

RL-TR-94-111
Final Technical Report
August 1994



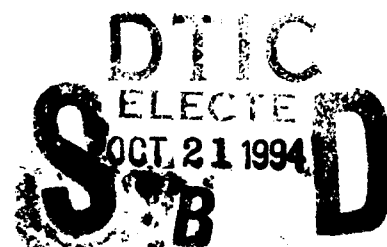
PST: A SIMULATION TOOL FOR PARALLEL SYSTEMS

AD-A285 680



Clarkson University

David J. Potter, William A. Rivet, and Hisham Awad



DTIC QUALITY INSPECTED 2

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

102B

94-32742

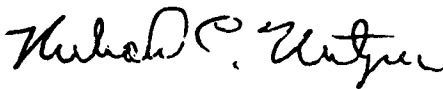


Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

941

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-111 has been reviewed and is approved for publication.

APPROVED: 
RICHARD C. METZGER
Project Engineer

FOR THE COMMANDER: 
JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE August 1994		3. REPORT TYPE AND DATES COVERED Final Feb 93 - May 94
4. TITLE AND SUBTITLE PST: A SIMULATION TOOL FOR PARALLEL SYSTEMS			5. FUNDING NUMBERS C - F30602-93-C-0055 PE - 62702F PR - 5581 TA - 18 WU - PC	
6. AUTHOR(S) David J. Potter, William A. Rivet, and Hisham Awad				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Clarkson University Dept of Electrical & Computer Engineering Potsdam NY 13699-5720			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CB) 525 Brooks Road Griffiss AFB NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-94-111	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Richard C. Metzger/C3CB/(315) 330-7650				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The objective of this research effort was to develop a tool to simulate various parallel computer systems. The tools would give users insight into the different classes of parallel machines in terms of architecture, software, synchronization, communication, efficiency, connectivity, and application specialties. In addition it would give valuable information towards the development of languages and tools which can be used independently of the machine architecture. This work would contribute to objectives set by the Software for High Performance Computers Group at Rome Laboratory, namely: a) developing technology for general purpose parallel computing and b) developing methods for predicting parallel software performance. The tool would be compatible with the Parallel Experimentation and Evaluation Platform (PEEP) at Rome, in particular totally compatible with X Windows workstations.				
14. SUBJECT TERMS Parallel Architectures, Simulation, Program Development			15. NUMBER OF PAGES 106	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

Table of Contents	2
Acknowledgements	4
Introduction	4
Purpose of Project	4
Project Summary	4
Project Overview	5
User Interface.....	5
RAP Language.....	9
Design	9
UIM Module.....	10
Architecture Module	10
Language Module.....	12
Evaluation Module.....	13
Processor Handler	14
Communications Handler	16
Report Organization.....	16
User Interface Module.....	16
Architecture Module.....	18
Language Module.....	19
Evaluation Module	22
Overview	22
The Sequent Model.....	23
Memory Models.....	24
Local Memory Models	25
Shared Memory Models	25
Cache Memory Model	26
The Process Model	27
The Processor Model	27
Variable Allocation.....	32
Sequent Model Details	33
The IPSC, Delta and CM-5 models	33
The CLIP Model.....	34
Clip Extensions	35
Communication Model.....	35
General	35
Description of Hypercube, Delta and CM-5 Communication	
Networks	37
Delta.....	37
Hypercube	40
CM-5.....	42

General Communication Model Outline	44
Hypercube Model	47
CM-5 and Delta Extensions	61
Delta	61
CM-5	65
Performance Summary	68
Future Work	69
References	69
Appendices	70
1. List of Other Documentation	70

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/Availability	
Availability Codes	
Dist	AVAIL and/or Special
A-1	

Acknowledgements

The authors wish to thank the following.

Rome Labs, New York for funding and support of this project. In particular we would like to thank Rick Metzger and Loretta Auil for their contributions and support.

Sun Microsystems Inc. for the donation of a Sun SPARC 10/52 workstation to develop PST.

Jacqueline Wu and Han-Can Wang for writing a great deal of Lex and Yacc code.

William Mauer and Eric Wright for writing a great deal of the X-interface code.

Introduction

Purpose of Project

The objective of this research effort was to develop a tool to simulate various parallel computer systems. The tool would give users insight into the different classes of parallel machine in terms of architecture, software, synchronization, communication, efficiency, connectivity and application specialties. In addition it would give valuable information towards the development of languages and tools which can be used independently of machine architecture. This work would contribute to objectives set by the Software for High Performance Computers Group at Rome Laboratories, namely: a) developing technology for general purpose parallel computing and b) developing methods for predicting parallel software performance. The tool would be compatible with the Parallel Experimentation and Evaluation Platform (PEEP) at Rome, in particular totally compatible with X Windows workstations.

Project Summary

This section describes the accomplishments of the research project in a summarized form.

A parallel simulation tool PST was developed at the Department of Electrical and Computer Engineering Department at Clarkson University for Rome Labs, New York. The tool was written in C using Motif to develop the user interface. The code has been designed to support the following features:

- ease to add new or additional parallel architectures
- ability to simulate many different algorithms because of C-like coding used for all architectures
- ability to monitor many different aspects of computer performance

- ease to add new tutorials describing different aspects of parallel computation.

PST enables the user to select a computer application from a menu and determine how a particular algorithm performance differs when evaluated on different parallel systems. PST allows users to change various parameters of a given machine such as the number of processors and the size of memory to see how different parameters can effect the performance. PST allows users to compare different architectures in terms of performance and efficiency for the same application. Also, PST helps in introducing users to the issues and problems of parallel programming, i.e. synchronization, communication, blocking, deadlock, etc. The user is not limited to the algorithms provided by the tool, she can program any one of the simulated machines with a C like language to investigate the performance of her own algorithms. In addition, the user can add other parallel computer architectures to the simulator with relative ease.

Currently, PST evaluates algorithms on the Connection Machine model CM-5, the Intel iPSC/2, the Intel DELTA, CLIP4, the Sequent Balance Series and a single processor machine. Some tutorials have been written to demonstrate the tools use as a teaching aid.

In terms of accomplishments relative to the tasks proposed in the research proposal, all major issues have been completed. In addition, the graphing tool extends the power of tool first proposed. Due to time constraints, some of the higher level C* constraints for the CM-5 such as *shape* were not completed by the due date of the project. However, it is a minor trivial task intended to be finished in the near future.

Project Overview

User Interface

The X-window User Interface Module (UIM) provides a robust, easy to use multi-window user interface. The UIM allows the user to control the PST operations through mouse and keyboard input and pull down menus. Both run-time and summary statistics resulting from the evaluation are made available to the user through the UIM.

Through the UIM the user can select the machine, change its architecture parameters, select the application, select the evaluation criteria and select what results to graph. Also, through the UIM the user can access a tutorial that gives her a quick introduction to parallel algorithms and architectures by taking her through the process of selecting relevant architectures, applications and parameters.

At the beginning of the program, a menu with three choices: **New Record**, **Tutorial**, and **Quit PST** appears. Each click on New Record brings up a new record window. The record window controls how each individual simulation will run. Tutorial provides on-line help, stored in the form of text files. Quit PST ends all parts of the simulator and exits to the operating system.

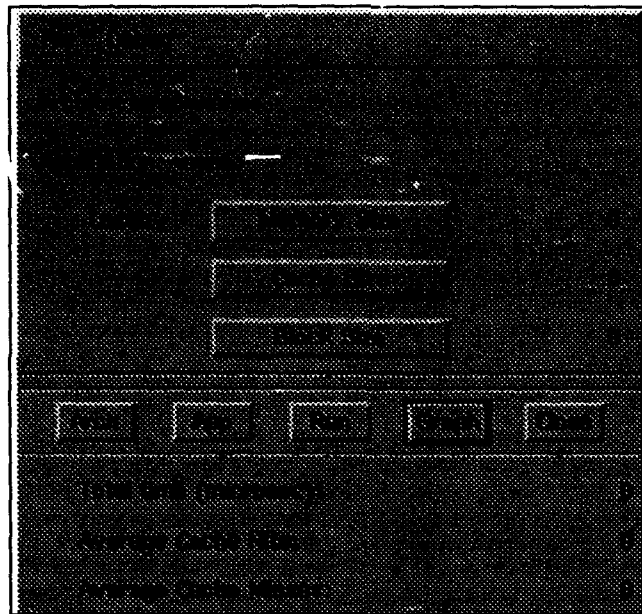


Figure 1

Pressing New Record causes the Architecture Simulator window to pop up (see Figure 1). The top of the window contains the menu bar, with two pull-down menus: **File** and **Debug**. File provides **Architecture**, **Application**, **Run**, **Graph**, and **Close** buttons. Architecture brings up a file selection window, allowing you to choose an architecture to simulate. Architecture files must end with .ACH. Similarly, Application allows you to choose a program to run on the current architecture. Applications must end with .RAP. Run will simulate the selected application on the selected architecture. Run will not work if you haven't selected one (or both) of these, nor if the application and architecture are of incompatible types. Graph calls up the corresponding graph module for this particular record. Close will get rid of the record, and all corresponding windows. The debug pull-down menu allows you to turn debugging on or off. When on, debug information is sent to a resizable scrollable window. Debugging information includes diagnostics such as which processor is executing and which instructions are being executed.

The upper pane of the window contains the architecture file name and the application file name, along with three buttons controlling local memory options, **Memory Size**, **Cache Size**, and **Block Size**. The only limitations on the values are: block size must divide evenly into cache size, and cache size must divide evenly into memory size. When applicable, global memory buttons with the same restrictions appear.

The middle pane of the window contains quick buttons having the same effect as the buttons on the File menu of the menu bar. The last pane provides the results of the simulation: **Total Time** (in micro-seconds), **Average Cache Hits**, and **Average Cache Misses**.

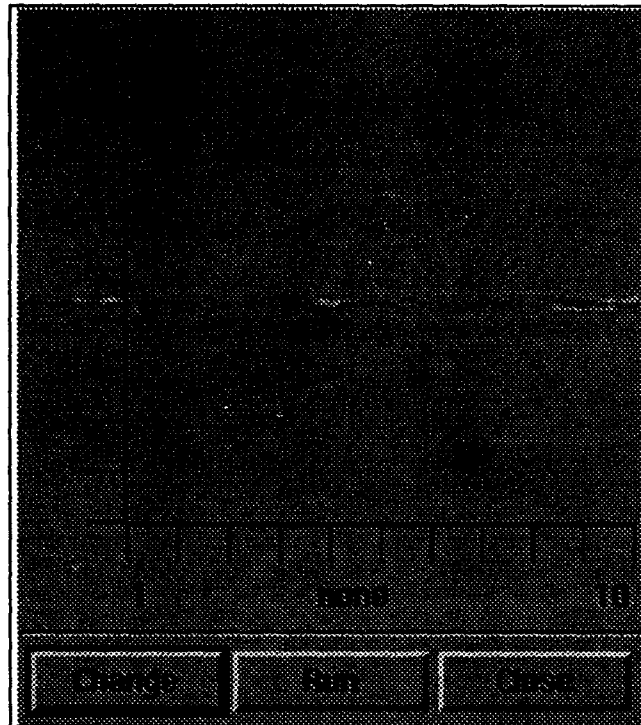


Figure 2

The graph module allows the user to run the simulation multiple times by changing one parameter and comparing the change against another. The upper pane of the window contains the area where the graph appears, along with labels to tell the user what the graph means (see Figure 2). The lower pane contains three buttons: **Change**, **Run**, and **Close**.

The **Change** button pops up a menu of buttons (see Figure 3) used to modify the Graph Module: **X Label**, **Y Label**, **Runs**, **X Start Value**, **Y Start Value**, **X Increment**, **Y Maximum**, and **Close**. **X Label** sets the x axis parameter. This will be the value which will be varied over. **Y Label** sets the y axis parameter, which will be compared against the x axis. **Runs** sets the number of simulations the graph will show. **X Start Value** lets the user change the value to start the simulation's first x value. **Y Start Value** is the lowest value recorded on the y axis of the graph. **X Increment** is the amount the x axis parameter will increment between each test run. **Y Maximum** is the largest value recorded on the y axis of the graph. Excluding **X Label** and **Y Label**, a window with editable text will pop up if one of these buttons is pushed. Clicking in the text window, typing the new value desired and pressing **OK** changes the value in the Graph Module.

When the **X Label** or **Y Label** button is pressed a selection window appears (see Figure 4), from which there are limited choices. Clicking on one of the choices twice, or clicking on it once and pressing the **OK** button sets the axis to the selected field. The **Run** button of the Graph Module runs the simulation, and graphs the results in the upper pane. The results of the Graph Module are not posted on the Record Window. The **Close** button closes the Graph Module window.

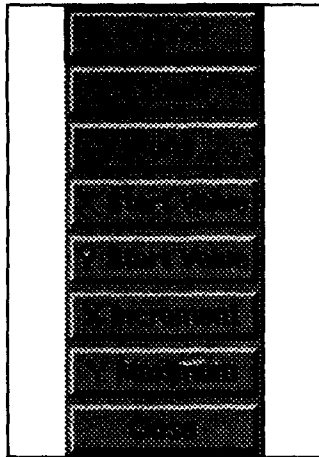


Figure 3

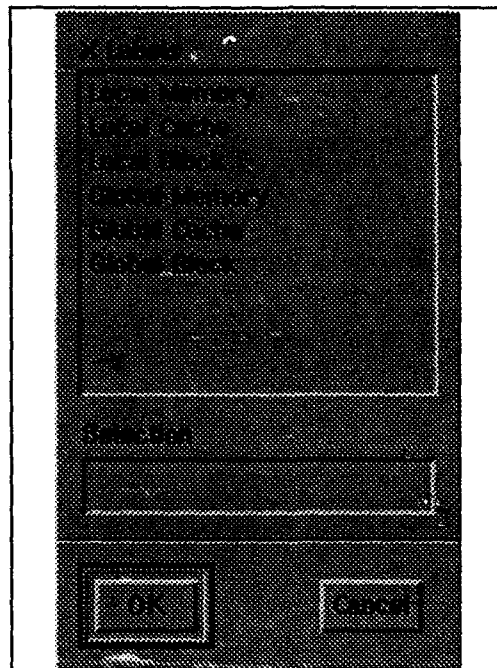


Figure 4

The tutorial button on the main window allows you to have one or more help files present on the desktop while you are running a simulation. After clicking on the tutorial button, a file that corresponds to the topic of interest can be chosen from a selection list. A window with the tutorial text appears; you can resize the window, scroll through the text using the horizontal and vertical scrollbars, or close the window by pressing the close button. The

tutorials explain how to load the particular files and run the simulations for different computing issues.

