instructions, and

- instructions and data in caches and pipelines.

Cache and pipeline effects can be compensated for in a simple way if the timing documentation for the processor's instruction set provides explicit estimates of these effects on execution time. In general, however, there is no way to predict with absolute certainty at compile time the number of cycles that an arbitrary instruction will require to execute
at run time. Indeed, for a pipelined processor, it becomes problematic to assign a given execution cycle count to a given instruction because of instruction execution overlap.

4.2. Standalone Profilers

The first step in checking the accuracy of a standalone profiler is an analytic one. We take a simple program and analyze it to find its associated basic block graph, with edges representing flow of control and vertex weights representing execution cycle counts. If the program is sufficiently simple, say a series of control flow loops with the number of times through the loop determinable at compile time, we may derive an analytic expression for the cycle count cost of the program in terms of the vertex cycle count weights and the number of times through the loops. This expression can be compared with the cycle count estimate generated by the execution of the profiled code. This step acts as a check that the profiled code is computing the correct desired function of basic block cycle counts. See Figure 6.1 for an example.

The next validation step is to take a simple sequential program and compare the profiler prediction with actual measured execution time on the real system. This step measures how accurate an execution time prediction can be expected from the profiler's table-driven lookup of instruction times for an assembly language program. This section presents validation results for two standalone profilers, one for the SUN 3 (68020-68881 based) and one for the Intel iPSC hypercube (80286-80287-based).
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        body
    }
}

source code

basic block structure

cycles(n) = (c3+c2)*n*n + (c2+c1+1)*n

analytic cycle count expression

Figure 6.1
analytic cycle count profiling
4.2.1. 68020-68881 Profiler for the SUN 3

Accurate profiling for the 68020 is affected strongly by its relatively complex architecture. The chip features an on-chip data cache and an instruction pipeline. Depending on the state of the processor and the effectiveness of pipeline and cache operation, the time required to execute an arbitrary instruction in the middle of a program execution may fall within a wide range of values. As described in Chapter 4, timing documentation for the 68020 provides a detailed analysis of the timing behavior of all instructions for three different idealized processor states or cases: best, cache, and worst. For cases that feature instruction overlap, timing documentation maintains a consistent scheme for assigning a given cycle to one of several possible overlapping instructions, so that the number of cycles required for series of instructions may be computed by summing up the table values for each individual instruction, without the need for the user of the table to calculate overlap himself.

The SUN 3 systems on which validation work was performed also feature an MC 68881 math coprocessor. Instruction times for the coprocessor already take into account the typical amount of overlapped execution time with the main processor that can be expected, so again the user need not calculate this overlap himself. As will be seen in the discussion of the 80286-80287 profiler in the next section, this is not always the case. The interaction between the 80286 and 80287 follows a more complex protocol, and the profiler itself must estimate overlap of execution time between the two chips.

The 68020 validation was performed on two differently configured SUN 3 systems:
• a 3/50 with a 15 MHz 68020, a 15 MHz 68881, and no external cache, and
• a 3/280 with a 25 MHz 68020, a 15 MHz 68881, and an external cache.

Results of the validation for a simple program and a global distribution local sort (GDLS) sorting program are given in Figures 6.2 and 6.3. Figure 6.2 presents results for execution of the following programs on both the 3/50 and 3/280.

```c
main() {
    register int i;
    register long x;
    for (i = 0 ; i < 2000000 ; i++) x++;
}
```

```c
main() {
    int i;
    long x;
    for (i = 0 ; i < 2000000 ; i++) x++;
}
```

The first program increments a variable in a loop two million times, with all data in registers, so that only instruction fetches require memory references. The second program is identical except that variables are moved from registers onto the stack, so that operands also must now be fetched from memory.

The top graph in Figure 6.2 shows results for the register version. Profiler predictions for each of the three cases for both the 3/50 and the 3/280 are divided by the actual measured time. A prediction that approaches unity is the best predictor. On both machines, the cache case prediction falls very close to the measured time. In the bottom graph, the results for the stack version of the program are given. We see that perfor-
Figure 6.2
SUN 3 standalone profiler
validation results
GDSL on SUN 3/50

worst

cache

best

prediction normalized to measured time

GDSL on SUN 3/280

worst

cache

best

prediction normalized to measured value

Figure 6.3
SUN 3 standalone profiler validation results for GDLS
mance on the 3/280 remains similar to the previous program. That is, the cache case remains the best predictor. On the 3/50, the best predictor is about halfway between the cache and worst case numbers. That is, actual time on the 3/50 is moving closer to the estimated worst case time.

We can infer that the difference in performance is due to the presence of the external data cache in the case of the 3/280. As variables are moved out of registers into memory on the 3/280, the external cache captures them and prevents any appreciable degradation in performance, keeping execution time near the cache case. On the 3/50, however, the cache is not present, and an extra wait state for each memory reference is required. To be more exact, the profiler should be modified so that an accurate count of total memory references for each instruction is maintained, and the cycle count estimate for an instruction on the 3/50 should be incremented by one for each such reference.

Figure 6.3 displays results for a global distribution local sort (GDLS) algorithm. The top graph gives results for the 3/50 and the bottom for the 3/280. Again, the results show that the cache case profile estimate is the best predictor for the 3/280, and that if extra cycles were added to the estimates for the 3/50 to account for memory wait states, the cache case would be a more accurate predictor for the 3/50 as well.

4.2.2. 80286-80287 Profiler for the Intel iPSC

The 80286 is a simpler architecture in some respects than the 68020, especially from the perspective of assigning times to individual instructions. There is no on-chip cache, and instead of an instruction pipeline, there is only a 6-byte instruction prefetch
buffer. Documentation for instruction times is correspondingly more straightforward.

As a simple example, consider the following program.

```c
int i, j; /* case EXT */

main( ) {
    int i, j; /* case LOC */
    register int i, j; /* case REG */
    for ( i = 0 ; i < 30000 ; i++ ) {
        j++;
        j--;
        ...
    }
}
```

Three versions of the program were considered, the versions being distinguished by the location of variables used in the program. Version REG uses local variables in machine registers, version LOC uses local variables allocated in memory on the stack, and version EXT uses global variables, which are located in a different segment. Validation results are given in Figure 6.4. Three versions of the program with different numbers of statements -- 2, 10, and 200 -- inside the inner "for" loop are also considered for each variable location version.

Results give the percent difference between profiler prediction and actual measured time. Positive percent differences indicate a profiler prediction that is lower than the real execution time. For REG and EXT versions of the program, results show that as the "for" loop body becomes longer, the predictions improve. This is most likely because the
Figure 6.4
iPSC standalone profiler validation results
longer loops mitigate the performance degradation effect of instruction prefetch buffer
flushes that are required whenever a successful branch occurs.

The LOC version, in contrast, shows an increased percent error as loop size
increases, with the error probably asymptotically approaching a steady state value of
around 8%. Earlier profile results showed a relatively large percent difference for ver-
sion LOC. Since versions REG and EXT showed small percent errors, we reason that the
source of the error is due to contention for bus access between instruction and operand
fetches. A correction of adding one extra cycle for every memory reference changes the
agreement between measured time and prediction from 12% or 15% to 8%, with the
remaining error likely reflecting a further small shortcoming in the modeling of memory
bus contention. The EXT version of the program makes references to memory as well,
but in this case, the much longer operand access time keeps contention from developing
as it does in the LOC case. EXT memory references take longer than LOC memory
references, because in the EXT case, a segment register must be loaded with the base
address of the memory segment holding the variable being referenced.

4.3. Cross Profilers

Several cross profilers, whose design was discussed in Chapter 4, have been
developed as RPPT library tools, and others are currently being built. These include
68020-80286, 68020-V System, and 68020-32020 cross profilers. Once a standalone
profiler is validated against real execution time, it remains to demonstrate that no addi-
tional loss of accuracy is incurred by moving from the standalone to the cross profiler. In
general, the inaccuracy introduced by the cross profiling scheme results from basic block
mismatches that occur in characteristic ways for the two compilers. Chapter 4 discusses in detail how these mismatches arise.

4.3.1. 68020-80286 Cross Profiler

Figures 6.5, 6.6, and 6.7, compare cycle count results between the 80286 standalone profiler and the 68020-80286 cross profiler for three sequential application programs, fft, GDLS, and an eigenproblem solver, respectively. Each application was run for a series of data sizes. For all applications, the error introduced by cross profiling is always less than one percent, and for the examples other than GDLS, the error is less than .2 of one percent. The primary source of error for the GDLS case is the block mismatch caused by "if" constructs without an "else" clause. One can imagine a pathological code arrangement such as

```c
for (i=0;i<BIGVALUE;i++) {
    if (test) {
        short code fragment;
    }
}
```

if the "for" loop contains nothing but an "if" without a corresponding "else", and if the "for" loop constitutes a large fraction of the computation of the program, then the profiling error cause by the block mismatch for the "if" construct gets magnified by the large number of "for" loop repetitions.