RAP Language

PST offers its own language: the RAP language. RAP language is a subset of C, but has extra features to support parallel computation. RAP recognizes several modes: CM-5 mode, DELTA mode, Sequent mode, iPSC/2 mode and CLIP mode.

The simulator parses the application program (RAP code) via a Lex and Yacc generated parser which transforms it to the mode that is appropriate for the machine under consideration. For example CM-5 mode for the CM-5 architecture. This is important for an appropriate mapping of the application code to the specific machine. To get better understanding of the execution of the code, the simulator allows the user to trace the code as it is being executed.

In the different modes, most of the operations that are specific to each machine are supported. For example, in the CLIP mode, the CLIP boolean operations (pointwise and local) and in the Sequent mode, `m_fork` Sequent library routine are supported.

Design Overview

As PST is required to evaluate various interactions within a parallel system, several issues needed to be addressed. Since parallel systems have many important activities occurring at the same time, and PST is required to be a sequential, non-parallel system, the general modeling of arbitrary systems by PST is difficult. One way to evaluate system performance is to actually program the parallel system and record the real performance of the system. This is not possible in most situations, and is why PST was developed. Another way to determine system performance is to formulate analytic expressions that can be used to determine system performance. These analytic expressions are difficult, if not impossible, to formulate correctly and does not lend itself to a system that allows easy evaluation of new parallel systems. Finally, a simulation of various components of a parallel system can be used to determine system performance. A parallel computer would be described, a model constructed, and a parallel program would then be run on the system and the performance reported. Although this means a fairly detailed and time consuming evaluations of all the processors, it has the potential of showing bottlenecks as the system is evaluated. This is the method PST uses.

PST was designed to allow easy addition of new system models. In order to accomplish this, care was taken to keep the design of subsystem models modular. For example, local and shared memory are first modeled, then a general cache is modeled to act the same way as "normal" memory, except that it is capable of using another memory device. When a memory device is to be connected to a processor, it is possible to connect local, shared,

cached local, cached shared, or any other combination of memory devices that is desired. New memory models can easily be added following the general rules of memory devices.

The processors were designed with similar issues in mind. For Single Instruction machines, it is not necessary to simulate all processors since they all do the same operations at the same time. For these machines, only one processing element is evaluated. For Multiple Instruction machines, multiple processors are evaluated. Since PST is a sequential system, each processor is evaluated in turn, simulating a parallel system. Any concurrent activity is modeled in this manner. Each concurrent activity will have its own time associated with it, so that the time it takes the complete system to complete a task can be reported.

In order to allow several different experiments to be displayed simultaneously, all the data needed for a given experiment is kept in a separate Record_Table. There is a place for the communications model, processor models and timing information in each Record_Table. PST itself is broken down into four modules. The UIM (User Interface Module), the AM (Architecture Module), the LM (Language Module) and the EM (Evaluation Module). All these modules work together with the data kept in the Record_Table to evaluate the performance of a given hardware/software system.

UIM Module

The User Interface Module (UIM) handles all the interaction between the user and the Evaluation Module. The visual appearance of the UIM has been described above and is also described in the User Manual. It is through the UIM that the user is allowed to specify what architecture and application is to be evaluated. The user is also allowed to evaluate the system after changing some parameters or even changing the entire system.

Architecture Module

The main job of the Architecture Module (AM) is to parse the architecture files (.ACH files) into the Record_Table in a form the Evaluation Module and AM's architecture models can handle. Lex and Yacc were used to generate the AM lexer and parser.

The lexer is responsible for counting lines, filtering out comments, and tokenizing all terminal symbols such as numbers, time units, memory units and architecture parameters. The parser checks statement validity and handles errors. The parser also fills in the architecture parameters in the Record_Table, as illustrated below.

.ACH file

```
/* Sequent ACH file */  
machine = SEQUENT  
num_processors = 4  
int * int = 53 us  
...
```

Record_Table

```
(ignored by AM, line counted)  
Architecture_type=SEQUENT  
dimension.x[0]=4  
(report error in line 4)  
...
```

Before the EM can be run, the machine has to be constructed. There are routines in the AM to construct each machine PST can handle. These routines create the appropriate number of processors and connects the correct caches and memory devices to the processors. "Connecting" devices means setting up function and data pointers that can be used instead of calling routines directly. The result is a modular design that allows easy reuse and addition of different models for various devices. Figure 5 shows pictorially how processors in the Sequent are connected to two caches, each Processor has it's own local memory, but they all connect to the same shared memory.

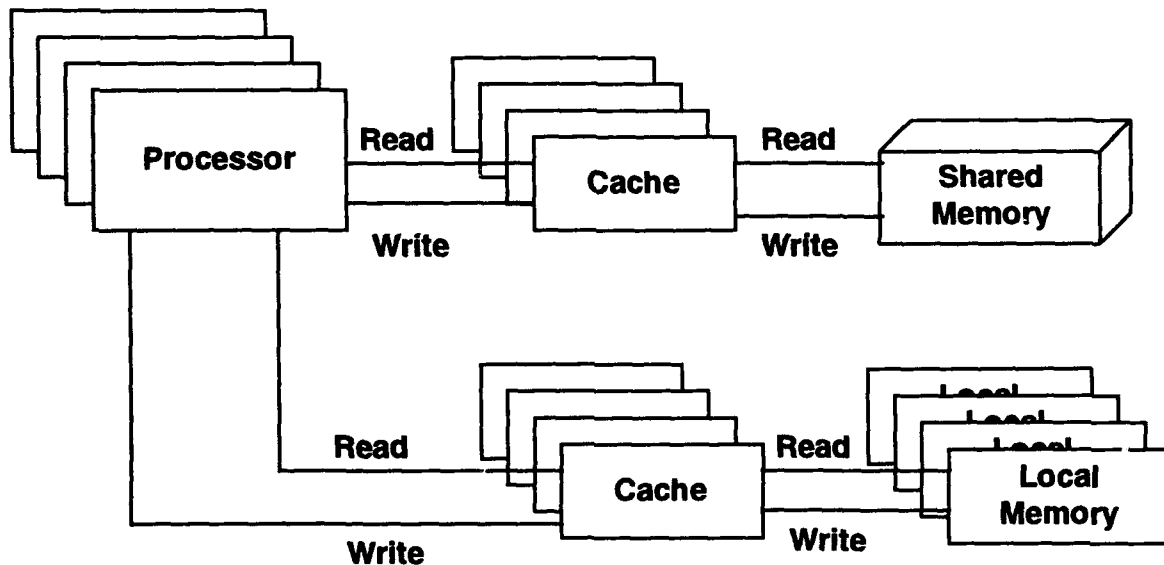


Figure 5.

Once these pointers are initialized the AM's job is essentially finished, since the EM will not have to know what routines to call. The EM need only call the routine pointed to by the device handler with the associated data. In the case of memory, a cache's data can be placed in the Record_Table for some memory range. The EM will call the routines for the cache whenever that memory device is accessed. This is how the second job of the AM is accomplished. All architecture models are utilized in this way. For example, a shared variable on the Sequent would be read from the shared memory range. From the figure, we can see that the request will go through the cache.

Language Module

The Language Module (LM) also has a Lex and Yacc generated parser in it. The lexer, again, simply counts lines, filters out comments and tokenizes all non-terminals. Since the LM has to recognize several modes of languages, all of which are similar to C, the LM parser is much more complicated than the AM parser.

The LM lexer uses several tables to recognize terminals. These tables allow easy addition of new terminal symbols and built-in functions. The LM parses .RAP files into the Record_Table. Each line of code is tokenized and saved in a Record_Table as an array of tokens. The parser also handles making sure that symbols used by the program are defined, and that called functions that are not built in are also defined. The parser also generates tables of variable descriptions for each function it parses. A sample section of code with a partial list of the sequence of tokens that PST uses to represent the program is shown below.

mtest() { int a; a = my_pid(); if(a==1) sendit(); else recvit(); }	proc_begin, reduce semicolon, var_name a, declare_type, declare_local_class, reduce, get_line, var_name a, mypid, assign, reduce, var_name a, const 1, log_equ, if_0, reduce, func_call 3(sendit()),reduce, ...
--	---

Below is a portion of the tables describing the statistics of the function mtest. These statistics are used when PST evaluates a function call, return, and variable allocation.

Function Stats:

name:	mtest
start position:	11
return type:	VOID
num params:	0
total vars:	1
var names:	a
var types:	INTEGER
var dimension:	0
var_data:	(internal structure representing INTEGER variable)

Before the EM can do its job, the LM startup routine must be called, which initializes the processor execution stack with the first line of parsed code. Then the LM is only needed by the EM to obtain and process the tokens.

Evaluation Module

The backbone of PST is the Evaluation Module (EM) and its associated Record_Table, but the EM relies heavily on tables and function/data pairs that are stored in the Record_Table by the Architecture Module (AM) and Language Model (LM). The Record_Table has entries in it for system performance parameters, architecture specific functions (i.e. communications models) as well as parsed RAP code and symbol tables. The entire model of a system is built on these Record_Tables. Since the EM relies only on these Record_Tables, it is a simple matter to allow multiple systems to be constructed and evaluated independently.

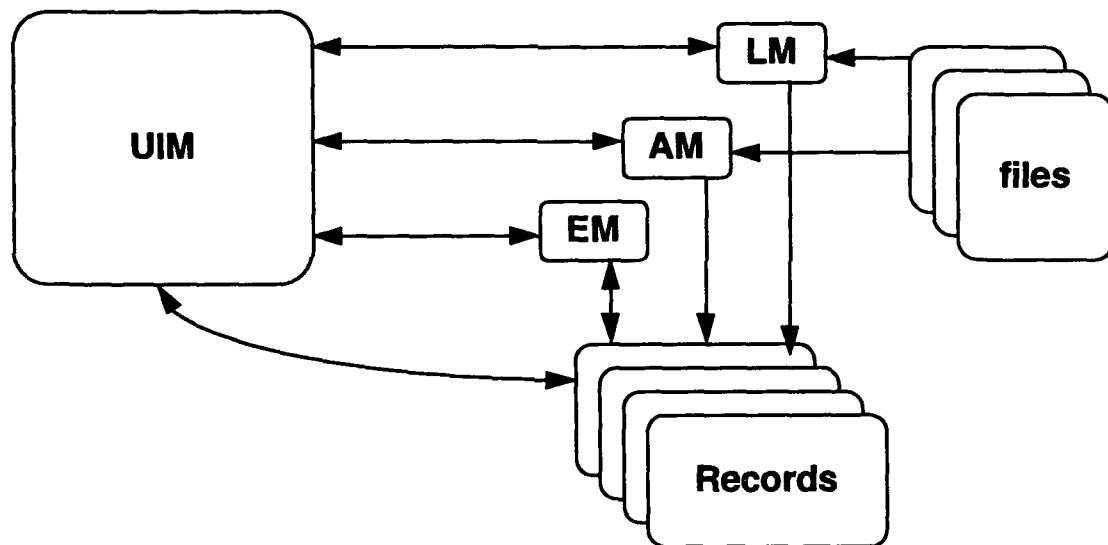


Figure 6.

Notice that all modules have access to the Record_Table. The AM and LM are responsible for part of the table. The EM is responsible for using the data stored in the Record_Table to evaluate the performance of the system.

The Evaluation Module (EM) is by far the most important part of PST. It is responsible for scheduling all the processes in the system. Before the EM starts scheduling process events, all data is flushed, the AM construction routines need to be called to construct the model of the experimental system and the LM startup routine has to be called to setup the first line of code.

PST associates a different time with each process. Generally, these times simply indicate the amount of time a process has consumed. The process with the lowest time is the process that has been waiting the longest amount of simulated time to run. By giving the process with the lowest time attention, processes are given a "first come first serve" priority with respect to simulated time. There are special times that are treated differently in PST. For example, INFINITY is a time that is used to indicate that a process cannot continue without another processes influence. INFINITY is used to block a process that is, for example, waiting for a message from another processor.

The main loop of the EM is the process scheduler which simply passes control to the process with the lowest time associated with it. When the EM has completed the program evaluation, results are tallied, garbage collection is performed and control is handed back to the UIM.

There is a routine for each architecture that builds the model of the machine and otherwise gets the system ready for evaluation. The EM calls on various AM routines to setup memory devices and communications networks. The LM is called on to setup execution of the first line of code on the appropriate processors in the system. The use of these routines come into play later when program evaluation requires the use of these devices.

Once the system has been built, the EM simply goes into a loop that repeatedly passes control to the process with the lowest time. There are three types of processes. The first type, the Processor type, handles evaluating the execution of a program on a processor. The second type is the Communications process, which handles the simulation of a machine's communications network. The final type, the Temp process, is used to make the evaluation of communications routines simpler and will not be discussed further in this overview. The Processor and Communications processes are described below by the handlers used to implement them.