A slight effect of this sort is being seen in the results for the GDLS application, but since this is not necessarily representative of typical application code, the profiler will on the average perform better. Even if the error were as large as one percent, we could still
Figure 6.5
SUN 3 - iPSC cross profiler
sequential fft results
Figure 6.6
SUN 3 - iPSC cross profiler
sequential GDLS results
Figure 6.
SUN 3 - iPSC cross profiler
sequential eigenvalue-eigenvector results
conclude that the error introduced by the cross profiler represents one of the smallest contributions to the overall prediction error from any of the RPPT modeling tools.

4.3.2. 68020-32020 Cross Profiler

This profiler, as discussed in Chapter 4, introduces an extra complication in the block merging phase. In comparison to the 68020-80286 cross profiler, the 32020 compiler produces code in some instances with a fundamentally different basic block structure than that from the 68020 compiler. Fortunately, the matching scheme presented in Chapter 4 is very successful, resulting in a relative error that is a small fraction of one percent.

4.4. Message Passing and Architecture Modeling

The applications considered in this thesis rely almost exclusively on nearest-neighbor message passing communication. The one exception is the lu step in the LU decomposition. This step uses an occasional arbitrary node to node exchange of pivot rows to enforce load balancing as described in Chapter 5. The differences between nearest-neighbor and non-nearest-neighbor communication are outlined below.

In nearest-neighbor communication, the message routing features of the hypercube node operating system (NOS) are not brought into play. In this scheme, node i communicates only with nodes j for which i XOR j treated as a binary number has a single bit set. Algorithms that require only global broadcast and gather communication steps are good candidates for arrangement into a nearest-neighbor communication organization. In the algorithms considered here, this is done by mapping a minimum spanning tree onto the
cube, then implementing the broadcast and gather recursively, with sends and receives by a given node at any recursion depth involving only directly reachable neighbors. Code listings for broadcast (subroutine bcast) and gather (subroutine incast) can be found in the appendices.

The real system execution time advantage of such an organization has not been investigated here. The modeling advantage with respect to RPPT applications is that the timing effects of message forwarding can be avoided. The timing effects of non-nearest-neighbor communication currently have the potential to degrade prediction accuracy because of the extra processor activity required on intermediate nodes to forward messages. Currently the RPPT simulation environment makes no attempt to solve any processor scheduling problems. Nodes are seen as dedicated to basically a single process, with possibly an occasional short-lived auxiliary process. Without any modeling of processor time slicing, simulations of message forwarding activity will not be able to correctly estimate some synchronization time intervals. Also, since the message forwarding activity is performed by code which is not available for profiling, its execution time will have to be estimated.

There are two obvious enhancements for addressing the modeling shortcomings described above. First, the simulator can be given the ability to model interrupt service intervals. Second, processes running on nodes can be forced to time slice. This time slicing feature solves other modeling problems as well within the RPPT.
4.5. Validation of Complete RPPT Applications

Timing results for the complete RPPT applications are given in Figures 6.8 through 6.13. The general format for reporting the times is to present three pairs of graphs for each figure. The pairs give results for computation time, communication time, and total time, as defined earlier in the chapter. In each pair, the first graph gives the comparison of measured time with prediction for a range of problem sizes -- matrix dimension or log of the number of points in an FFT. The second graph plots the percent difference between prediction and observation on a point-by-point basis. A positive percent difference indicates a prediction that is smaller than the observation. The following sections give a presentation and discussion of the results for each application in turn. For technical details on the programs, see Chapter 3. Program listings can be found in the appendices.

4.5.1. LU Decomposition

The LU decomposition divides into two main subparts: the LU decomposition itself, which we will call "lu", followed by a backward and forward substitution step, which we will call "bk". The timing results for "lu" are given in Figure 6.8, and those for "bk" in Figure 6.9.

For small matrices, communication time is a large fraction of overall execution time, while for larger matrices, computation time is dominant. This is because each node is given \( n/p \) rows of the original matrix by the cube manager. The computationally intensive segment of the lu step is in reducing all the unreduced rows on a given node by
Figure 6.8

lu: prediction vs. measurement
Figure 6.9
bk: prediction vs. measurement
Figure 6.10
psytre: prediction vs. measurement
Figure 6.11
tridib: prediction vs. measurement
Figure 6.12
psyqtr: prediction vs. measurement
Figure 6.13
fft: prediction vs. measurement
the pivot row chosen in this step. As the number of rows per node grows large, a relatively greater amount of computation can be performed between successive pivot row transmissions.

For the lu step in Figure 6.8, all predictions are generally within 5%, except for the case of the smallest data size considered, a matrix of dimension 16. For this case again, the total execution time is on the order of tens of milliseconds, small enough that clock resolution problems render the timing measurements suspect.

For the bk step, in Figure 6.9, we have an algorithm that is mostly communication, for which prediction is within 5% of actual time, except for the smallest case when it is slightly more than 10%. The computation time of this application, however, is harder to measure accurately, appearing to be particularly sensitive to the amount of clock read correction time that must be subtracted from the results, as described above. Two curves for the measured execution time are given to display the magnitude of this effect. Clock read time is not subtracted from curve "real1", and is subtracted from curve "real2". We feel confident that if a more accurate determination of the cost of clock reads could be made, agreement would be closer.

4.5.2. Eigenvalue-Eigenvector Calculation

The eigenproblem solution breaks down naturally into three components: (1) tridiagonal reduction (psytre), (2) eigenvalue estimation (tridib), and QR iteration (psytqr). 
Psytre results are given in Figure 6.10, tridib in Figure 6.11, and psytqr in Figure 6.12. The three parts of the algorithm represent a useful range of program behaviors. The psy-
The step is a conventional matrix algebra decomposition step that requires a fixed number of steps for a given input problem size. The tridib and psytr step are iterative in nature and require a convergence criterion check for termination. The psytr step is totally sequential on each node. As with the lu step in LU decomposition, the psytr step is almost purely communication for smaller matrices, while for large matrices, the computation contribution begins to dominate the overall time. This step is performed on each node with no communication between nodes required. The only timing measurement reported is therefore total time, which is equivalent to computation time for this step.

For step psytre, Figure 6.10, computation time is modeled adequately, the percent error asymptotically approaching some value around 9%. Communication time is also acceptable, although some of the simplifying assumptions made by simulation with regard to message packetization are probably beginning to break down for the larger data sizes. Results for the tridib step, in Figure 6.11, show good agreement for all curves. There is one poor prediction for communication time for the smallest test case, but this is almost certainly another measurement resolution problem.

4.5.3. Radix 2 FFT

The FFT was run for only a small number of problem sizes. If smaller sizes are considered, the total execution time is sufficiently small to begin to give rise to measurement resolution problems, and the quality of the predictions becomes hard to evaluate. For larger sizes, the maximum size of an array in the 80286 C compiler grows too large for the memory model (large) for which the program was written. This could be rewritten for the huge memory model to accommodate the larger data sizes. Results for the
sizes that were considered are shown in Figure 6.13.

Computation time prediction accuracy is shown to vary from 7.5% to 8.0%, figures which are representative of the expected performance of the profilers. Communication time modeling, however, is relatively unsuccessful for this application. This is almost certainly due, however, to the small times that are being measured, giving rise to the clock resolution inaccuracies discussed earlier. Since communication time contributes a relatively small amount to the total time, the predictions for total time are also in the acceptable range, from 4% to 11%.

5. Discussion

The collective application results show that, in general, computation time is accurately predicted by the timing profiler to around 5%. The trend is for accuracy to increase as problem size and total computation time increases. Exceptions are bk (Figure 6.9), psytre (Figure 6.10), and fft (figure 6.13). For bk, the accurate determination of computation time on the hypercube is limited by clock resolution. The bk computation time is a relatively small percentage of the overall time, or in other words, the bk step is almost purely communication. The computation step time must be derived by taking the difference of two nearly equal numbers. Another problem that appears in the bk results is that the small times involved are very sensitive to the clock value used. Attempting to correct for clock time in fact makes matters worse in this case. The two columns in the second and third figure reflect the timing results, first correcting for clock time, and second ignoring it.
Communication time predictions for most results are also close to measured values, usually to within 5% as well. The FFT communication time predictions are more erratic, again probably due to the shortness of the times considered. In general, though, the assumption of exclusively nearest-neighbor communication holds, even for the case of lu, which is mostly nearest-neighbor, but occasionally uses arbitrary node to node message passing for pivot row load balancing.

Because of the nature of the systems and applications being studied, a separation of communication time into synchronization time and software overhead time is not possible. A more accurate breakdown of the contributions to this time would be possible if the operating system source code itself were available, allowing it to be profiled just like the application program. This may be possible for future work with the V system running on a network of workstations.
CHAPTER SEVEN

Conclusion

1. Summary

This thesis has presented evidence that the RPPT and its execution-driven simulation technique are viable, attractive, and competitive in comparison with previous techniques, sometimes making available performance information that cannot be produced at all by other techniques. Validation results for hypercube applications show that testbed results are meaningful and accurate. Development of extensions and enhancements to the original should now be straightforward, furthering the accuracy of its techniques and its range of applicability. As experience with model development and validation grows, the testbed should become more flexible, adapting to the appearance of new architectures and new parallel programming constructs and usage styles. This chapter discusses the quality of predictions in Section 2 and the scope of the predictions in Section 3, and ends with an evaluation of what are the most promising future directions for further testbed development.