Processor Handler

The processor handler function is called by the EM scheduler. It simply looks at the processor's execution stack to determine what token needs to be evaluated. Each token has a token handler associated with it. There are many token handlers. For example, the source code "a=b+c;" would be parsed into the token string "VAR-a, VAR-b, VAR-c, BIN_ADD, ASSIGN, REDUCE". When tokens are pushed on the processor stack the REDUCE token is used only as a mark indicating that a complete action is ready to be evaluated, so the three VAR-? tokens are pushed first, the BIN_ADD, then the ASSIGN (REDUCE is not pushed on the stack).

(bottom) ... VAR-a, VAR-b, VAR-c, BIN_ADD, ASSIGN ← (top of the stack)

The ASSIGN token handler first pops the ASSIGN token off the stack. Next, the stack is prepared by checking the types of the tokens on top of the stack. If the top token is not a VALUE, the token handler for that token is called. In this case, the BIN_ADD token. The

BIN_ADD token also prepares the stack in the same manner. So the **VAR** token handler is called, which will convert the **VAR-c** token to a **VALUE** token by calling on the memory device routines set up by the AM to evaluate a memory read, and by looking up the value associated with the variable "c". The **VAR** token handler then returns, and each of the previous handlers also return. The stack now looks like this:

(bottom) ... **VAR-a, VAR-b, VALUE, BIN_ADD, ASSIGN** ← (top of the stack)

When the processor handler is called for this processor again, the same sequence of events occur, except that the **BIN_ADD** token handler finds a **VALUE** on top of the stack, so it pops this value (one of the operands) off the stack, and looks for the next **VALUE**, but a **VAR** token is on top, so the **VAR** token handler is called again, to convert the **VAR-b** token to a **VALUE**. The **VALUE** that was popped is now pushed back on the stack and all these handlers return as before, leaving the stack as follows:

(bottom) ... **VAR-a, VALUE, VALUE, BIN_ADD, ASSIGN** ← (top of the stack)

The next time this processor is scheduled, the **BIN_ADD** token will see two **VALUES** on top of the stack, so the **BIN_ADD** can now be reduced by removing the two **VALUES**, adjusting the processor time for the addition, adding the two values, and pushing a **VALUE** on the stack. Now the stack contains:

(bottom) ... **VAR-a, VALUE, ASSIGN** ← (top of the stack)

Now the **ASSIGN** token sees the **VALUE** on top, and the variable name next, so the **ASSIGN** can now be reduced by copying the contents of **VALUE** to the variable "a", and by calling the memory read handler set up by the AM to evaluate the writing of this value to memory. The **ASSIGN** token leaves only the **VALUE** on the stack:

(bottom) ... **VALUE** ← (top of the stack)

The **VALUE** token handler simply removes the **VALUE** from the stack. Now the evaluation of "a=b+c" is complete. Notice that there were two memory reads, one addition and one write evaluated.

Other examples of tokens and token handlers are those associated with the communications routines such as "send", and the "if...else" structure. The communications routines are set up to allow different communications handlers to be used. The **ISEND** token handler, for example, calls the **ISEND** routine setup by the AM. This way, any communications network can be used with the same communications token handlers.

In nesting constructs such as **IF/ELSE**, the structure has a token and token handler for the **IF_n** token, the **ELSE_n** token and the **ENDIF_n** token, where **n** is a number that allows the EM to search the code and stack for matching **IF**, **ELSE**, **ENDIF** and other intermediate tokens. This matching allows arbitrarily nested "if...else" structures to work.

The intermediate tokens are used to allow the EM to pause the evaluation of each part of the "if...else" structure to allow another process to have some processing time. IF1_n and IF2_n are examples. These helper tokens are needed to allow the simulated concurrent evaluation of the conditional statement, and then the execution of the true or false part of the "if...else" statement. Basically, the IF1_n, IF2_n, etc. tokens are used to keep track of which stage the evaluation of the "if...else" structure is at. For example, IF0_n is used when evaluating the conditional, the others are used for evaluating the true part of the conditional, and yet another for the optional false (else) part. A more detailed description of all these tokens and token handlers will be covered later.

Communications Handler

The communications handler handles the communications network by simulating the movement of data on the network. When a processor needs to send a message, it places the appropriate data to the Communications process. The Communications handler takes the data and simulates its movement through the simulated network. When the data arrives at the destination, it also simulates the final delivery of the data to the destination processor.

Report Organization

The remainder of this report describes the PST tool design in detail. The following four sections describe the key modules of the system: the User Interface Module (UIM); the Architecture Module (AM); the Language Module (LM) and the Evaluation Module (EM). Being the largest of the modules by far, the evaluation module section is further broken down into sections describing the various models used such as memory, processor and communications.

Following the description on the design of the tool, a summary of tool performance is given as well as directions for future research and development of the tool.

User Interface Module

The user interface module is divided into several sections: the UIM main loop, record interaction, graph interaction and architecture specific buttons. All of these sections rely on a supplementary data structure (widget) called the NewRecStruct. The UIM main loop sets up the main window, initializes global variables and handles functions that do not fit in with the other main sections. The record section handles all functions pertaining to the creation and modification of a record; the NewRecord() function creates a new record widget, initializes its unique NewRecStruct, and sets up the callbacks for all of its buttons. The graph section handles the functions pertaining to the graph module. Lastly, the buttons section defines the functions that insert buttons specific to a particular architecture into the record widget.

NewRecStruct is a structure that contains all of the widgets and variables inherent to each individual instance of a record window. When NewRecord() is called, it creates a new NewRecStruct. To keep the notation simple, we will refer to the NewRecStruct. When the NewRecStruct is passed to callback functions, there is an entry, w, which holds the parent record widget defined by NewRecord. There is also an entry, temp_widget, which is initially set to NULL. It points to widgets that are to be destroyed at the end of a callback. For example, when you pop up a file selection dialog to choose an architecture, temp_widget gets set to the newly created FileSelection widget; when the user presses OK or Cancel, XtDestroyWidget() is called on temp_widget. There is also an entry, rec, which is a pointer to a Record_Table. This entry gets updated every time the user picks a new architecture or application. After a run, several values stored in rec are used to report system performance information (such as cache hits, cache misses, and total time). The rest of NewRecStruct contains variables that, for example, determine whether or not windows are opened or closed and label widgets that get updated.

Most of the window layouts are straightforward, with the possible exception of the Record window. This window begins with a form-shell, which contains a main-window widget. The main-window widget contains a paned-window widget with four panes. The main-window widget contains a menu bar. The first (top) pane has a form, inside of which are the buttons and labels that pertain to all architectures. The second pane has a form with nothing in it (initially). When a Sequent architecture file is selected, the SequentButtons() function is called, which puts a second form inside of the first, and fills that form with buttons specific to the Sequent architecture. When a different architecture is selected, XtDestroyWidget() is called on the second form, thereby allowing forms to be inserted into the first form later on. The third pane contains a form, in which there are "speed buttons" that merely duplicate the functions found on the menu bar. Lastly, the fourth pane contains a form and several labels that report the results of a simulation run.

When the Graph button is pressed it pops-up the Graph window, and creates a new NewRecStruct with all of the settings of the original. The Graph window consists of a form widget with a paned-window widget of two panes. The top pane contains a form which has three drawing areas: the left and bottom rulers, along with the main graphing area. It also holds all of the labels showing the current settings of the module. The second pane consists of a form with three buttons in it. The Change button calls up the menu by which the settings can be changed, and the Run button calls the GraphRun() function. This function runs the simulation according to the "Number Of Runs" button. It does not affect the Record window in any way because of the new NewRecStruct created at the start of the Graph module. Note that the GraphModule will not run if you do not have a valid architecture and application selected.

The buttons section handles all of the functions pertaining to specific architectures. The function names are prefixed with the name of the architecture, followed by the word Buttons. (Ex.: SequentButtons, IPSCButtons) New architecture buttons can be implemented by duplicating the existing set-up code, and then modifying it to suit the new architecture's needs.

Architecture Module

The AM is primarily responsible for parsing architecture definition files (.ACH files) into memory, and reporting errors as they are found. Lex and Yacc are used to implement the AM parser. The parser is fairly straightforward although the AM parser is modified to avoid conflicts with the LM parser. For each parameter that the user can specify there is a matching terminal symbol defined in the Lex/Yacc code. When the lexer recognizes these terminal symbols they are passed back to the parser, which recognizes collections of symbols according to the rules of its grammar. As the .ACH file is parsed and complete parameter assignments are recognized, the corresponding entries in the Record_Table are filled in. When errors are encountered yyerror is called with a meaningful message. As is standard practice, yyerror is redefined to print errors to the standard error channel, provide line number information and error counting capabilities. The following is an example of the ACH source file and the entries that the parser fills in:

.ACH file:

```
#architecture iPSC
num_processors = 8 /* comments are also
filtered out */

int + int = 250 us
...
local_cache_block_size = .5 kb
...
```

Record_Table:

```
architecture = IPSC
size.x[0]=8 (check that 2^n=8
where n is an integer )(comment
ignored)
intadd=2500
...
local_cache_block_size = 512
...
```

Tables are used by the parser to recognize number-unit pairs as memory or time specification. These tables contain pairs of strings and numbers. For example "byte" is paired with "1" and "kb" is paired with "1000". When the parser parses "kb", it searches the table and sees that "kb" means "*1000" memory units. Similar tables are used to convert other user strings such as machine names to architecture numbers that can be stored in the Record_Table.

time units list symbol table:

```
"ms" -- 10000
"us" -- 1
```

memory unit list symbol table:

```
"bytes" -- 1
"kb" -- 1024
"kbytes" -- 1024
```

When parsing is complete, some parameters are checked for validity while others are computed. The error count is then returned to the user interface so that the user can be prompted to fix the problem and try again.

Conceptually, there is another part of the AM. The collection of routines used to construct and implement hardware models such as processors, shared and local memory, cache, communications networks, and bit memories, can all be considered part of the AM, but their discussion will appear in the EM section.

Language Module

The Language Module (LM) is responsible for parsing RAP source code into a form the Evaluation Module can use. Lex and Yacc are again used to generate the one-pass parser. For memory management simplicity, there are permanent storage tables that temporarily hold the parsed code and tables. The parser requires that the first line of code define the language mode, which gets stored in the Record_Table. Beyond that, the parser is a fairly standard C like parser.

As with the AM parser, symbol tables are used to associate text with symbols, although the LM parser makes more use of them since it is much more complicated. In addition, symbol tables are dynamically built as functions and variables are defined. Errors are reported in exactly the same manner as the AM parser. Errors are counted and line numbers reported with the error messages. Care was taken in the grammar to re-use sections of the grammar wherever possible so that modifications or additions to the grammar would be simplified.

As the parser recognizes code, tokens are added to the Record_Table. These tokens have several values associated with them, the most important being the token type. Tokens will be primarily referred to by their token type. The tokens are generated in a reverse polish-like manner. That is, the arguments appear first, followed by the actions. Here is an examples:

.RAP code:

a=b*c+d;

parsed code (tokens):

VAR-NAME-"a",
VAR-NAME-"b",VAR-NAME-"c",
BIN_MULT,
VAR-NAME-"d", BIN_ADD, ASSIGN,
REDUCE

Notice that reading parsed code must be done in reverse, and that the order of operations is always clear. The parsed code in the above example says that there will be an assignment of the addition of the variable "d" and the result of the multiplication of variables "c" and "b" to the variable "a". The REDUCE token is just a place keeper that means "execute", or "stop and reduce the current line of code". In order to allow arbitrarily mixed and nested "if...else" and "for(...)" constructs, there is a count associated with "for" and "if..else" constructs that counts how deeply nested the code is. When the sequence of tokens associated with "if...else" or "for" are generated, these counts are used to mark the tokens.

.RAP code:	if counter:	parsed code:
if (a==9){	0	IF_TOKEN(0),REDUCE
...code...	1	...code...
if (b==2){	1	IF_TOKEN(1),REDUCE
...code...	2	...code...
}	1	
else{	1	ELSE_TOKEN(1)
...code...	2	...code...
}	1	ENDIF(1)
}	0	ENDIF(0)
...code...	0	...code...

In the above example, a nested "if...else" construct is shown. The count is used and then incremented as each part of the "if" is parsed. The count is decremented after each part is parsed. In this manner, a nested "if" will always use a count that is one greater than the previous level of "if" statements. It may be noticed that the "if...else" and "for" constructs appear in a "forward" direction, that is, they appear in the same order as you might naturally read them. "if" come first, then the "else", and so on. This is due to the fact that it is unknown ahead of time what code will be executed. When the "if" and "for" tokens are evaluated, the appropriate sections in the code are found and executed, using these token types and counters.

As variables and functions are defined, tables are constructed containing their descriptions in addition to recording the token strings that declare them. When variables are declared, their name, type, size and dimension are recorded and stored in a variable table and variable name table as shown below:

.RAP code:	Variable names:	New Var_Data:
int x;	index: name	tracking=FALSE
	0:a	address=unknown until run-time
	1:b	size=4 (bytes)
	2:x (new variable)	dim_list=NULL (because it is not an array)
		dim_count=0 (because it is not an array)
		value=? (not initialized)

In the above example, an integer, x, is declared. The variable name is added to the list of variable names, and a Var_Data table is created and added to the Var_Data list. Later, when the variable x is used, the variable name is indexed in the variable names list, and this

index is stored in the VAR_NAME token to indicate which variable the token refers to. These indices are later used to find the correct Var_Data table. Since variable might be global or local, there are two different places for these tables. Global variables have their own set of tables. Local variable tables are part of function tables. This way it is easy to keep variables that are local only available when used in the function block that declared them.