2. Quality of Predictions

The predictive power of the testbed for total, communication, and computation times for a range of applications was demonstrated in Figures 6.8 through 6.13 in Chapter 6. In general, we see that total execution time is accurate to within 10%. Breaking this time down into computation time and communication time provides an opportun-
ity to validate individual components that contribute to the overall performance estimate. Problems can arise in the measurement process on the real system if the level of clock resolution (5 msec on the iPSC hypercube) is sufficiently small. Probe effect (changing the behavior of a system by instrumenting it for observation or timing measurements) can also arise, making it difficult to speak with confidence about the behavior of any program on the hypercube if the total time duration is smaller than 100-200 msec.

2.1. Concurrent C and TPROF: Modeling of Computation Time

Figures 6.2 through 6.4 show validation results for the standalone profilers in isolation, and Figures 6.5 through 6.7 show similar results for the cross profilers. Graphs labeled "computation time" in Figures 6.8 through 6.13 take the performance of complete applications and isolate the contribution from the TPROF profilers. For complete applications, standalone profilers are always accurate to within 10%, and are usually within 5%. Predictions are consistently low, implying that documented instruction times are consistently optimistic. The documented instruction times tend to discount the effects of certain interactions between instructions at runtime that increase actual execution times. Based on the validation results for the standalone profilers, it may be that the profilers are pushing against a fundamental limit to the accuracy of the method of around 5%. Any consistent improvement beyond this may require an expensive dynamic cycle-by-cycle simulation of program execution.

Cross profiling has proven to be a powerful and reliable component of the testbed, never adding more than a small fraction of 1% of error to the prediction in addition to the error already introduced by the standalone profiler. Errors in cross profiling are almost
all due to minor mismatches between basic blocks produced by the two compilers for the same source program. In many cases, these mismatches can be patched easily by hand to bring the error very close to zero. Development of better profiling tools to automate such corrections would make the process more reliable. The technique has been successfully for 80286, 80386, and 32020 assembly language targets mapping onto 68020 assembly language hosts, suggesting that the technique is robust, since it works for quite dissimilar target-host pairs. If a C compiler can be built for a system, almost certainly an accurate cross profiler can also be written.

2.2. CSIM and ASIM: Modeling of Communication Time

For the programs considered here, communication time is that time spent inside calls to the message passing subsystem, i.e., calls to "sendw()" and "recvw()". The total communication time for an algorithm or subpart of an algorithm is measured as the accumulation of many short subinterval measurements, as described in Chapter 6. Figures 6.8 through 6.13 show that communication time is modeled accurately to within 10%, often to within 5%, with the exception of 3 points. These points are off by large margins (about 30%-60%), yet they are each for programs of very short total duration. As discussed in Chapter 6, the problems are therefore almost certainly due to the inability to accurately measure the many small time intervals required with the low resolution clock.

Modeling of the communication requirements of the programs considered happens to be particularly straightforward. These programs rely almost exclusively on nearest neighbor communication between nodes, the exception being the "lu" step of the LU decomposition, which uses an occasional non-nearest neighbor message pass. Modeling
the timing behavior of nearest-neighbor message passing is somewhat simpler than the
gen-eral case, since no message forwarding by intermediate nodes is required. A large
amount of such forwarding would be perceived in the results as a degradation in the
speed of processes running on the intermediate nodes, since the cpu would be interrupted
for servicing the requests for forwarding activity. The ability to model interrupt-driven
activity in the simulation is currently hampered by the lack of sufficient process schedul-
ing control in Concurrent C. An adequate treatment in the RPPT of hypercube programs
that generate heavy message forwarding activity will have to wait until this enhancement
to Concurrent C is implemented.

Also, the size of the messages exchanged for the programs considered are relatively
small, almost always less that 1 kbyte. Longer messages passed on the hypercube are
broken up into 1 kbyte packets, bringing into play performance effects due to breaking
up messages into multiple packets. These effects are modeled by the software overhead
functions that must be specified as part of every application, as described in Chapter 3.
The functions used for these applications ignore effects due to multiple packets since
messages were in general small enough to fit into one packet.

For RPPT simulations concerned with validation, the measurement and modeling of
communication time suffers from relatively more uncertainty than for computation time.
Ideally, communication time should be broken down into synchronization time, software
overhead time, and transmission time, as defined in Chapter 6. Breaking down the time
spent inside operating system calls, however, requires access to source code so that it can
be profiled by TPROF in the simulation version and be timed by the clock in the real
system version. This access was not available for the Intel node operating system, but may be possible in future work with the V system. If the operating system code is profiled, there is no longer any need to account for it by means of the software overhead functions. Direct profiling would remove much of the guesswork now required to account for these effects. This uncertainty is less of a problem for simulation of hypothetical systems, for which no software overhead measurements can be made. In such cases, the software overhead functions used by the RPPT applications may actually be the most useful and appropriate way to represent these costs.

As can be seen from the range of applications presented here, many programs exhibit a sufficiently high ratio of computation to communication time that message passing modeling accuracy has little effect on total time prediction accuracy. For the matrix theoretic algorithms, the ratio increases for larger matrices, so that accurate message passing modeling becomes more crucial for smaller data sizes and smaller total time durations.

3. Scope of Applicability for Validation Results

Results presented here confirm that loosely coupled systems can be accurately modeled, at least as far as it is possible to accurately measure the quantities of interest on the real system. More specifically, any loosely coupled system with point-to-point (nearest-neighbor) message passing should pose no conceptual modeling problem for the RPPT. Store-and-forward (non-nearest neighbor) message passing will be less successful until the enhancements to Concurrent C discussed above are available. For other types of architectures, the categorizations of what times are of interest and should be measured
will have to be modified. For example, communication time will be of less interest for programs running on a shared-memory system. Time effects related to enforcement of cache coherence policies will be of more interest, and a systematic scheme for correctly measuring them will have to be implemented.

4. Enhancements to the RPPT Tools

The RPPT is likely to continue to be a viable research project for years to come. Several important areas into which the available resources may be profitably channeled over this time frame are discussed below.

- **STATTOOL Integration**

Now that the STATTOOL implementation is largely done, it can provide help with the collection and display of performance statistics of interest in testbed simulations. To integrate it into the testbed, any code in the remainder of the system that needs statistical services must be modified to output raw data and some specification information to STATTOOL in the expected format.

- **Concurrent C Interrupt Servicing**

As discussed above, in order to model some message passing subsystem activity, Concurrent C will need modifications to its process control mechanism to support service interrupts. This will allow modeling of such effects as message forwarding at intermediate nodes, and might also allow more experimentation on the effect of processor scheduling policies on performance.
• **TPROF Enhancements**

More sophisticated profilers and profiler support tools will be needed. Profilers that take into account cache effects have been written. Currently, they restrict data designated as shared from being cached. The extra profile overhead required to simulate cache effects is significant, but may still be less than trace-driven simulations. Ultimately, at the cost of more overhead, this scheme could be expanded to relax the shared data restriction. The characteristic of such programs that determines how expensive the system is to simulate is the granularity of global interaction between processes. If shared data may be cached, all references to shared data become process interaction points, requiring global simulation time to advance at a finer level of granularity. If the total amount of data designated as shared is reasonably small, as is typical, the extra simulation overhead may remain manageable.

Tools for better interpretation of cross profiling results would streamline the process of correcting occasional mismatches generated by the block matching process, as outlined in Chapter 4.

• **OS Code Profiling**

Access to system code in validation work so that it can be profiled would remove some of the guesswork currently involved in simulating the effect of operating system software overheads for validation studies. Unfortunately, software account functions will have to remain for some models, since it is unlikely that the OS source code will be available for every system of interest.
• I/O Subsystem Modeling

Modeling of I/O systems will result in a more realistic assessment of overall program performance. Currently, the RPPT environment does not provide features for modeling the downloading of processes and data from the front end processor to the multiprocessor environment. Predictions are all relative to some point in the program at which processes are active, have their data, and are ready to run. This limitation makes it difficult for the RPPT to provide certain unbiased performance figures, such as speedup, since it must discount or ignore the penalty paid to initialize the program and data. If a system provides poor I/O facilities for getting data to its processors, the simulation should probably reflect this.
References


Appendix: Hypercube Program Listings

This appendix provides the C source code listings for the LU decomposition, eigenvalue-eigenvector solution and FFT programs for which prediction and measurement comparisons were given in Chapter 6. The LU program is adapted from [23], the eigenvalue-eigenvector program is a translation into C of Cleve Moler's original FORTRAN programs [36], and the FFT program is taken from [33].
LU Decomposition
typedef struct MSG1 {
    int n;
    int p;
    int p1;
} MSG1;

def struct MSG2 {
    float val;
    int node;
} MSG2;

def struct TIMEMSG {
    int node;
    long tinsync;
    long bkttotal;
    long bksync;
    long back;
    long becr;
} TIMEMSG;