User defined functions are treated in a similar manner to variables. When functions are declared, the starting location within the parsed code, the number of parameters, return type and list of variable tables that describe the parameters are all stored in a function table.

.RAP code:

```
void fred(int a)
{ int b[10];
...
}
```

Function statistics table:

```
start_pos = 10 (start of fred is 10th
token)
ret_type = VOID
num_params = 1
total_vars = 2
var_names={"a","b"}
var_types={INTEGER,INTEGER}
var_dim = {NULL,{1}}
var_dim_count = {0,1}
var_data = NULL (these are filled in at
run time as variable are
created)
```

Notice that parameter variables are listed first. When "fred" gets called, all the parameter variables are initialized. Variables which are declared inside a function block are initialized as their declaration is evaluated. This way, all the variable tables can be created at once when the function is called and the entries can be filled in as they are declared and used.

When a variable is accessed, the variable tables of the current function are checked followed by the global tables. If the variable doesn't exist in the current context, an error is reported. If a function call is parsed, only the function name and number of parameters are recorded within the token. After parsing is complete, a check is made to verify that all functions were defined and used consistently.

The parser also treats some code differently based on the language. Depending on the language, built-in functions such as communications or image processing functions may or may not be valid. If a given language does not have a communications network, communications functions are not allowed. If a machine has shared memory, variables can be specified as shared or local. These checks are easy to modify if new language modes or features are added.

Since the parsed code is stored in permanent (global) storage and needs to be copied to the desired Record_Table, the exact amount of memory can be allocated for this table. At the same time, a) the code and supporting tables are copied to the desired Record_Table; b) function references are resolved by verifying that the functions are defined and were called with the correct number of parameters and c) the function names are replaced by the function table index (to allow efficient access). The permanent storage and supporting tables can be used again when another file needs to be parsed.

After the parsing is complete, and the parsed code is stored in the Record_Table, the number of detected errors is returned to the user interface and reported to the user. Once the AM parser and LM parser have been called, and there were no errors, the Evaluation Module can be called on to evaluate the system.

Evaluation Module

Overview

The Evaluation Module (EM) is PST's engine. On the highest level, the EM is very simple. Once the architecture model has been constructed, the EM can just call the routines that have been set up, which may, in turn, call other routines. The main job of the EM is to schedule the attention of PST across the various activities that must appear to happen in parallel. There is a time associated with each concurrent process in PST. The scheduler simply calls the handler with the lowest time without regard to what kind of process it is. Since any memory access might be to shared memory, each access must cause the evaluation of a program by a processor to pause in order to give other processors a chance to operate, in case another processor is affecting the memory at the same time. This arrangement also keeps all processors at about the same simulated time.

Before the Scheduler is called, the system model must be constructed. As mentioned earlier, the entire model is built on a Record_Table. First, any models already in the record are destroyed. This is accomplished by destroying all current architecture models and language tables. Then a general architecture construction is initiated by the general_init routines. For example, the Sequent models are constructed by calling the SEQUENT_general_init routine. All these routines are very similar. They all construct the appropriate number of processors, initialize communications, create and connect memory models. After the processor models are constructed, they are initialized with the first line of parsed code.

Once the entire system has been modeled, the EM scheduler continuously passes control to the process with the lowest time until all the processes are done or have indicated that they are waiting forever (Figure 7). Once this condition is met, all processors in the system have completed evaluating their code, so the performance of the system can be recorded. There are machine specific routines to record the performance of the evaluated system, since each machine might have a slightly different organization. When the EM returns control to the user interface, these performance parameters are displayed.

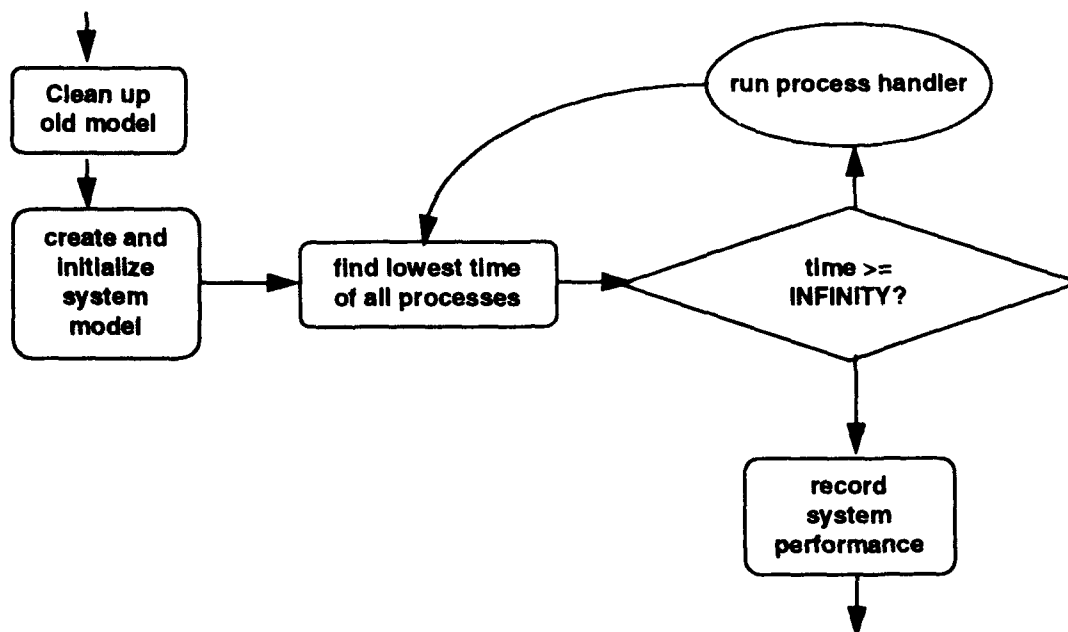


Figure 7.

To describe how the models were designed and implemented, we will describe the Sequent model as a base form. Other models will then be described in terms of additions and modifications to the Sequent model.

The Sequent Model

The Sequent is a shared memory, multiple processor system that generally has between 2 and 30 processors, each running their own copies of a variant of the UNIX operating system. All processors are connected to the same shared memory through 8 KByte caches. Each processor is also connected to its own small local memory. All inter processor communications is done through the shared memory.

The difficulties involved with shared memory are as follows: only one processor can have access to the memory at a time and caching is difficult, since another processor might change the data at a memory location that is in another processor's cache.

There is not much that can be achieved for the first problem. In practice, the effort required to allow more than one processor to access shared memory is considered to be much greater than the advantage such a scheme would provide.

To deal with the second issue, the Sequent implements a First In First Out, Write-Through cache (FIFO WT cache). The cache is a fairly standard one, with one modification to make it compatible with a shared memory system. Each cache ensures that if it contains a block containing a memory location that has been modified by another processor, that block will become invalid. Notice that since the caches are write-through, whenever a processor writes to shared memory, the shared memory gets written to, regardless of whether or not the block was in cache. Since the writing processor can update its cache, it does not need to re-read the entire block. Any other cache that has a block containing the address, however, needs to re-read the block if the processor reads any data from the same block. Notice that the other caches do not need to immediately re-read the block, their copy of the block just becomes invalid, and is thus free for re-use.

To model the Sequent, it is necessary to model shared memory, local memory, and caches. Since this is the first system described, the models used to carry out the evaluation process will be described. The next system that is covered builds on the models developed for the Sequent to a great extent.

Memory Models

To model memory in general, there are a few features that are required for all models. The address range must be described, and access functions must be available. The information to determine access time is only needed within the memory model and is *only needed* when constructing the model. Once a memory model is constructed, simulating a memory access to the device involves calling the memory model's general read or write routine with the current time, address, and size of access. The access routine determines and returns the access time. By making all memory models conform to these rules, memory can be connected to a processor in any arrangement. Shared memory, local memory, and cache devices are all written to conform to these rules which means that any combination of these devices can be modeled by simply creating and connecting the models as desired.

general memory model:

<code>memory_read</code>	evaluate a read of the memory device
<code>memory_write</code>	evaluate a write of the memory device
<code>destroy_memory</code>	used to destroy the memory device

Specific memory models all have the general memory model in common. Specific models only add information to the model. To read and write to these memories, the general memory model is always used. Since the memory read and write access functions will be set to the corresponding model functions, the specific data will be available to those routines. All these memory devices can be connected to the processor model with the `connect_???` routines, which simply add the memory device to the processor model's list of memory devices for the particular architecture ???.

Local Memory Models

The local memory model is the simplest model. Since local memory only has one processor accessing it, the only information it needs, in addition to the general model, is the access time. To simulate a read, the local memory only needs to compute the amount of time it takes to read the amount of data requested. This access time is returned. The same procedure is used for writing. Since only one processor is connected to a local memory, there are no other issues to consider. The time at which the memory is accessed is irrelevant since no other processor or device affects local memory. The memory can always compute the access time easily. The access time is:

$$\text{Access Time} = (\text{size of read}) * (\text{access_time}).$$

To create a local memory, the `create_LM` routine is called. It accepts the start and end address, as well as the memory access time. This routine creates a local memory data table, fills in the access time, addresses, and access functions. When a general memory model read or write is executed, the local memory access functions for read or write will be called. These functions, of course, have access to the extra information in the local memory model.

Shared Memory Models

The shared memory model is only slightly more complex. A shared memory is memory that can be accessed by more than one processor. In this model, shared memory is Exclusive Read/Exclusive Write. That is, only one processor can read or write to shared memory at a time. A processor that tries to access shared memory has to wait if another device is already accessing it. In PST, only one task can be accomplished at a time, so some method of enforcing the ER/EW must be devised.

When a processor model to access shared memory, it is assumed that the processor making the access is the one with the lowest time (this is the way the scheduler functions) and should be the first to win access. The processor is given the access time and evaluation continues. The next access of shared memory cannot happen until this access is complete. So the shared memory model keeps track of the earliest time that it can be accessed, that is the time it again becomes available. The waiting processor's access time will be longer. This second processor also gets an access time, but if the access was requested before the memory was available, it will get a longer access time, otherwise the access time is computed as before. In either case, the next available time is updated.

This protocol ensures that processors get access to shared memory on a first come first serve basis, and that processors have to wait for other accesses before it can complete its own. The only added responsibility of the shared memory model keeping track of the earliest time that it can be accessed. The access time is computed as follows:

$$\text{Access Time} = \text{MAXIMUM}((\text{avail}), (\text{time of access})) + (\text{size of access}) * (\text{access_time}).$$

As with the local memory, there is a routine called `create_SM`, that accept the address range and access time. This routine creates the shared memory model data, and initializes it with the shared memory access functions, memory ranges, access time and an initial available time of zero. When the shared memory access functions are called through the general memory model, these shared memory access functions have access to the extra data needed to implement the above described model.

Cache Memory Model

Caches are memory devices that are used to speed up accesses to some other memory device. Caches also act in the same manner as other memories. When a cache is accessed, the cache checks its tables to see if the data is in the cache, if it is not, it accesses the cached device, also following the general rules of memory access. It updates its tables, and returns the total access time.

The Sequent uses a First In First Out Write-Through (FIFO WT) cache. This type of cache handles reading and writing differently. When a read access occurs, the cache determines whether the cache contains the requested memory (referred to as a "hit") or not (referred to as a "miss"). If it is in memory, the cache calculates the time to access the (local) cache memory. Otherwise, the block containing the requested data must be swapped into cache memory. If there is not enough room in the cache, a block must be swapped out to make room. In a FIFO cache, the first block that was swapped in is the first one that gets swapped out. This means that there are three possibilities on a read: the data is in cache; the data is not in cache, but there is room for a block to be read; the data is not in memory, and there is no room for a new block to be read. The first case is the most desirable, while the third is least.

A write, on the other hand appears simpler. A cache is said to be Write Through when it always writes the data to the cached memory, regardless of whether or not it is in cache. As suggested above, there is a hitch. If a cache is connected to shared memory, it is necessary to know when another processor changes data that is in this processor's cache. This will be referred to as invalidation. The caches in PST have an "invalidate" option which allows them to invalidate the blocks of other caches whenever a write occurs. With this option, caching of shared memory can be modeled.

The additional data required by the FIFO WT cache are the same as for shared memory PLUS a table of blocks and an entry for the cached memory device. Notice that the cached device can be any valid memory device that follows the general rules of the memory model. The function `create_FIFO_WT_cache` handle building the cache. This function accepts the following parameters: start and end address, size of cache, cache block size, cache memory access time, hit and miss penalties (time to decide hit or miss), the mode of operation (does it invalidate other caches?), and the memory device that is being cached.

The Process Model

Now that the memory devices that the Sequent uses have been modeled, it is necessary to model the processors, and to connect the models together. The highest level of the Evaluation Module has already been described briefly above. Any model that needs to appear to operate in parallel will be broken down into units called processes. In the case of the Sequent, the only processes we have are processor processes. Since caches are attached to processors, and memories are attached to caches, these models can be attached to the processor model and do not need to be considered separately. To make each process behave as though it were running in parallel, the process with the lowest time always gets control. It is the responsibility of the process to only do one discrete task at a time, and to return its new time. It is assumed that discretizing the tasks of each process will have the effect of allowing each process to be given control in turn, with the result being each process behaves as though it is continuously running alongside the other processes.