#define INIT 0x0000
#define FS sizeof(float)
#define FP sizeof(float *)
#define IS sizeof(int)

char *malloc();
long clock();
float urand();

#define NNODES 16
#define INCLUDE <stdio.h>
#define INCLUDE <math.h>

#define max(a,b) ((a>b)?(a):b)
char *strncpy();
char *strlcpy();
char *strlcat();
int strlenmsg;

#include "string.h"
#include "stdio.h"
#include "math.h"

FILE *fp;
double avg1;
double avg2;
double avg3;
double elkcost = .5525; /* cost of call to clock() in more */

/* cube manager main */

main(argc, argv);
int argc;
char *argv[1];

int nseed;
int nproc,mmodis,infad;
int type,crn, as, p1b ;
float **n,*s;
/*
 * read b from stdin
 */

for (i=0;cin[i]++; scanf("%e", &x[i], &n);

/*
 * send out A and b to nodes (message type=1)
 */

for (i=0;cin[i]++; sendmsg((cl, i+(n+1)*FS), nproc, 0);

/*
 * receive back x (1 message of type=3) and
 * receive back timing info (14 messages type=0)
 */

recvbuf = malloc((n*FS, sizeof(TIMEMSG)));

x = (float *) malloc(n*FS);

nexttime = 0;

for (i=0;proc[i][c]++; )

recvmsg’d(&cl, &c, &n, &type, &recvbuf, &nexttime);

if (type==3) {

ptr1 = (char *) x;

ptr1 = (char *) recvbuf;

for (i=0;i<n;i++) *ptr1++ = *ptr2++;

}

else if (type==0) {

ptr1 = (char *) &timemag[nexttime];

ptr2 = (char *) recvbuf;

for (i=0;i<n;i++) *ptr1++ = *ptr2++;

}

}/*
 * clean up cube
 */

dealloc(x-1);

dealloc(cl-1);

close(0);

/*
 * print out results
 */

fp = fopen("output","a");

fprintf(fp,"at=",n);

for (i=0;i<n+1) fprintf(fp,"E",0,i);

close(fp);

/*
 * print out timing
 */

fp = fopen("timings","a");

avg1 = 0;

avg2 = 0;

avg3 = 0;

for (i=0;proc[i][c]++; {

avg1 += (double) timemag[i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c][i][j][c]}
```
TIBM(MINUSG);
char msg[120];
long tf[4];
long ts[4];

/*
 * node main
 */
main() {
    int i, nprow, nrows, ncmds;
    int j, row, col, node, nod, pid;
    int enrow, enot, ent;
    int *maps, **nbr;

    float **row, **rowb, **rowo, **rowh;
    float L[11, 11]; /* a[i][j] */
    float b[11, 11]; /* b[i][j] */
    float *ares, *work;

    MSG1 Info;
    MSG2 xmax;

    cl = copen(0);
    me = mynode(0);
    globalize = me;
    seed = me;
    ncmds = 1;
    row = 0;
    col = 0;
    ent = 0;

    /* receive problem information (message type=0) */
    recv(c[0], Info); /* Info[15][5], &node, &pid); */
    a = Info.a; /* dimension of matrix */
    nprow = Info.p; /* number of processors */
    ncmds = Info.cmd; /* initial number of plot rows */
    nrows = nprow; /* number of rows per processor */

    /*
     * allocate storage for arrays
     */
    maps = (int *) malloc(nprow * FS);
    row = (float **) malloc(nrows * FP);
    rowb = (float **) malloc(nrows * FP);
    rowo = (float **) malloc(nrows * FP);
    rowh = (float **) malloc(nrows * FP);
    b = (float *) malloc((nrows * FP);
    x = (float *) malloc((nrows * FP);
    y = (float *) malloc((nrows * FP);
    v = (int *) malloc(nprow * FS);

    nbr = (int *) malloc(nprow * FS);
    for (i = 0; i < ncmds; i++) { row[i] = (float *) malloc(nrows * FS);
        row[i] = row[i];
    }
    /*
     * define submatrix for this node and set row pointers
     */
    for (i = 0; i < nrows; i++) {
        row[i] = (float *) malloc((nrows + 1) * FS);
        row[i] = row[i];
    }
    /*
     * receive A and b from manager (message type=1)
     */
    for (i = 0; i < nrows; i++) {
        recv(c[1], rowo); */
        for (i = 0; i < nprow; i++) send(c[0], rowo, 1, !, 0);
    if (msg == nprow + 1)
        for (i = 0; i < nprow + 1) send(c[0], Info, 1, 1, 0);
        else recv(c[0], Info, 1, 1, 0);

    } /*
     * wait until all processors have their submatrix
     */
    sync[me] = 0;

    /*
     * LU factorization
     */
    done = 0;

    for (k = 0; k < nrows; k++) {
        /* find local minimum */
        xmax.val = 0.0;
        xmax.node = me;
        for (i = 0; i < ncmds; i++) {
            if (row[i][k])
                if (row[i][k])
                    xmax.val = i;
                    pivot = i;
                }
        }
    /*
     * find global minimum */
    incrast(c[1], 1, &xmax, aproz(MSG2, rowo, nprow,
        &crnt, &node, &pid, !, !,
        &nt, &node, &pid);
        next = xmax.node;
```
If (enr == proc) for (i=0;i<nproc; i++) x[i] = 0;
map[k] = dmap[imatcher];

If (map[k] == next) /* exchange rows if necessary */
  next++;  
  if (me == map[k]) {
      #ifdef SENDSYNC
        async[me] = clock(); examples[me]++;
        endif
      send(cl, row[done][n-1]*FS, next, 0);
      #ifdef RECVSYNC
        if (me == map[k]) examples[me]++;
        endif
        endif
      recv(cl, row[done][n-1]*FS, &cnt, &node, &pid);
      #ifdef RECVSYNC
        tsys[n][me] = clock(); examples[me]++;
        tsys[n][me] += tsys[n][me] - async[me];
        endif
      } else if (me == next) {
      #ifdef SENDSYNC
        async[me] = clock(); examples[me]++;
        endif
      send(cl, row[pivot][n-1]*FS, map[k], 0);
      #ifdef RECVSYNC
        if (me == map[k]) examples[me]++;
        endif
        endif
      recv(cl, row[pivot][n-1]*FS, &cnt, &node, &pid);
      #ifdef RECVSYNC
        tsys[n][me] = clock(); examples[me]++;
        tsys[n][me] += tsys[n][me] - async[me];
        endif
      }

  #ifdef RECVSYNC
  if [0] = row[done][k];
    #* broadcast pivot row */
    broadcast(cl, row[0][n-k]*FS, me, nproc, &cnt, &node, &pid);
  done++;
  
  } else {
    #* receive pivot row */
    broadcast(cl, row[0][n-k]*FS, map[k], nproc, &cnt, &node, &pid);
  
  prs[0] = row[done][k];

    } /* modify submatrix */
  
  int(d, done, rows, nmode, nrow, prs);

    } /* recline */

    #ifdef RECVSYNC
    t[0] = async[me];
    #endif
    #ifdef RECVSYNC
    for (i=0;i<nrows;i++) b[i] = *(row[i]+n);
    #endif
    #ifdef RECVSYNC
    tsys[n][me] = 0;
    #elif examples[me] = 0;
    #endif
    #endif
    t[0] = clock();
    backsub(cl, map, row, b, y, p, proc);
    t[1] = clock();
    #ifdef RECVSYNC
    t[2] = tsys[n][me];
    #elif examples[me] = tsys[n][me];
    #endif
    #* accumulate x's into area (message type 2) */
    areas = (float *) malloc(n*FS);
    work = (float *) malloc(n*FS);
    for (i=0;i<nrows;i++)
    if (map[i] == me) *areas[i] = *areas[i] + *x[i] - 1.0;
    else *areas[i] = 0.0;
    }
    incast(cl, 3, row, p*FS, 0, nproc, &cnt, &node, &pid, work);
if (v[next] == 0) {
    v[next] = nhr[v]+1;  
    while (v[j] == nhr[v]+1) v[j] = 1;  
    return(0);
}
}

v[next] = 1;
return(next);

form_nbr(np,nhr)
int np,nhr;
  int start,cnt,nde;
  start = 0;
  cnt = 1;
  nhr[0] = 0;
  while (cnt<np) {
    nde = nhr[start]+1;
    for (i=0;i<nde;i++)
      if (i==nde nhr[cnt+i] = nde+1;
  }

/* node "root" broadcasts to cube with "np" nodes */

bcast(cltype,vec,root,np,cnt,node,pid)
int cltype,vec,root,np,cnt,node,pid;
char *vecw;
  int cdim,me;
  me = mynode()+root;
  cdim = 1;
  while (cdim<n) {
    if (me==cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      #ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
    }
    else if (me<2*cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
    }
    else if (me>2*cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
    }
  }
  cdim *= 2;
}
}

/* cube with "np" nodes incasts to node "root" */

incast(cltype,vec,root,np,cnt,node,pid,work)
int cltype,vec,root,np,cnt,node,pid,work;
float *vecw,*workw;
  int cdim,me,cltype,vec,root,np,cnt,node,pid;
  me = mynode()+root;
  cdim = np/2;
  while (cdim<n) {
    if (me==cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      #ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      }
    else if (me<2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    else if (me>2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    }
  }
  cdim *= 2;
}

incast(cltype,vec,root,np,cnt,node,pid,work)
int cltype,vec,root,np,cnt,node,pid,work;
float *vecw,*workw;
  int cdim,me,cltype,vec,root,np,cnt,node,pid;
  me = mynode()+root;
  cdim = np/2;
  while (cdim<n) {
    if (me==cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      #ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      }
    else if (me<2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    else if (me>2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    }
  }
  cdim *= 2;
}

incast(cltype,vec,root,np,cnt,node,pid,work)
int cltype,vec,root,np,cnt,node,pid,work;
float *vecw,*workw;
  int cdim,me,cltype,vec,root,np,cnt,node,pid;
  me = mynode()+root;
  cdim = np/2;
  while (cdim<n) {
    if (me==cdim) {
      ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      #ifdef RECVSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        recvv(cltype,vec,cltype,me-cdim)*root,0);
      #ifdef SENDSYNC
        sync[globalmem] = clock(); examples[globalmem]++;  
        endif
        sendv(cltype,vec,vec,me-cdim)*root,0);
      }
    else if (me<2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    else if (me>2*cdim) {
        ifdef RECVSYNC
          sync[globalmem] = clock(); examples[globalmem]++;  
          endif
          recvv(cltype,vec,cltype,me-cdim)*root,0);
      }
    }
  }
  cdim *= 2;
}
Eigenvalue-Eigenvector Solution
# include <stdio.h>
# include <math.h>
# include <errno.h>

#define INS sgo0d(double)
#define IP sgo0d(double *)
#define MTYPE 1024
#define setl(x, n) (n)
#define setll(x, n) (n)
extern double unit();
extern char *malloc();
extern long time();

typedef struct MSGI {
    int n,m,p;
    double epsf,epsa;
} MSGI;

typedef struct nodetimerec {
    long startsync,umynsec;
    int elkm1;
    long cumcomp1,umynsec1;
    int elkm2;
    long cumcomp2,umynsec2;
    int elkm3;
    long cumcomp3,umynsec3;
    int elkm4;
} nodetimerec;

extern int n;
extern nodetimerec atrecl;
int erma;
double frexp();

double mysqrt(arg)
    double arg;
{ double x, temp;
    int exp;
    int i;
    if(x <= 0){
        if(x < 0){
            errno = EDOM;
            return(0);
        } x = frexp(x, &exp);
        while(x < 0.5){
            x = 2*x;
            exp--;
        }
    }*/
    /*
    * this wont work on 1's comp
    */
    if(exp & 1){
        x = 2;
        exp--;
    }
    temp = 0.5*(1.0 + x);
    while(exp > 60){
        temp *= (1<<30);
        exp -= 60;
    } while(exp < 60){
        temp /= 2;
        exp -= 60;
    } if(exp > 0){
        temp *= 1L<<(exp/2);
    } else temp /= 2;
    for(i=0;i<6;i++)
        temp = 0.5*(temp + arg/10);
    return(temp);
}

static double log2 = 0.6931471805599453094;
static double log10 = 2.3025850929944491764;
static double sqrt2 = 0.7071067811865475244;
static double p0 = -3.4028236692863288361;
static double p1 = 0.3057928118377567583;
static double p2 = -9.3670909337684015181;
static double p3 = 0.4210372112179797400;
static double q0 = -1.2060589979600525562;
static double q1 = 0.19469966070488973122;
static double q2 = -0.8911109037943135713;

double mylog(arg)
    double arg;
{ double s, zaq, temp;
    int exp;
    if(arg <= 0){
        errno = ERNOM;
        return(HUGE);
    } s = frexp(arg, &exp);
    while(s < 0.5){
        s = 2*s;
        exp--;
    }
    x = (s-1)<<exp;
    zaq = -s*1;
    temp = (p3*zaq + p2)*zaq + p1)*zaq + p0;
    temp = temp1((q3*zaq + q2)*zaq + q1)*zaq + q0;
    temp = temp1 * exp * log2;
    return(temp);
double log10(xarg) double arg; {
        return(ylog(xarg)/ln10);
    }

/* node "root" broadcasts to cube with "np" nodes */
hcast(cltype,vec,root,np,ncnt,node,pid)
int cltype,vec,root,np,ncnt,node,pid;
char *vec; {
    int cdim,me;
    me = mynode()*root;
    cdim = 2;
    while (cdim<n) {
        if (me==cdim) {
            # ifdef TIME
            nrec[cltype]kcmt++;
            nrec[cltype]async = clock();
            endif
            send(cltype,vec,size,(me+cdim)*root,0);
            # ifdef TIME
            nrec[cltype]cumasync += clock() - nrec[cltype]async;
            nrec[cltype]kcmt++;
            endif
        } else if (me==2*cdim) {
            # ifdef TIME
            nrec[cltype]kcmt++;
            nrec[cltype]async = clock();
            endif
            recv(cltype,vec,size,me-2*cdim)*root,0);
            # ifdef TIME
            nrec[cltype]cumasync += clock() - nrec[cltype]async;
            nrec[cltype]kcmt++;
            endif
        } cdim *= 2;
    }
}
/* cube with "np" nodes incasts to node "root" */
incast(cltype,vec,root,ncnt,node,pid,work)
int cltype,vec,root,ncnt,node,pid;
double *vec,*work;

double *p = *a1;
    *p = (double *)malloc(*a1*BS);
    if (*p == NULL) { fprintf(stderr, "malloc failure\n"); exit(1); }
}

Data2(a,n,n)
double *a2;
int n;
{
    free(a);
}

Data2(a,m,n)
double **a2;
int m,n;
{
    int i;
    for (i=0;i<m;i++) free(a2[i]);
    free(a2);
}

double uniform()
    return((double)rand()/(2147483647.0);
}

#define A 12869
#define C 6925
#define M 23368
#define S 3.0517578125e-S
do double uniform(double seed)
    int *seed;
    long t;
    t = *seed;
    t = (t^C)%M;
    *seed = t;
    return(V0);

/*
 *              cube manager main
 */
do double **a2,**a2,*a2,*a1,*a1,*a1,sp1,**tmp1,**tmp2;
    MSGI buf1;
do double vec1,vec2;
    int type,ext,node,phd;
    int seed;
main(argc,argv)
    int argc;
    char *argv[1];
{
    int i,j,m,n,p;
    int n,ph,extflag;
    int c;
    
    double t;
    double eps1,eps10;
    
    if (argc==4)
    {
        n = atoi(argc[1]);
        p = atoi(argc[2]);
        inflag = atoi(argc[3]);
        eps1 = 1.0e-08;
        eps10 = 1.0e-19;
    }
    else if (argc==6)
    {
        n = atoi(argc[1]);
        p = atoi(argc[2]);
        inflag = atoi(argc[3]);
        eps1 = atoi(argc[4]);
        eps10 = atoi(argc[5]);
    }
    else
    {
        fprintf(stderr, "usage: cmmain n p foln\n");
        fprintf(stderr, "or cmmain n p lo eps1 eps10\n");
        exit(1);
    }

    #ifdef GINPUT
        fprintf(stderr,"cmmain\n");
        fprintf(stderr,"n: %d\n\n",n);
        fprintf(stderr,"p: %d\n\n",p);
        fprintf(stderr,"inflag: %d\n\n",inflag);
        fprintf(stderr,"eps1: %d\n\n",eps1);
        fprintf(stderr,"eps10: %d\n\n",eps10);
        fprintf(stderr,"n\n\n");
    #endif
    
    close(0);
    close(1);
    close(2);
    
    /* load process "nmain" into each node */
    for (i=0;i<p;i++) bload("nmain",i,0);

    /* Send startup message to each node (message type=1) */
    for (i=0;i<p;i++)
    for (j=0;i<m;i++)
    if (i==j)
        buff.m = ex1;
    else
        buff.m = ex2;
    sendmsg(cl1,1,buff,1,MSGI,1,0);
}
/*
* Generate dense real symmetric matrix A
*/
if (inflag==0) {
  /* generate random matrix */
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      if (i==j) a[i][j] = unif(&seed);
      else {
        t = unif(&seed);
        a[i][j] = t;
        a[j][i] = t;
      }
    }
  }
  /* read in matrix from stdin */
  for (i=0;i<n+1;i++) {
    for (j=0;j<n+1;j++) {
      scanf("%f", &a[i][j]);
    }
  }
}
#endif OUTPUT

#define OUTPUT
fprintf(stdout,"A\n\n"); p2e(stdout,a,n,n); flush(stdout);
#endif OUTPUT

/*
* Initialise "x" to the identity matrix
*/
for (i=0;i<n+1;i++) {
  for (j=i;j<n+1;j++) {
    if (i==j) x[i][j] = 1.0;
    else {
      x[i][j] = 0.0;
      x[j][i] = 0.0;
    }
  }
}
#endif OUTPUT