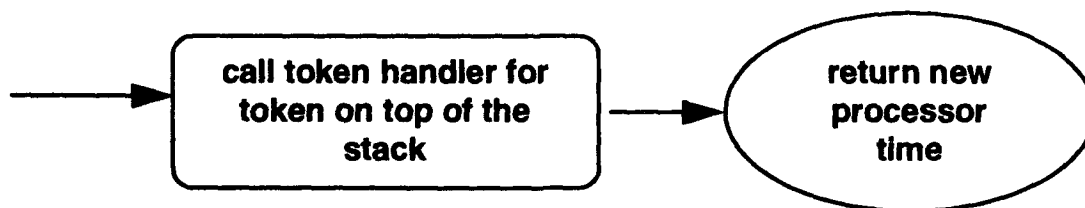
All processes are treated in the same manner, even though they may be doing completely different tasks. All processes are based on a generic process that has a handler and collection of data. Each process has a handler (function)/data pair. The way control is passed to a process is by calling the process handler for that process with the data associated with the process. In general, all processes have a number, process tag, and destroy_process entry. The process tag is used to uniquely identify each process, the number is used to give the process a meaningful number that, in the case of a processor, is the same as that processor's node number.

All these processes return their current notion of their time so the EM scheduler can update its tables. There are special times that PST treats differently. INFINITY means that a process is in the middle of doing something that relies on another process' influence to complete, and that it may be blocked for an infinite amount of time. PROCESS_DONE (greater than INFINITY) means that the process is done and needs no further evaluation. If no process has a finite (less than INFINITY) time, then PST assumes that evaluation is complete since all the processes are either done, or waiting for forever. When this happens, the scheduler calls the machine specific parameter recording routine which does a final tally of system performance by calculating average cache hits/misses and any other parameters that need to be calculated. Control is then returned to the UIM.

The Processor Model

Since all machines have some kind of processor, and PST will not be modeling the differences caused by using a different microprocessor, the processor model will be general. A processor is just the device that is responsible for all the computation in a system. Processors have memory devices attached to them. They also know how to execute parsed code. The processor model has a very simple high level routine. A

processor simply processes the execution stack one more step. When the processing step is complete, the processor simply reports its notion of time.



Since the main task of the processor is to simulate execution of the parsed code, a discussion of how code is treated is appropriate here. Recall that we are still describing a system of evaluating a Sequent system, even though many aspects will be applicable to other models. Since PST will only be evaluating the performance of a parallel system, and should run in a reasonable length of time, only the values of integers are tracked. With this in mind, the language for the moment is limited to handling integer data types, and arrays of integers. It is straightforward to determine the value of an expression. Any value derived exclusively from known values (variable that have had some known value stored in them, or constants) will also have a known value. If a value is derived from one or more unknown values, it, too, will have an unknown value. The process of determining whether or not a value is known will be referred to as tracking. That is to say, values are either tracked (known) or not (unknown). Due to this fact, decisions (if conditions, for loop condition, etc.) cannot be derived from untracked expressions. For example, elements of an array cannot be used to make decisions, although reading, writing, and other mathematical manipulations are completely supported. The reasoning behind this is that it is not necessary to actually, for example, convolve two images to determine the performance of a convolution algorithm. The impact of these assumptions will affect the details some of PST's token handlers, which will be discussed shortly.

Remember, a process is only supposed to do one discrete task, then return control back to the scheduler. To accomplish this a processor utilizes an execution stack, program counter, memory models, call stack, function tables, variable table and its own notion of time (the processor time). To process user code, PST implements a fairly standard stack-based algorithm. The only difficulty encountered here is the restriction of accomplishing only one task at a time. In an ordinary stack based algorithm, parsed code is executed by calling token handlers, that, if necessary, call other token handlers. Each token handler does its job and returns. The problem is that in PST, each token handler might not be able to complete its task if it or any token handlers that descend from it cannot complete their task. In fact, very few tokens can complete their task in one call, since they are required to return control as soon as they have done one discrete task. It is observed that the only modification that is needed is to make each token handler re-entrant. Another way of looking at it is to say that each token handler must save its state on the stack before it returns, so that the next time the stack is processed, the correct token handlers will be called, and they can continue their job.

To help explain the design and use of token handlers, an example will be presented. When a processor needs to execute the line "a=b*c+d", it simply reads each token, VAR_NAME-"a", VAR_NAME-"b", VAR_NAME-"c", BIN_MULT, VAR_NAME-"d", BIN_ADD, ASSIGN, and REDUCE, pushing each one on the execution stack in turn, so that REDUCE is on top. The routine that does this is "Get_line_of_code", which simply reads tokens from the parsed code, and pushed them on the execution stack until a REDUCE is encountered. The REDUCE is never left on the stack.

Parsed code:

...VAR_NAME-"a", VAR_NAME-"b", VAR_NAME-"c", BIN_MULT,
VAR_NAME-"d", BIN_MULT, ASSIGN, REDUCE....

Execution stack :

before:	after:
	ASSIGN
	BIN_ADD
	VAR_NAME-
	"d"
	BIN_MULT
	VAR_NAME-
	"c"
	VAR_NAME-
	"b"
	VAR_NAME-
	"a"
...(other tokens)	...(other tokens)

There is actually a token handler that is responsible for calling the Get_line_of_code routine. It is the GET_LINE token, and its sole job is to bury itself under other tokens. Each time the execution stack is reduced down to the GET_LINE token, it reads more code on top of itself. We will assume that the current state of the stack is as shown above. When this processor is called by the EM scheduler, the ASSIGN token handler is called.

The ASSIGN handler first removes the ASSIGN token from the stack. It then looks for two arguments: a VALUE and a VAR_NAME. Each is taken from the stack if found. If ASSIGN finds both, the assignment is made (VALUE is copied to the variable names by VAR_NAME), and the VALUE is returned to the stack. The reduction looks like the following:

Stack before reduction:

ASSIGN
VALUE
VAR_NAME
...(other tokens)...

Stack after reduction:

VALUE
...(other tokens)...

In the "a=b*c+d" example we are considering, ASSIGN does not see VALUE on top of the stack, the ASSIGN cannot be reduced. Instead, ASSIGN simply calls the token handler for the token on top of the stack, the BIN_ADD token handler in this case. When this token handler returns, the ASSIGN token will replace the top token, and itself return. The next time the ASSIGN token is called, the same sequence of operations will be followed based on the new contents of the stack.

The BIN_ADD token handler falls into the category of a binary operation. Since all binary operations are similar, they all use the same token handler. This token handler treats all binary operations in an identical manner with the exception of what it actually computes, which is determined by the type of the token. The binary token handler removes the BIN_xxx token, then looks for two VALUES. If they are found, they are removed, the given binary operation is carried out, and the resulting VALUE is pushed on the stack, yielding the following stack reduction:

Stack before reduction:

BIN_ADD
VALUE
VALUE
...(other tokens)...

Stack after reduction:

VALUE (sum of two VALUES)
...(other tokens)...

Recall that some VALUES may not be tracked, and the tracking rules must be enforced in this token handler. In our example, the VALUE token is not found, so the BIN_ADD cannot be reduced. The stack is again processed by calling the token handler for the token that is currently on top. In this case, the VAR_NAME token handler.

The VAR_NAME token handler is very simple. It simply looks up a variable's contents, address, and size information. The access time determined by the memory models is added to the processor time and a VALUE is created and pushed on the stack to replace the VAR_NAME token. Thus the reduction of a VAR_NAME token is simply the replacement of the VAR_NAME with a VALUE. The VAR_NAME handler now returns.

In our example, the VAR_NAME handler returns to the BIN_ADD handler. The BIN_ADD token handler replaces the BIN_ADD token to the stack before it returns. Now the ASSIGN replaces its ASSIGN token and returns to the EM scheduler. The net result is as follows:

Before processing:

ASSIGN
 BIN_ADD
 VAR_NAME-"d"

 BIN_MULT
 VAR_NAME-"c"
 VAR_NAME-"b"
 VAR_NAME-"a"
 ...(other tokens)

After processing:

ASSIGN
 BIN_ADD
 VALUE - (the value of variable
 "d")
 BIN_MULT
 VAR_NAME-"c"
 VAR_NAME-"b"
 VAR_NAME-"a"
 ...(other tokens)

The next time through the process, the BIN_ADD token will get the first VALUE, but will not get the second. Instead, the BIN_MULT handler will be called, which will look for a VALUE. It will instead see the VAR_NAME-"c" token. The VAR_NAME token handler will be called, and another memory read will occur. Each token handler will replace its tokens on the stack, referred to as "saving its state", and return. The next time, the variable "b" will be read, and each token will again save its state. The next time, the BIN_MULT will be reduced, and so on until finally the ASSIGN can be reduced. The following table shows the state of the stack after each step. The asterisks indicate tokens that will be reduced.

Step 1:	Step 2:	Step 3:	Step 4:	Step 5:	Step 6:
ASSIGN	ASSIGN	ASSIGN			
BIN_ADD	BIN_ADD	BIN_ADD			
VALUE	VALUE	VALUE	ASSIGN		
BIN_MULT	BIN_MULT	BIN_MULT *	BIN_ADD *		
VAR_NAME-	VALUE	VALUE *	VALUE *	ASSIGN	
"c"*					
VAR_NAME-	VAR_NAME-	VALUE *	VALUE *	VALUE	
"b"	"b"*				
VAR_NAME-	VAR_NAME-	VAR_NAME-	VAR_NAME-	VAR_NAME-	VALUE
"a"	"a"	"a"	"a"	"a"	
...(other	...(other	...(other	...(other	...(other	...(other
tokens)	tokens)	tokens)	tokens)	tokens)	tokens)

All other token handlers are implemented in a similar manner. When a user defined function is called, the FUNC_CALL token handler is called. It behaves in a similar manner as the binary operation token handler, except that it looks for as many VALUES as it has parameters. When it has all the parameters it needs, it looks up the location of the called function. The current function is suspended by placing all the local variable tables and the current program counter on the call stack. Then the current program counter is changed to the new one. Get_line_of_code is called to start the new function, and a new set of local

variable tables is constructed. A series of assignments between parameters and VALUE is also pushed on top of the stack to initialize the parameters before the first line of parsed code for the function gets processed.

The details of other token handlers will not be covered here in as much detail. Techniques are used to allow the stack to remain fairly small while still allowing nested "if...else" and "for(...)" constructs. The solution lies in using the counts supplied by the LM with these tokens and matching intermediate tokens to keep track of each level. For example, there are a total of ten tokens used to implement a "for(...)" construct. There is the FOR_TOKEN, which initiates the loop, the END_FOR, which marks the end of a for loop in the code, as well as several tokens that handle evaluating the initialization, conditional, body, and increment statements.

While the details of other token handlers will not be discussed, the methods used by the Evaluation Module to allocate variable storage will be discussed. Since PST does not allow the dynamic allocation of memory, allocation of function variables within a function block can be accomplished in a stack-like manner.

Variable Allocation

When functions are called, new variables must be created. This means that there must be some kind of memory management model that keeps track of memory usage, and most importantly, variable addresses. The Sequent has shared class variables which exist in global memory, and local class variables, which exist in local memory. For this reason, processors need to model the memory usage of these memories.

Since PST does not allow the dynamic allocation of memory, variables in local memory are declared and freed in opposite order. It is only necessary to keep track of the highest memory location that is available for use. As variables are allocated, they are given this address, and this top of memory (referred to as top_mem) is incremented. When the variable is destroyed, the top_mem is replaced by the address of the variable. For example, when one function calls another, the parameters and variable of the new function are allocated. When that function returns, all its variable storage is given up, and the memory usage returns to the same state as before the call. Since the memory usage of local memory are not affected by other processors, we are guaranteed that this property will hold regardless of the program being evaluated.

Allocation of variable storage in shared memory is more difficult, since each processor may allocate and free memory at different times. Each processor will allocate and free memory in a similar fashion as described above for local memory. Two processors may not happen to allocate and free memory in the same order, especially if one takes more or less time to execute. This complicates shared memory allocation substantially. As with the local memory, PST keeps track of the highest available memory location in shared memory, but a technique is used to help determine the new top_mem (top of memory). There is a table of memory addresses that mark the end of each variable allocation. A new

entry is created for each new allocation. When a variable is freed, this table is scanned to determine the new `top_mem`. While there are other schemes, this one provides a simple, easy to compute, memory management system.

Sequent Model Details

So far we have described how the memories that the Sequent uses are modeled. The processor model, complete with the methods used to evaluate user code and memory management have also been developed. The only thing that is left is to explain how all these models are brought together.

The data that the AM parser set up in the `Record_Table` is used to construct the system model. The Sequent general initialization routine creates a *shared memory* and as many processors as specified in the `Record_Table`. For each processor in the system, a local memory is created, and two caches are created. One cache is created for the local memory, the other for the shared memory. The memory usage tables are initialized. Finally, one processor is initialized with the first line of code and is added to the EM scheduler.

Now when the scheduler is called, the system will perform the evaluation of the Sequent. The only issues left to discuss are the system calls. There are several Sequent specific functions that are supported. The `mfork()` function initializes a specified number of processors with the first line of code of a specified function. Two other functions are needed to complete the Sequent system calls: `mypid()`, which returns the node number of the current processor, and `numnodes()`, which returns the number of processors in the system.

The `mfork()` routine is the more difficult of the three to implement. It is assumed that the processor that is initially running is processor number 0. When `mfork` is called, for example, "`mfork(4,fred)`", 3 more processors are added to the EM scheduler at the current time of processor 0, and they are initialized with the first line of the function `fred`'s code. When the `mforked` processors finish, they are removed from the EM scheduler. Processor 0 always waits for all `mforked` processors to return before continuing with the next line of code.

The function `myid()` and `numnodes()` are trivial. They simply return the number of the current processor, and the number of nodes in the current system respectively. Since these data are in the processor table and `Record_Table`, they are simply copied to a `VALUE` token, which is left on the stack for the evaluated code to use.

The IPSC, Delta and CM-5 models

The iPSC, CM-5, and Delta are all multiple processor machine connected in different pattern. Each processor in these systems has only local memory, and communications

between processors is through the communications network. When these machines execute programs, all processors are started at the same time. To model the iPSC, the general initialization routine needs to construct many of the same models as the Sequent.