/*
* distribute COLUMNS of A and ROWS of Z to nodes
* in wrap fashion (message type-2);
*/

/*
* receive back d, n, and sigmas (eigenvalues)
* from node 0 (message type-4)
* receive back final eigenvalue (6)
* receive rows of Z back from each node
* (message type-7) and
*/
#else OUTPUT
for (i=0;i<n+1;i++) {
  recvmsg(ch,i,type,evec,n*D3,sizeof(evec),&n);
  switch(type) {
    case 3: evect(c,n,evec); break;
    case 4: evec(n,evec); break;
    case 5: evec(sigmas,n,evec); break;
    case 6: evec(evec2,n,evec); break;
    default:
      if ((type==7) || (type==7+n)) {
        evect(cset(type-7),n,evec);
      } else if ((type==7+MTYPE-E) || (type==7+MTYPE+n)) {
        evect(cset(type-MTYPE-E),n,evec);
      } else {
        fprintf(stderr,"default type out of range: %d",type);
      }
      break;
  }
}
#endif OUTPUT

int main(int argc, char *argv[])
{
  initarg(argc, argv);

  return 0;
}
#include "stdio.h";
#include "net.h";

main() {
    int n,m,p,d;
    int e1,eps,eps1,eps2;
    double *b,*a,*sigma,*work,*wu,*work1;
    double eps3;
    int ex,ey,tx,ty,tx1,ty1;
    cl = copen(1);
    id = mnode(1);
    /* receive startup message (message type=1) */
    n,m,p;
    recv(c1,cl,dbuf1,1024); // receive command message (message type=2)
    n = bufl.n;
    m = buf2.m;
    p = bsp1.p;
    eps1 = bsp1.eps1;
    option = bsp1.option;
    tx = sp1.tx;
    ty = sp1.ty;
    cnode = node;
    cmdid = pid;

    if (id + ex) {
        m1 = id + ex + id;
        m2 = m + ex;
    } else {
        m1 = id + ex + id;
        m2 = m + ex - 1;
    }

    /* allocate storage */
    d = (double *) malloc(n * sizeof(double));
    if ((d == NULL) { exit(1); })
    e = (double *) malloc(e * sizeof(double));
    if ((e == NULL) { exit(1); })
    b = (double *) malloc(b * sizeof(double));
    if ((b == NULL) { exit(1); })
    sigma = (double *) malloc(sigma * sizeof(double));
    if ((sigma == NULL) { exit(1); })
    wu = (double *) malloc(wu * sizeof(double));
    if ((wu == NULL) { exit(1); })

    work = (double *) malloc(work * sizeof(double));
    if ((work == NULL) { exit(1); })
    work1 = (double *) malloc(work1 * sizeof(double));
    if ((work1 == NULL) { exit(1); })
    a = (double **) malloc(a * sizeof(double *));
    if ((a == NULL) { exit(1); })

    for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        s[i][j] = (double *) malloc(s[i][j] * sizeof(double));
    if ((s[i][j] == NULL) { exit(1); })

    /* receive m columns of a and n rows of z (type 2) */
    for (i = 0; i < n; i++)
        recv(c2,cl,dbuf2,1024, &n, &m, &p, &a);

    /* end */
}

/* Program Definitions */
char chbuf[1024];
node_t *netrec[16];