Before the initialization routine can start, the AM parser must be expanded to recognize the new communications parameters, such as link latency and bandwidth. Notice that these new parameters must be valid in the correct types of .ACH files and not valid in Sequent. The LM parser must also be expanded so that in iPSC mode recognizes the new communications system calls such as `csend()` and `crecv()`, while not allowing the Sequent system calls.

The data that the AM parser sets up in the `Record_Table` is used to construct the system model. The general initialization routine creates as many processors as specified in the `Record_Table`. For each processor in the system, a local memory is created, and a cache is created for the local memory. The memory usage tables are initialized. Finally, all processors are initialized with the first line of code and is added to the EM scheduler. Finally, the communications network is initialized.

Now when the scheduler is called, the system will perform the evaluation of the given machine. As with the Sequent, there are special functions that are valid in these new machines. They are all communications related and are covered in the Communications section of this report.

The CLIP Model

The CLIP is a single instruction, multiple data machine, which means that there is only one operation happening at a time, but it is happening in all the processing elements. The CLIP is connected to a host machine that gives it its instructions.

The CLIP is a 96x96 array of bit processors which are generally intended for image processing. Each processor has only a few bits of memory, usually 32 bits. The processors can operate on these bits, update the bits, pass one of its bits to a neighbor and accept one bit from its neighbor. There are special functions that load an entire bitplane into the CLIP memory, or swaps out an entire bitplane. There are also functions to latch the bits that are to be processed, as well as functions that cause a bit operation to be carried out. Since each processor only processes a bit at a time, multiplication of images with 8 bits by images with 8 bits requires the processing of each bit separately. While this makes the code long, there is an advantage in that all 96x96 processors will be running at the same time, and will complete the operation at the same time. Software on the host computer implements a FIFO cache scheme with the bit memories.

To model the CLIP, it is only necessary to evaluate one processor since each processor will be doing exactly the same thing. Since the CLIP is a completely different kind of machine, which primarily processes images, PST must be expanded to handle images as well as bit memories.

Clip Extensions

The CLIP memory model also follows the general memory model. The extra data the CLIP memory has is as follows: length (number of bits per processor), processor register A and B and bitplane information. The CLIP memory is viewed as occupying one memory location. All reads and write to CLIP memory are empty functions, since images are not directly accessed. The CLIP memory model is so simple because all the action happens based on the user code.

As before, the AM parser must be expanded to recognize the CLIP parameters, such as number of bitplanes, bitplane access times and primitive operation times. The LM parser must be modified to recognize image types, and the CLIP system functions such as LDA, LDB, PST_local, PST_pointwise, and image locking functions. Once the parsers have been updated to include the CLIP constructs, the token handlers corresponding to the new system calls in the Evaluation Module must be added.

Both the LDA(), and LDB() system calls simply latch the specified bitplane into the CLIP bit memory. If the bitplane is not already in memory, it is loaded. The PST_local() and PST_pointwise() first ensure that the destination bitplane is in CLIP memory, if it is not, it designates a bitplane to be the destination, and the old one is discarded. If there was no room for the destination image, a bitplane is swapped out to make room. While the evaluation of PST_local() and PST_pointwise() are very similar, they may be specified to have different execution times since they actually carry out different operations.

Since the bitplanes are cached, the bitplane load and store operations follow the rules of a FIFO cache, with one modification. Bitplanes can be locked. A locked bitplane will only be swapped out if there are no other planes that can be swapped. That is, if an image must be swapped in, unlocked images will be swapped out before locked images. The system calls im_lock() and im_unlock() to set a lock flag in the specified images.

The CLIP general initialization routine simply creates one processor and creates local CLIP memory. Both memories are connected and the processor is initialized with the first line of code. Now the EM will evaluate the user code, and all the CLIP system calls will be evaluated.

Communication Model

General

The communication model simulates the message passing operations performed on the distributed-memory parallel MIMD systems considered by the PST; currently the Intel Hypercube, Intel Delta and Thinking Machines' CM-5. Systems of this kind are made up of a set of nodes, each of which consists of a main processor, memory, and interface to the

network. Nodes process information independent of one another and communicate by sending and receiving messages. This independence gives these systems what is called a loosely-coupled architecture.

Even though each processor has its own goal, all must work together to produce the overall objective of the parallel application. Therefore, communications among processors in a loosely-coupled system is essential. In fact communication on such systems serves both to synchronize processes and to exchange code and data information among processes.

The importance of communication on such systems rises from its cost and the problems it may cause. The cost of sending messages between processes is the time required to send, transfer and receive them; time that could be spent doing computations that advance the solution of the problem. In general, minimizing the ratio of the number of messages to the number of calculations performed on each node will improve the running time of the algorithm.

Bottlenecks and deadlocks are serious problems that may result from careless message passing algorithms. A bottleneck results when all the nodes in the system send a message to a single node at the same time. Deadlock may result when all nodes are waiting for a message that never get sent.

As explained above, each processor has its local memory. When information that is stored on one node is required by another node, one node must send the information and the other node must receive it. This is the concept of message passing that is used by the above three machines. Sending a message requires cooperation from the sending node, the receiving node and the communication network. On the above three machines, when the communication network accepts a message it takes the full responsibility of delivering it to its destination. Because of its independent operation, PST views the communication network as another process running on the system that needs to be evaluated.

As explained above, the Evaluation Module (EM) evaluates one line of code from the application code at a time. A line of code can specify a simple operation or an architecture specific operation. A simple operation is an operation that requires only one processor local resources for its execution. An architecture specific operation is an operation that involves more than one processor.

The EM provides routines that are called to evaluate architecture specific operations. Message passing operations, being architecture specific operations, are evaluated by calling the appropriate handler. For example, to evaluate a `csend()` function call on the DELTA, the DELTA `csend` handler is called.

PST divides the messages passing operation into two functions: The processor function and the communication network function. The processor function also has two functions: the sender function and the receiver function. Each of these functions gets evaluated be

calling the appropriate message passing handler. When a processor issues a `crcv()` function call, `???_Crcv()` - where `???` is the machine name - is called to evaluate it. Similarly, when a processor issues a `csend()` function call, `???_Csend()` is called to evaluate it. Evaluating the communication network function of the message passing is done by calling the `Handle_???_Communications` routine. This routine gets called by the EM when there is an active message on the communication network.

Description of Hypercube, Delta and CM-5 Communication Networks

Delta

The Touchstone DELTA system has 576 nodes, where 512 are computational nodes, connected in a 2-dimensional 16x36 mesh. Each processor is connected to a VLSI Mesh Router Chip (MRC) which connects it to its nearest neighbors: this means four connections per MRC, except for the top, bottom, and side MRC, which have three (see Fig. 8).

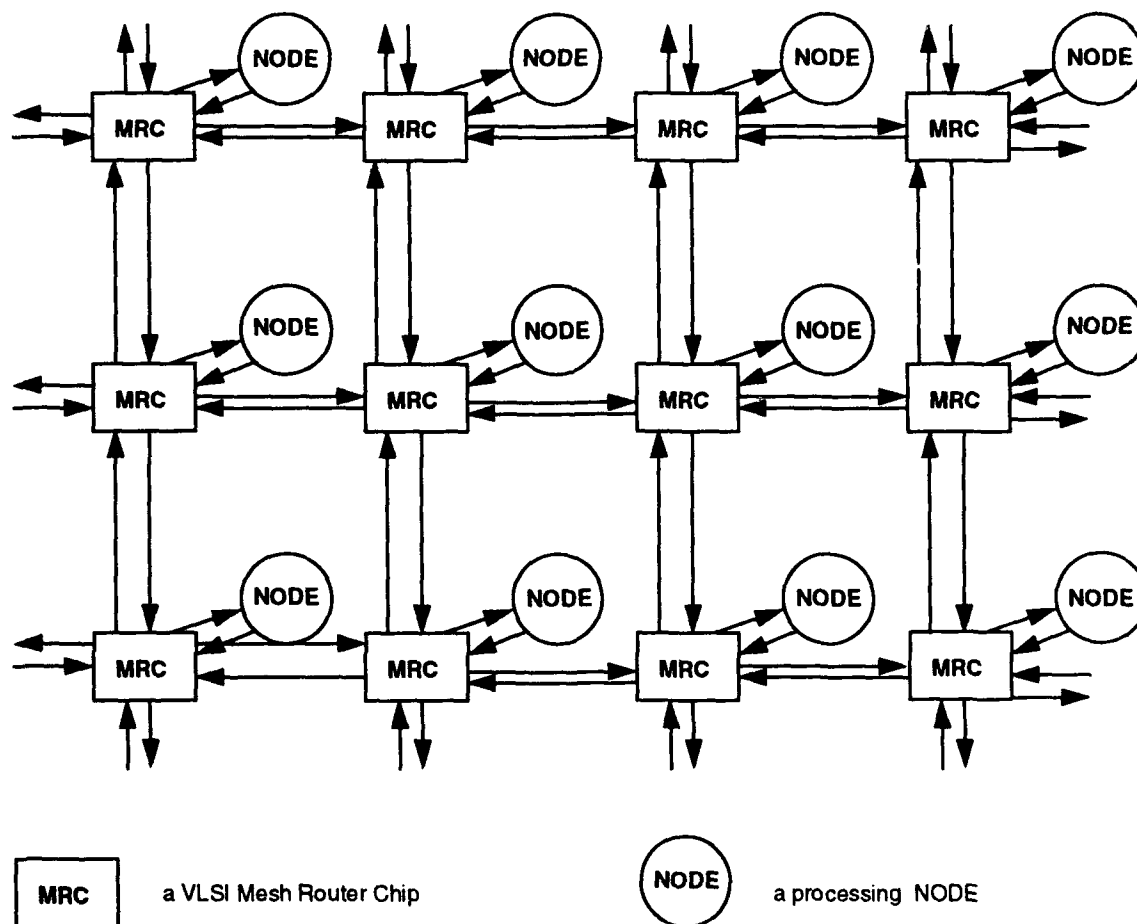


Figure 8.

The system contains two system buffers (array of packets) that are statically located at the system initialization time. The first is called the free pool. This space is used on a first-come-first-served basis. The free pool allows large messages (or many small ones) to be buffered by the receiving node. The free pool can hold up to 2884 packets. The second is called the reserved pool. It is logically divided among all of the other nodes in the system. It guarantees that a certain number of packet buffers are available for every other node in the system. The reserved pool has enough space to reserve 6 packets for each other node in the system.

Each node maintains 3 counters for every other node in the system to keep track of flow control of the message passing protocol. One counter (c-1) keeps the number of packet buffers the node has reserved for each other node in the system. Another counter (c-2) keeps the number of packet buffers that each other node in the system has for this node. Finally, a third counter (c-3) keeps the number of packet buffers that the node owes each other node in the system.

Each potential sender is guaranteed at all times to have a certain amount of buffer space at the receiver. Both sender and receiver know this (by the counters). The sender can send packets to the receiver until it has used up its guarantee. Then the sender blocks until the receiver "gives back" some guaranteed buffer space.

At the receiver, the guaranteed buffer space is replenished immediately if possible. This can be done in one of two ways. If the incoming packets had to be buffered, but the free pool had space, then the guaranteed buffer space is replenished from the free pool. Another way to think of it is that the packets were actually put in the free pool, and thus did not consume guaranteed space. If the incoming packets did not actually have to be buffered (because the corresponding receive had already been issued), then the guarantee is just incremented to account for buffer space not actually consumed.

The receiver keeps track of the sender's view of the guarantee, as well as its own. When the receiver finds that the sender's view has gone below some threshold, then the receiver "gives back" the difference between its view and the sender's by including it in the header of a packet going the other way, if there is one, or by sending a packet just for that purpose.

Given N is the number of buffers reserved for each node in the reserved pool, the communication protocol on the DELTA can be described as follows: if a receive was posted for the message before the message arrived, the sender decrements c-1 and the receiver decrements c-2 and increments c-3. If a receive was posted when the message arrived, but the free pool was not full, the sender decrements c-1 and the receiver decrements c-2 and increments c-3. Finally, if a receive was not posted when the message arrived, but the free pool was full, the sender decrements c-1 and the receiver decrements c-2 but does not increment c-3. When c-3 goes to zero, senders get blocked (prevented from sending messages to that node). The above three counters (c-1, c-2, c-3) enable the

sending processors of knowing if the receiver has enough space to provide temporary storage for the message until its corresponding receive is posted. This is a kind of hand-shaking that helps in minimizing the chances of errors in the message passing.

The DELTA uses its bi-directional communications links to implement a wormhole routing algorithm. The wormhole name is chosen to indicate that even though the message is sent packet-by-packet, the routing algorithm guarantees that the packets will arrive at the destination node in the same order they were sent. Messages on the DELTA are broken into packets before they are sent. Each packet consists of a 32 bytes header and a maximum of 480 bytes of data. The DELTA provides up to a 10 megabytes/second link bandwidth and a nearest neighbor hardware latency (delay) of less than 1 microsecond.

The DELTA communication library provides for both synchronous (blocking) and asynchronous (non-blocking) communication functions. The synchronous send function call (`csend()`) and the asynchronous send function call (`isend()`) take the following form:

```
csend(type, buf, len, node, pid)
isend(type, buf, len, node, pid)
```

where:

`type`: Is the type of the message being sent.

`buf`: Is a pointer to the buffer that contains the message being sent.

`len`: Is the size (in bytes) of the message. Message size is limited only by the memory available for the buffer.

`node`: Is the node to receive the message being sent. Setting `node` to -1 implies sending the message to all nodes except the sending node.

`pid`: Is the ID to receive the message (always 0; other values ignored.)

The synchronous receive function call (`crcv()`) and the asynchronous receive function call (`irecv()`) take the following form:

```
crcv(typesel, buf, len)
irecv(typesel, buf, len)
```

where:

`typesel`: Is the message type.

`buf` : Pointer to the buffer in which to store the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.

`len` : Is the size (in bytes) of the message. Message size is limited only by the memory available for the buffer.

As explained above, `crecv()` is a synchronous call. The calling process waits until the receive completes. To receive a message without blocking the calling process, `irecv()` (an asynchronous call) instead of `crecv()` is used. To achieve the correct operation of `irecv()` and `isend()`, the DELTA communication library provides another two function calls that informs the programmer when `irecv()` and `isend()` are completed. The first is `msgwait()` which takes the message ID as a parameter and returns when `isend()` or `irecv()` are completed. The second is `msgdone()` that returns TRUE if `isend()` or `irecv()` are completed, otherwise it returns FALSE.

`Msgwait()` is a synchronous function call while `msgdone()` is asynchronous. `Msgwait()` blocks the calling process until the corresponding `isend()` or `irecv()` is completed. On the other hand, `msgdone()` checks if the corresponding `irecv()` or `isend()` is done or not, and returns TRUE or FALSE (does not wait).

Hypercube

The iPSC/2 system consists of a collection of single board processors or "nodes" interconnected with full-duplex bit-serial channels to form a hypercube. In a hypercube where each node has N nearest neighbors nodes, the system is said to have dimension N. The nodes are assigned unique addresses so that the address of any two nearest neighbors nodes differ only by one binary digit. The dimension of a channel between two nodes is determined by taking the binary exclusive-or of the two nodes addresses. The bit position that remains a one is the dimension of that channel. For example, the channel connecting nodes 5 and 7 is determined by exclusive-or of 111 and 101. The result is 010 and because the "one" is in bit position one, that channel is in dimension one (Fig. 9).

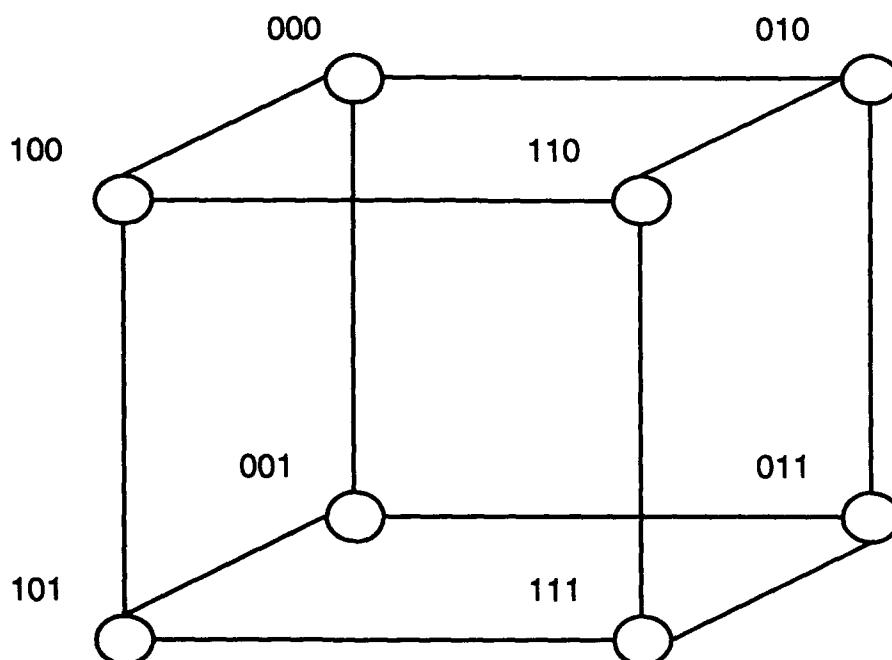


Figure 9.

Each hypercube node has a Direct-Connect router which allows simultaneous bi-directional message traffic between any two nodes. The routers form a circuit-switched network that dynamically creates a synchronous path from a source node to a destination node, and these remain open for the duration of the message. The path is composed of a series of channels that form a unique route from the source node to the destination node and may pass through some number of intermediate routers associated with other nodes.

The Direct-Connect router supports connections for eight full duplex channels and can be interconnected to form a network of up to seven dimensions containing 128 nodes. Each of the eight channels is routed independently allowing up to eight messages to be routed simultaneously. The router communicates with nodes over two unidirectional parallel busses.

The combination of channels that compose a path are defined by the e-cube routing algorithm. The algorithm guarantees deadlock free routing by only allowing the messages to be routed in increasingly higher dimensions channels until the destination is reached.

For example, looking at Fig 9, one can see there are two paths with the same length that a message sent from node 010 to node 111 may take. The first is to send the message to node 110 and then to node 111. The second path is to send the message to node 011 then to node 111. But using the e-cube routing algorithm, the first path must be taken since XORing 010 and 111 results in 101. The first 1 in the result (left most 1) is in position 2 is of higher than the 1 in position 0 (right most). Complementing the bit in position 2 in the source id (010) we get 110. So, the message is first sent using the channel that connects node 010 and 110 then to node 111.

A complete path is built in a step-by-step process involving arbitration for additional path segments at each router. The channels that constitute a path are held for the duration of the message. A channel is released when the tail of a message passes between the routers connected by that channel. Taking over the whole path eliminates the need for flow control buffering in the intermediate routers.

Like the DELTA, each node in the iPSC/2 maintains 3 counters for every other node in the system to keep track of flow control of the message passing protocol. The functionality of these 3 counters is the same as the ones for the DELTA and will not be repeated here. Also, as with the DELTA, the iPSC/2 system contains two system buffers (array of packets) that are statically located at the system initialization time: the free pool and the reserved pool. The functionality of these two system buffers on the iPSC/2 differ from that on the DELTA.

The free pool on the iPSC/2 is used on a first-come-first-served basis for only long messages (>100 bytes). The reserved pool is logically divided among all of the other nodes in the system. It guarantees that a certain number of short messages (<= 100 bytes) buffers are available for every other node in the system.

The iPSC/2 provides two levels of communication protocol, one for short messages and one for long messages. Messages of 100 bytes or less use one trip protocol. The reserved pool for each node provides a large number of short message buffers to provide temporary storage for short messages. When a node wants to send a short message to another node, if the receiving node has a reserved buffer to receive the short message, the sending node sends a probe to take over the path and then transmits the message. If the receiver had its reserved buffers used up, the sending node holds the message until a reserved buffer is available to receive the message.

Messages longer than 100 bytes use a three-trip protocol. The sending nodes first send the first 100 bytes of the message the same way it sends short messages. This first 100 bytes serves as proxy for the entire message. The proxy gets saved in the reserved pool at the destination node until a receive is posted for the message or there are enough spaces in the free pool to receive the whole message. When one of the above conditions become true, the operating system sends the sending node a control message to send the rest of the message. Receiving this control message, the sending node sends the rest of the long message. When the rest of the message arrives at the destination node, the operating system puts the message together and places it in the free pool or in the application buffer.

The iPSC/2 system uses its bi-directional communications links to implement a wormhole routing algorithm that provides 2.8 megabytes/second bandwidth. Also, the iPSC/2 guarantees a nearest neighbor hardware latency of 25 microseconds. The iPSC/2 communication library is the same as that of the DELTA described previously.

CM-5

A Connection Machine model CM-5 system can contain thousands of computational processing nodes. These nodes are connected by a Control Network and a Data Network. We focus our discussion on the Data Network which provides point-to-point communication that is considered in this model.

The CM-5 Data Network is a 4-ary fat tree - so called because some branches are "fatter" (of higher bandwidth) than others (see Fig. 10). The size of the CM-5 Data Network is often described by its height, which is the base-4 logarithm of the number of network addresses spanned. The height of the network equals one-half the number of bits in a processor address. A CM-5 scale 3 system, for example, contains a height-3 fat tree, which can span $4^3 = 64$ network addresses (see Fig. 10).

Each internal node of the fat tree is implemented as a set of Data Network switches, each a separate VLSI chip. The number of switches per node depends on where it is in the tree: the closer to the root, the fewer nodes and the more switches per node. Each switch has four children and either two or four parents. Fig. 10 illustrates a fat tree with 16 leaf nodes.

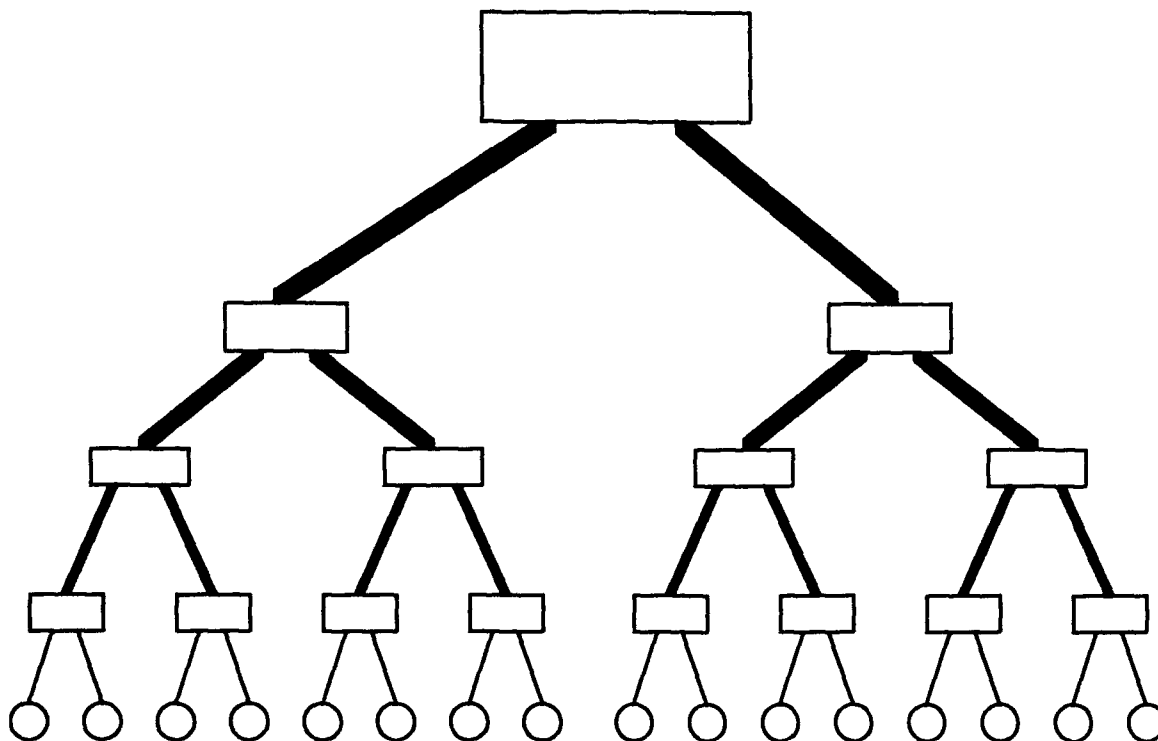


Figure 10.

The routing algorithm is very simple. The message can take any path up the Data Network fat tree. Once the message has reached the necessary height in the tree, it must then follow a particular path down to its destination.

The processor breaks the outgoing message into packets and transmits the message packet by packet to its destination node. When sending a packet, the processor writes it to the memory mapped Network Interface (NI) chip, and the NI is responsible for sending the packet via the fat-tree intercommunication network to the destination processing node NI. The destination NI alerts (or is polled by) the SPARC chip that there is a packet waiting to be received.

The Connection Machine communication library provides for both synchronous (blocking) and asynchronous (non-blocking) communication functions. The library also provides a number of global functions that operate under the same general protocols as the point-to-point functions. The global functions include:

- broadcasting data from one node to all nodes
- reducing data from all nodes to all nodes or to the host (like global sum)
- performing scans (parallel prefix operations) across the nodes
- performing segmented parallel prefix operations
- concatenation of elements into a buffer on all nodes, or into a buffer on the host

Communication on the CM-5 can be explicit or implicit. In explicit message passing the programmer must specify the destination node, the starting address of the buffer to send and its length. Implicit communication is implied through the data type used, for example, if A and B are arrays and X is scalar quantity, the statement $A = B + X$ implicitly broadcasts X to all processors so that the value of X can be added to every element of B.

General Communication Model Outline

We will start the description of the PST communication model by defining some terms. The Evaluation Module in the Parallel Simulation Tool, has a scheduler that keeps a list of all the processes in the system. For each process, the scheduler associates a handler, data and time. The scheduler calls the handler for the process with the lowest time to evaluate the process. Hence, when a process's time is set to INFINITY (very large time), the process is blocked because its handler will not be called until its time is changed from INFINITY to a smaller time.

The Evaluation Module (EM) keeps two kinds of handler: Processor and Temp. EM associates with each processor a Processor handler which evaluates one line of its code then advances to the next line of code to evaluate it and so on. Temp handlers evaluate one line of code but does not advance to the following line. So, if a processor's handler was changed from Processor to Temp, the processor gets stopped at a line of code and cannot advance to the next line. The communication model uses this method to force processors to wait (block) until some commands (synchronous operations) finish.

When a processor initiates an operation whose completion depends on another processor in the system such as a receive command which does not complete until the message arrival from another node, evaluations of such an operation depends on whether it was done synchronously or asynchronously. If a processor's operation was done synchronously, the communication model replaces the calling processor's Processor handler and data with a Temp handler and temporary data to complete evaluating it. For asynchronous processor operations, the communication model creates a new process and associates with it a Temp handler that completes evaluating the operation.