int m,n,p;
int e1,eps,eps1,eps2;
#ifndef MSG1
#define MSG1 bufl
#endif

/* * Driver */
```
for (i=0;i<n;i++) { self(x); n*DS, &cnt, &node, &pid); }

/*
  1. tridiagonalization via Householder transformations
   H = Z*A*V, where
   H is tridiagonal;
   Z = (P(i)’*P(i))’ - P(i), where
   P(i) is an elementary Hermitian matrix that
   effects a Householder’s transformation for
   column i;
   P(i) is unitary, but not symmetric;
   diagonal of H stored in D(i), lower;
   subdiagonal of H stored in E(i), lower;
*/
incast(i,i,work,n*DS,p, &cnt, &node, &pid, work1);
bestat(i,i,work,n*DS,p, &cnt, &node, &pid, work1);

/*
  1. time
*/
int nirec(i, id, cmsgs, 2) = clock(i) - nirec[i][start];
int nirec[i][worker] = nirec[i][worker];

/*
  1. output
*/
if (id == 0) sendw(i, n*DS, cmnode, cmpld);

/*
  1. eigenvector calculation via qr iteration
*/
incast(i,i,cmnode,p, &cnt, &node, &pid, work1);
bestat(i,i,cmnode,p, &cnt, &node, &pid, work1);

/*
  1. eigenvalue estimation via bisection and sturm sequences
*/
for (i=0;i<n;i++) {
  setf(i, d, n*DS, cmnode, cmpld);
}

/*
  1. output
*/
for (id=0; id<n; id++) sendw(id, n*DS, cmnode, cmpld);

/*
  2. return to manager (type=6)
*/
extern onedimrec renedj;

void callee(
for (i=0; i<n; i++) free(size(i));
free(d);
free(e);
free(c);
free(a);
exit(0);
)

extern double mysig();
extern double mylog();
extern char char();

/*
*/

FSYSRE, Parallel SYmetric TriDiagonal REDuction.
FSYSRE reduces a real symmetric matrix to tridiagonal form.

ON ENTRY

double A[N][M]
The symmetric matrix, distributed across P nodes.
Only the diagonal and lower triangle need be supplied.
The jth column is stored in node (j-1) mod p.

Int N
The order of the matrix A.

ON RETURN

A[N][M]
The diagonal and lower triangle contain information
about the transformations that can be used by FSYBAK.
The upper triangle is unchanged.

double D[N]
The diagonal of the symmetric tridiagonal result.

double E[N]
The offdiagonal of the result, E[1] through E[N-1].

double Z[M][N] ***Note transposed dimensions***
N rows of the transforming matrix.
Will become a portion of the eigenvector matrix.

REMARKS

matrix A distributed by columns in wrap-around fashion
How-chooler reduction of A to tridiagonal form
/*
 * Do the transformation
 * only if the column is nonzero.
 */

if (alpha != 0.0) {
    for (t = k+1; t < n; t++)
        setl(t, t) /= alpha;
    setl(k+1, k+1) += 1.0;
}

for (t = k+1; t < n; t++)
    setl(t, k+1) = setl(k+1, t);

bcast(c3, MTYPE=E, &setl[k][1:n-k-1], n-k-1)*DS, r,p,cnt, &node, &pld);
}

/*
 * Rank two update of A, compute only lower half.
 * A = A + u'*u + v'*v = ||A||^2 + ||V||^2
 */

l = m;
for (j = l; j < m+1; j++)
    setl(j, j) += setl(j)*setl(j);

for (k = l; k < m+1; k++)
    setl(k, j) += setl(k)*setl(j);

l += p;

/*
 * Accumulate m rows of transformation matrix.
 * Z = Z*H
 */

for (j = l; j < m+1; j++)
    gamma = 0.0;
for (k = l; k < m+1; k++)
    gamma += setl(k)*setl(k);

for (k = l; k < m+1; k++)
    setl(j, j) += gamma*setl(k);

/*
 * v = v + beta
 * gamma = ||v||^2*||u||^2
 * v = v + gamma*u
 */

if (id == 0) {
    for (t = k+1; t < n; t++)
        setl(t, t) /= beta;

    gamma = 0.0;
    for (k = l; k < m+1; k++)
        gamma += setl(k)*setl(k);

    gamma /= 2.0*beta;
    for (t = l; t < m+1; t++)
        setl(t, t) += gamma*setl(t, t);

    bcast(c3, MTYPE=E, &setl[k][1:n-k-1], n-k-1)*DS, r,p,cnt, &node, &pld);
}

else {
    bcast(c3, MTYPE=E, &setl[k][1:n-k-1], n-k-1)*DS, r,p,cnt, &node, &pld);
}

/*
 * tridiag: tridiagonal bisection
 * finds the eigenvalues of a real symmetric tridiagonal matrix
 * translation of Algol procedure bisect, handbook for automatic
*/
/*

Input
- c[n]: main diagonal of input matrix
- b[n]: off-diagonal
- beta[n]: squares of off-diagonal
- a: dimension of matrix
- m1-m2: indices of eigenvalues to be computed
- eps1: desired precision
- eps2: machine epsilon

Output
- eps2: relative accuracy of results
- x: vector of eigenvalues m1 through m2
*/

testib(c,b,tau,a,m1,m2,eps1,eps2,x,wx)
double *c,*b,*tau,*beta,eps1,eps2,*wx;

int n,m1,m2;

double b,xmin,xmax;
int i;

int a,k;

double q,x1,xu,x0,0;

set(beta,0) = 0.0;
set(0,0) = 0.0;
xmin = set(c,n-1) - fabs(set(b,n-1));
xmax = set(c,n-1) + fabs(set(b,n-1));

for (i=m-2;i>=0;i--)
{
  h = fabs(set(b,i) - fabs(set(0,1)));
  if (set(c,i) + h) xmax = set(c,i) + h;
  if (set(c,i) - h) xmin = set(c,i) - h;
}

if ((xmin + xmax)0) *eps2 = eps2 * xmax;
else *eps2 = eps2 * xmin;
if (eps1 < 0.0) eps1 = *eps2;
*eps2 = 0.5 * eps1 + 7*eps2;

/* inner block */
x0 = xmax;
for (k=m1;k<=m1+k--;)
{
  max = x0;
  for (i=k;i>=m1;i--)
  {
    if (x < set(i+1))
    {
      x = set(i+1);
    }
  }
}

*/
*/

if (x>set(m,k)) x = set(m,k);

x = (x+w)/2.0;
while ((x-w)/x>eps1) (x+w+fabs(fabs(x)+fabs(w))/eps1)
{
  /* Storm sequence */
  a = -1;
  q = 1.0;
  for (i=0;i<n;i++)
  {
    q = set(i+1) - x;
    if (q<0.0) q = set(i+2)/q;
    else q = fabs(set(i+2)/eps1);
  }
}

if (x>eps1)
{
  if (a<11) {
    xu = x;
    set(wu,wx) = x;
  }
  else {
    xu = x;
    set(wu,wx) = xu;
  }
}

if (set(m-1) - set(m))
{
  x0 = x;
  x1 = (x0+w)/2.0;
  set(x0,x1) = (x0+w)/2.0;
}

return (sqrt(a*a + b*b));

*/
*/

PSYQR, Parallel Symmetric Tridiagonal QR.
PSYTRQ computes the eigenvalues and, optionally, the eigenvectors
of a real symmetric tridiagonal matrix.

ON ENTRY

D DOUBLE PRECISION N
The diagonal of a symmetric tridiagonal matrix.

F DOUBLE PRECISION N
The offdiagonal of the matrix, (2,2) through (N-1,N).

SIGMA DOUBLE PRECISION N
Eigenvalue estimates to be used as initial shifts.
Probably obtained from parallel bisection.
**N** INTEGER
The order of the matrix.

**Z** DOUBLE PRECISION (MM,N)
Usually, the output of 
**TSTT3RE**.  
If the tridiagonal matrix is the primary data, 
Z should be MM rows of the identity matrix.

**MM** INTEGER
The number of rows of Z.

**WORK** DOUBLE PRECISION (Z*N)
**WORK** is a scratch array, used only if JOB > 0, 
in which case its dimension must be at least Z*N.

**ON RETURN**

**D** The eigenvalues, in increasing order.

**Z** Some rows of the corresponding eigenvectors, if requested.

**INFO** INTEGER
= 0, usual case, results are accurate.
= nonzero, results may be inaccurate, see below.

**The two special cases, JOB = 2 and INFO nonzero, involve tradeoffs 
between speed and accuracy that apply only to so-called graded 
matrices whose elements vary over several orders of magnitude in 
a uniform way. Graded matrices should be permuted so that their 
large elements are in the upper left corner. JOB = 1 initiates a 
fast version of an algorithm which leads to vectors whose residuals 
are small relative to the norm of the input matrix. JOB = 2 
initiates a slower version of the algorithm which leads to eigenvectors 
whose individual components may be more accurate. The execution times 
of the two versions may differ by as much as 50 percent.

In rare situations associated with underflow, the output value of INFO 
may be a nonzero value, N, indicating that the iteration for the M-th 
eigenvalue did not converge satisfactorily. This is a warning that the 
computed eigenvalues D(I) through D(M) may be inaccurate. Better 
results might be obtained by multiplying the matrix by a large scale 
factor, say 1.0E30, computing the eigenvalues again, and then dividing 
the computed eigenvalues by the scale factor.

**EISPACK** SX: This version dated 01/28/87.
Clive Moler, Intel Scientific Computers

/*

** F ** The element being "chased" by the implicit QR iteration.
** C** cosine and sine of Givens transformation.
** T** Temporary variables in shift calculation and QR step.
** SHIFT** The QR shift.
** TSTT, TST2** Used in convergence test.
** PYTHAG** External function. **PYTHAG** (A,B) = SQRT(A**2 + B**2).
** J** Index, most likely candidate for vectorization.
** JK** Other indices.
** L** Start of submatrix.
** M** End of submatrix and index of eigenvalue being found.

**ITER** Iteration counter.
** MAXIT** Maximum number of iterations.
** PHASE** There are two phases. In phase 1, the implicit 
tridiagonal QR algorithm with a shift from the lower 
2 by 2 is used to compute a single eigenvalue, without 
accumulation of transformations. If JOB = 0, only phase 
1 is used.
In phase 2, the QR algorithm with a shift obtained from 
phase 1 ("perfect" shift) is used with accumulation of 
transformations to obtain all eigenvalues. If JOB = 1, 
only one step of phase 2 is used for each eigenvalue.
** This is fast, and produces accurate eigenvectors for most 
matrices. If JOB = 2, the phase 2 steps are repeated 
until a stringent local convergence test is satisfied.
** This takes more time, but produces more accurate 
eigenvectors for graded matrices.
*/

pytrq(a,d,dma,p,q,mn,work,info,iter,plete)n

int

*double

f(c,e,a,r,1,shift,fmt,fmt2,fmt3,work,info,iter,plete,n)

int

*double

f(c,e,a,r,1,shift,fmt,fmt2,fmt3,work,info,iter,plete,n)

long

for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m++)
for (m=0; m<=1; m+)
/* Implicit QR iteration, chase nonzero V down matrix. */

\[
\begin{align*}
\text{if } & \text{ (phase==2) } \text{ continue; } \\
& \text{ (phase==1) } \\
& \text{ (phase==0) } \\
& \text{ (phase=-1) }
\end{align*}
\]
Radix-2 FFT
# define MASN 1
# define False 0
# define True 1
# define MAN_Type 0
# define BU_Type 1
# define SYM_Type 2
# define STAT_Type 3
# define MAN_NODE 0
# define BU_Type 0
# define BU_Type 16384
# define INDEX 30

#define zmqszed(complex)
#define zmqszed(int)
#define zmqszed(float)
extern int TBSIZ;
extern float COS[2];
extern float SIN[2];

typedef struct complex { float real; /* real part */ float imag; /* complex part */ complex; }

extern long clock();
extern char *malloc();

#define PL 3.141592654

/* table generation */
main(argc, argv)
int argc
int *argv[];
{ int i, n,
n, n2, n3;
double cos[4],
        sin[4];
FILE *fp;
        fopen();
argv++; i = atoi(*argv);
n = i << 1;
n2 = n >> 1;
fp = fopen("table.c", "w");
        j = 0;
        fprintf(fp, "int TBSIZ = \%d;\n", t - 1);
        fprintf(fp, "float COS[\%d] = \{\n", n2);
        fprintf(fp, "\%10.8f", cos[0]);
        for (i = 1; i < n2; i++)
        { temp = i * pi / (double) n2;
        fprintf(fp, "\n");
        [il+1] = 5;
        {fprintf(fp, "in " );
        j = 0;
        fprintf(fp, "\%10.8f", cos(temp));
        fprintf(fp, "in ");
        j = 0;
        fprintf(fp, "\%10.8f", sin(temp));
        for (j = 1; j < n2; j++)
        { temp = j * pi / (double) n2;
        fprintf(fp, "\n");
        [il+1] = 5;
        {fprintf(fp, "in " );
        j = 0;
        fprintf(fp, "\%10.8f", sin(temp));
        fprintf(fp, "in ");
        j = 0;
        fprintf(fp, "\%10.8f", -1 * sin(temp));
        for (j = 1; j < n2; j++)
        { temp = j * pi / (double) n2;
        fprintf(fp, "\n");
        fclose(fp);
        }
        exter FILE *fopen();
        */
        reverse bit reverses num, where num is a number specified
        * in a binary word of length len eg. 6001 is reversed to 1000.
        */
        reverse(num, len)
        unsigned num, len;
        {
            unsigned y, j, k;
            y = num << 1;
            j = 0;
            for (i = 0; i < len; i++)
            { j = j << 1;
            y = y >> 1;
            k = y & MASK;
            j = j * k;
            }
            return j;
        }
        /*
        * cube manager main
        */
        main(argc, argv)
        int argc;
        char *argv[];
        { complex temp;
        int n, /* no. of points in transform */
        int n2, /* 1/2 no. of points in transform */
int
unsigned
int
float
complex
char
int
int
FILE

argc++; 
len = strlen(argc++); 
log_procs = strlen(argc++); 
procs = MASK << log_procs; 
nb = len - log_procs - 1;
n = MASK << len;
numbot = MASK << nb;
n2 = n >> 1;

fprintf(stderr,"point %ld\n");
flush(stderr);

print("Points %d log %d numbott %d proc %d inc", 
    n, len, numbott, proc);
fp = fopen("out", "w");
fprintf(fp, "Point %d log %d numbott %d proc %d inc", 
    n, len, numbott, proc);

long_mlength = 2*numbott*SZlen + (numbott+1)*SZlen;

if (len && (Flush(
    "malloc(2*n*SZlen)\n");
    for (i=0;i<n;i++)
        if (msg_ptr1) 
            fprintf(stderr,"malloc falls on msg_ptr1\n");
        exit(1);
    
    bignum = (float) (1 << 16);
    fprintf(stderr,"point 3\n");
    flush(stderr);
}

msg_ptr1 = malloc(BUF_LEN); /* allocate msg space */

if (msg_ptr1 == NULL) 
    fprintf(stderr,"malloc falls on msg_ptr1\n");
exit(1);

load("fnod", -1, BUF_PID);
#endif PRINT
printf("Manager: Finished activating
");
#endif
cl = copen(MAN_PID);
/*
* Arguments to be passed to each butterfly
*/
*/ log(N) */
/* number of butterflies/stage/process */

*(int*) msg_ptr1 = len;
*(int*) msg_ptr1 + 1 = nb;
/*
* force it to msg format
*/

S = (complex *) (msg_ptr1 + 2*SZinc);
y = (complex *) ((char *) S + numbott*SZlen);
um = (int *) ((char *) y + numbott*SZlen);

for (i=0;i<n;i++)
    index = 2*i*mem;
for (i=0;i<numbott;i++)
    index1 = reverse(index + 2*i*mem);
index2 = reverse(index + 2*i*mem);

fprintf(stderr,"Sent msg with argc %d and nb %d to %d inc", 
    *(int*) msg_ptr1),*(int*) msg_ptr1+1);
#endif
#endif PRINT
printf("Manager: Sent all messages\n");
#endif
endfor (i=0;i<proc;i++)
    recvmsg(C1, type, msg_ptr1, 0, &cnt, &node, &pid);
#endif PRINT
printf("Received sync messages from node\n");
# endif

sending(c, SYNC_TYP, msg_ptr, 0, -1, BUT_PIDI);

# ifdef PRINT
printf("waiting for a message from any node\n");
#endif

for (i=0; i<number; i++)
{
    recvmsg(c, &type, msg_ptr, BUFS_LEN, &cnt, &node, &pid);
    for (j=0; j<number-1; j++)
    {
        ind = (*num)*nombres + j;
        (Maray+ind)->real = (x)*j+real;
        (Maray+ind)->cplx = (x)*j+imag;
        (Maray+ind+1)->real = (y)*j+real;
        (Maray+ind+1)->cplx = (y)*j+imag;
    }
    ifdef PRINT
    printf("bl ind %d to ind %d in %d num: %d OUTPUT\n", 
        ind, ind + n2, k, (*num));
    printf("bl 3rd %d bl 3rd %d to 2nd %d to 4th \n", 
        (x)*j+(y)*j+real, (x)*j+(y)*j+imag, 
        (x)*j+(y)*j+real);
    endif
}

# ifdef PRINT
printf("Received a message from all nodes\n");
#endif

# ifdef DEBUG
/*
 * print out the first few points of the result
 */
printf("an OUTPUT ARRAY\n");
for (i=0; i<1; i++)
    printf("i= %d (real = %f, imag = %f)\n", 
        (Maray+i)->real, (Maray+i)->cplx);
#endif

free(Maray);
free(msg_ptr);
close(c);
waitall(-1, -1);
}

#define top 0
#define bot 1

# define TIME

/* node main */
main()
{
    complex *x,*y,*z,*w, *temp;
    float temp, temp1;
    int l, j, i, p, myid, deeld, *num;
    int tstage, pstate, gstage, cstage;
    int *type, *msgptr, *num, *number;
    int scale, sigind;
    int cl, sw, n3, j, id, pid;
    char *msgptr, *unbuf, *buf;

    # ifdef TIME
    struct start, stop, sync, sync1;
    int clkres;
    # endif

    msgptr = malloc(BUF_LEN);
    buf = malloc(BUF_LEN);
    myid = mynode();

    # ifdef PRINT
    sprintf(buf, "Entered butterfly %d\n", myid);
    syslog(BUT_PIDI, buf);
    # endif

e = open(BUT_PIDI);
    recv(c, MAN_TYP, msgptr, BUFSIZE, &cnt, &node, &pid);
    tstage = *(int *) msgptr;
    nb = *(int *) msgptr + 1;
    pstate = 0;
    number = MASK << nb;

    /*
     * force it to msg format
     */
    x = (complex *) msgptr + 3*SZint;
    y = (complex *) (char *) x + number*SZcom;
    num = (int *) (char *) x + number*SZint;
    smnum = numbuf*SZcom;

    # ifdef PRINT
    sprintf(buf, "Recvd msg from %d tstage %d nb %d\n", 
        node, tstage, nb);
    syslog(BUT_PIDI, buf);
    # endif

    /*
     * Attempt at synchronization using a global send.
     */
    send(c, BUT_TYP, msgptr, 0, MAN_NODE, MAN_FIDI);
    recv(c, SYNC_TYP, msgptr, 0, &nd, &node, &pid);

    # ifdef PRINT
    syslog(BUT_PIDI, "Received sync message from cubemanager\");
    # endif

    # ifdef TIME
    sync = 0;
    start = clkres;
    # endif
}
for (psage = 0; psage < stage; psage++)
{
    pwr = MASK << psage;
    scale = MASK << (TASSIZ - psage);
}

/* find the correct twiddle factor depending on
   * myid and the present stage number
   */
for (i = 0; i < number[psage]; i++)
{
    j = (myid * number) % pwr;
    j = (j + i) % pwr;
    snindx = j * scale;
    w.real = COS[snindx];
    w.complex = SIN[snindx];
    a = x[i];
    b = y[i];
    ifdef PRINT
    sprintf(buf,"butterfly %d: stage %d input:",
            myid * number[psage] + i, psage);
    syslog(BUF_PID, buf);
    sprintf(buf,"x[%d] = x[%d] + x[%d]");
    syslog(BUF_PID, buf);
    sprintf(buf,"y[%d] = y[%d] + y[%d] + y[%d]");
    syslog(BUF_PID, buf);
    endif
    /* Compute butterfly using
       * assigned twiddle factor
       */
    temp1 = b->real * w.real;
    temp2 = b->complex * w.complex;
    temp.real = temp1 + temp2;
    temp1 = b->complex * w.real;
    temp2 = b->real * w.complex;
    temp.complex = temp1 + temp2;
    b->complex = temporary - temp.complex;
    b->real = a->complex - temp.real;
    ifdef PRINT
    sprintf(buf,"butterfly %d: stage %d output:",
            myid * number[psage] + i, psage);
    syslog(BUF_PID, buf);
    sprintf(buf,"x[%d] = x[%d] + x[%d] + x[%d]");
    syslog(BUF_PID, buf);
    sprintf(buf,"y[%d] = y[%d] + y[%d] + y[%d] + y[%d]");
    syslog(BUF_PID, buf);
    endif
    /* if the calculation has not ended find out where
       * the results of this stage are to be sent. Note:
       * One of the results, either the high or low part,
       * goes to the same buffer so only one message need
       * be sent.
       */
    if((MASK << psage) < number[i])
    {
        /* Communication is required only within
           * butterflies in this process. This is done
           * by memory exchanges.
           */
        for (i = 0; i < number[psage]; i++)
        {
            if((i / pwr) % 2 == 0)
            {
                a = x[i] * pwr;
                b = y[i];
                temp.real = a->real;
                a->real = b->real;
                temp.complex = a->complex;
                a->complex = b->complex;
                b->complex = temp.complex;
            }
        }
        else
        {
            notage = psage + 1;
            if((myid * number[psage] / 2) == 0)
            {
                step = 10 * notage + top;
                rstep = 10 * notage + bot;
                desid = myid * pwr / number[i];
                mbuf = (char *) y;
            }
            else
            {
                step = 10 * notage + bot;
                rstep = 10 * notage + top;
                desid = myid * pwr / number[i];
                mbuf = (char *) x;
            }
            ifdef PRINT
            sprintf(buf,"butterfly %d: Sending a message of type %d to %d",
                    myid, step, desid);
            syslog(BUF_PID, buf);
            endif
            ifdef TIME
            syslog(BUF_PID, buf);
            endif
            send(s, step, mbuf, unlim, desid, BUF_PID);
        }
    }
recev(c, rtyp, mbuf, mlen, 
    &ent, &node, &pid);

#
# defined TIME
#
# sync += clock() - sync;
#
}     |
else  |
    |
    /*
    * calculations are complete. Send data back to
    * parallel_fn
    */
    |
    /* location of value in final array */
    |
(*num) = myid;

#
# defined PRINT
#
syslog(BUT_PID, "Finished calculations. Sending results to cm");
#
# defined TIME
#
sync += clock();
#
send(c, BUT_TYP, msgptr, BUF_LEN, 
    MAN_NODE, MAN_PID);

#
# defined TIME
#
sync += clock() - sync;
#
}

#
# defined TIME
#
stop = clock();
printf(buf,"proc %d: (%6d,%6d),mynode(),stop,start,sync);
syslog(BUT_PID,buf);
#
endif

close(c);
exit(0);