Notice that the processor executing a synchronous operation gets blocked because its handler was replaced by a handler which does not advance to the next line of code. In the asynchronous case, the processor's handler was not replaced, so it can advance to the next instruction.

Having defined some terms, lets turn our attention to the communication model. Recall that the DELTA, the iPSC/2 and the CM-5 are distributed-memory MIMD systems. Systems of this kind are made up of a set of nodes, each of which consists of a main processor, memory, and interface to a network that connect them together. Nodes process information independent of one another and communicate only by sending and receiving messages.

The communication network used on the iPSC/2, the DELTA and the CM-5 allows messages to route through intermediate nodes (nodes on the path from the source to the destination) without interrupting the processes on these nodes, thus the communication model presented views the communication network as another process in the system with a dedicated task. This task of delivering messages is viewed to run independently from the other processes in the system.

When the network accepts a message, it takes the full responsibility of delivering it to the destination node. This allowed a communication model that divides point-to-point communications on the above three machines into two parts: the processor part and the communication network part. A processor may execute a send, receive, message wait or message done function call. The send or receive function call may be done synchronously or asynchronously. Message wait is a synchronous function call while message done is an asynchronous one.

The communication network's role in the model is to accept a message from one node and deliver it to another. This includes finding the correct route from the source node to the destination node which is different for each topology, updating the links (called channels on the iPSC/2) times and preventing message colliding. The operation of the communication network is the same as if the message passing function call was performed synchronously or asynchronously.

The network consists of nodes that are connected by links. These links are used by all the processors in the system to send and receive messages. Sharing this one communication network among all the processors requires some kind of scheduling to prevent messages collisions. The DELTA, the iPSC/2 and the CM-5 communication models adopt the first-come-first-served scheduling policy to prevent message collisions.

When a processor executes an asynchronous send (`isend()`) function call, it posts a send request to the communication network and continues to the next instruction. In our communication model, this is evaluated by increasing the sending processor time by the time to execute `isend()` function call and by creating a new process which continues evaluating the send operation. By this we ensure that the communication network accepted the message. When the communication network accepts the message this process gets removed from the list of processes that need to be evaluated.

If a processor executes an asynchronous receive (`irecv()`) function call, it posts a receive request to the communication network and continues to the next instruction. This is evaluated by increasing the processor time by the time to execute `irecv()` function call and by creating a new process that continues evaluating the receive operation. Continuing the `irecv()` may require waiting till the message arrives at the receiving node and copying it from the communication network to the application buffer specified by the `irecv()`.

Asynchronous send and receive function calls have another two function calls associated with them: message wait (msgwait()) and message done (msgdone()). These two function calls are used to determine whether isend() or irecv() operations identified by the message id is completed or not. This message id is given to the message when isend() or irecv() is initiated.

The PST communication model associates a FALSE value with each message id when isend() or irecv() is initiated and a TRUE value when they are complete. Evaluating msgdone(id) is performed by returning the value associated with this message id to the calling processor and by incrementing its time by the time to execute msgdone(). The msgwait() function call is evaluated differently because it is a synchronous function call.

Evaluating msgwait(id) function call is performed by replacing the calling processor's data and handler by a handler and data that continue evaluating it. This handler gets called repeatedly until the value associated with the message id in question is TRUE. When this happens, the calling processor time is set to the time when msgwait() is completed and its original data and handler get restored so it can continue to the next instruction.

PST evaluates synchronous send (csend()) and receive (crecv()) in the same way it evaluates isend() and irecv() with one exception. Instead of creating a new process that continues evaluation of the csend() or crecv(), the calling processor's data and handler get replaced by a temporary communication data and handler that continue evaluation of the operation. Also, the calling processor's time does not get changed until the operation is completed. This is because in a synchronous message type the sending process waits (blocks) until the message has left the sending process memory. This means the message was copied by the communication network but does not mean that the message has been received by the destination node. The process executing the receive waits (blocks) until the message arrives in the application buffer specified by the receive command. Remember, in asynchronous (non-blocking) message type the sending and the receiving processes continue to run while send or receive are being carried out.

In the DELTA and the iPSC/2, each processing node keeps a buffer which provides a temporary storage for messages which arrives earlier than their matching receive. This buffer is divided into two buffers on the DELTA and on the iPSC/2: the free pool and the reserved pool. The CM-5 does not provide such a buffer because its communication protocol does not allow sending a message before its corresponding receive is posted.

The reserved pool and the free pool at the receiving node effects the operation of the sending processes. The sender must insure that the corresponding receive for the message is posted at the receiver node or the receiver has enough space in its reserved pool or free pool buffers to receive the message. Otherwise, the sending process gets blocked until one of the above two conditions becomes valid.

The operation of the handlers which complete evaluating the send and receive operations are machine dependent: their role in the communication model will be presented when the

detailed communication model is discussed in the following 3 sections. The next section describes the communication model for the Intel Hypercube system. Later sections describe the differences of this model to those of the Delta and CM-5.

Hypercube Model

The communication module for the iPSC/2 starts by initializing the communication network channels, the system buffers and other data needed by the model. Initialization is performed by calling `iPSC_Communications_init()` which perform the following steps:

```
iPSC_Communications_init()
{
    1- initialize the communication network channels
    2- initialize the free_pool_sizes array to free_pool_size
    3- initialize the free_pool_msgs array to NULL
    4- initialize the reserved_pool_sizes array to recv_pool_size
    5- initialize the reserved_pool_msgs array to NULL
    6- initialize the recv_posted_msgs array to NULL
    7- initialize the recv_posted array to not_posted
    8- initialize the msg_id array to FALSE
    9- initialize the msg_id_time array to INFINITY
    10- initialize the message id counter to 0
    11- initialize the blocked_process array to FALSE
}
```

Step 1 above creates for each processing node in the system a number of channels that equals the dimension of the hypercube. For example, if the hypercube has a dimension of 5 (32 nodes), then each node has 5 nearest neighbors. So, step 1 above creates 5 channels for each node ($5 * 32 = 160$ channels).

Each channel has the following data associated with it: id, processor_num, avail, probe, in_use and busy. After creating the channels, step 1 gives each channel an id which is used to refer to it. Also, each channel is given a node number (processor_num) to which it belongs. Then, each channel is given a time which indicates when the channel is available to be used (avail). This time is initialized to 0 since each channel is available to be used when the system is started. The remaining data associated with each channel (i.e. probe, in_use and busy) will be explained later when they are used.

As explained earlier, each node in the iPSC/2 keeps two buffers to provide temporary storage for messages which arrive earlier than their matching receive. Each node keeps one free pool that provides temporary storage for long messages (> 100 bytes). The Parallel Simulating Tool (PST) allows users to change the size of the free pool to examine the effect of the free pool size on the program performance. Thus, step 2 above initializes each node free pool's size to the size provided by the user. And since at the start of PST, all the free pools contains no messages, step 3 initializes them to contain NULL.

Also, each node keeps several 100 bytes buffers for each other node in the system which provides temporary storage for short messages called the reserved pool. PST allows users to change the size of the reserved pool to see how this may effect the program performance. Thus, step 4 above initializes each node's reserved pool's size to the size provided by the user. As for the free pool, they are initialized to NULL.

In the iPSC/2, when a message whose corresponding receive is posted arrives at its destination node, it goes to the application buffer specified by the receive command. To make the operation of the communication model easier to understand and debug, we chose to place the message in a temporary storage (`recv_posted_msgs`) to be copied later to the application buffer. Step 6 above initializes this temporary storage to NULL, meaning it contains no messages.

When a processor issues a receive command, it posts a receive request to the communication network. When the receive is completed, this request is deleted. The PST communication model uses a 2-dimensional array to accomplish this. When a processor issues a receive operation, it places the message type in the column that corresponds to the processor number in the `recv_posted` array. So, when a processor tries to send a message to this processor, it checks the receiver column in `recv_posted` array to see if the receiver had posted a receive request to message with the same type to that it is sending. Since message types are positive numbers, step 7 initializes the `recv_posted` array entries to `not_posted` indicating no receive request is posted yet.

The two arrays initialized in steps 8 and 9 and the counter in step 10 are used to provide proper operation for `msgdone()` and `msgwait()` function calls. When a processor issues `isend()` or `irecv()` function call, the message is given an id (= the counter value) and the value that corresponds to this id in `msg_id` array is set to FALSE. When `irecv()` or `isend()` completes, the FALSE value is changed to TRUE and the time of completion is placed in the position that corresponds to the message id in `msg_id_time` array.

Step 8 initializes all positions in `msg_id` array to FALSE (not completed yet) and all positions in `msg_id_time` array to INFINITY (a large value). The counter in step 10 is used to give `isend()` and `irecv()` messages an id. It is initialized to 0 in step 10.

Finally, `iPSC_Communications_init()` initializes all the entries in `blocked_process` array to FALSE in step 11. In PST, each process is given a number that distinguishes it from the other processes. If a process gets blocked because of a message passing operation, the value that corresponds to its number in `blocked_process` array is set to TRUE to indicate it is blocked. This is done to keep track of all the processes that are blocked because of message passing. When a change happens on the communication network that may unblock this process, the blocked process gets activated to continue its operation. Since at the beginning of each program none of the processes are blocked, step 11 indicates that by setting all the entries in `blocked_process` array to FALSE.

After the initialization, the PST communication model is ready to evaluate the message passing operations. Lets follow a message starting from when the send was issued until the message is received. When a csend() function call is issued by a processor, iPSC_Csend() handler gets called which performs the following steps:

```
iPSC_Csend()
{
    1-set send type to CSEND.
    2-if( the message size <= 100 ) flag = 0 else flag = 3.
    3-create a probe for the message.
    4-save the probe and the processor's time in a temporary communication
      data.
    5-replace the calling processor data and handler by the temporary
      communication data and iPSC_send_cont().
}
```

Step 1 sets the send type to 1. The send type is set to CSEND for csend(), ISEND for isend(), CRECV for crecv(), IRECV for irecv() and MSGW for msgwait(). The value of send type is used later by the communication model and will be explained at that time.

The flag's value in step 2 is used to keep the stage in which the send operation is in. The send operation can be in one of 9 stages which will be explained later.

The probe created in step 3 has two purposes. The first is to save the message parameters (i.e. message size, source, destination, type, id) and some information that helps in delivering the message to its destination, such as the flag and send type mentioned above. The second purpose is to take over (control) all the channels from the source node to the destination node which are required to deliver the message.

Since csend() is a synchronous operation, the processor has to wait (block) until csend() completes. The PST communication model simulates the blocking of a processor by saving its data and handler and replacing them by a temporary communication data and a handler which continues evaluation of the blocked operation.

Step 4 saves the probe that holds all the necessary information to complete the csend() operation and the processor time in temporary communication data. Then step 5 blocks the sending processor by saving its data and handler and replacing them by the temporary communication data and iPSC_send_cont() which completes evaluation of the csend() operation. The Evaluation Module in PST associates a handler and data with each process which specifies its functionality. Step 5 above specifies the blocked processor's handler as iPSC_send_cont() and the temporary communication data as its data.

The send operation also can be performed by an isend() function call. If a processor issues an isend() function call, the communication model evaluates it by calling iPSC_Isend() handler that performs the following steps:

iPSC_Isend()

```
{  
    1- give the message an id and set msg_id[id] to FALSE and  
        msg_id_time[id] to INFINITY.  
    2- set send type to ISEND.  
    3- if( mess_size <= 100 ) flag = 0 else flag = 3.  
    4- create a probe for the message.  
    5- save the probe and the processor time in a temporary communication  
        data.  
    6- create a new process and put iPSC_send_cont as its handler and  
        the temporary communication data as its data.  
    7- increment the calling processor time by the time to execute isend().  
    8- return the message id.  
}
```

Step 1 gives the message an id that can be used by msgdone() or msgwait(). Asynchronous send and receive function calls have another two function calls associated with them: msgwait() and msgdone(). These two function calls are used to determine whether the isend() or irecv() operation identified by the message id (given in step 1) is complete. The PST communication model associates a FALSE value with each message id when isend() or irecv() is initiated and a INFINITY time as its completion time. This information is kept in msg_id and msg_id_time arrays in step 1.

Step 2 sets the send type to ISEND indicating an isend() function call. Steps 3, 4 and 5 are the same as iPSC_Csend() steps 2, 3 and 4. Step 7 increases the sending processor time by the time to execute the isend() function call. Since isend() is an asynchronous operation, the calling processor does not get blocked, thus step 8 creates a new process that completes the send operation. The Evaluation Module in PST keeps a list of all the processes in the system. It also associates a handler and data with each process that specifies its functionality. Step 8 above creates a new process and specifies iPSC_send_cont() as its handler and the temporary communication data as its data. This new process will be scheduled to run until the isend() is completed, it will get removed from the list of processes when isend() completes.

Notice the differences between iPSC_Isend() and iPSC_Csend() handlers. Since isend() is an asynchronous operation, the calling process was not blocked but its time was increased by the time to execute isend(). For isend() the handler created a new process that continues its evaluation, but for csend(), the calling process data and handler were replaced to continue the operation.

iPSC_send_cont() handler continues the evaluation of both csend() and isend() operations. When iPSC_send_cont() handler gets called, it performs the following steps:

```

iPSC_send_cont( )
{
    case 1: message size less than or equal to 100 bytes,
        if did not get space (flag = 0), then get space and set flag to
        1.
        if got space but did get first channel (flag = 1), then get first
        channel and set flag to 2.
        set the process's time to INFINITY, add it to the
        blocked_process array.

    case 2: message size greater than 100 bytes, sending the first 100 bytes
        if did not get space (flag = 3), then get space and set flag to
        4.
        if got space but did get first channel (flag = 4), then get first
        channel and set flag to 5.
        set the process's time to INFINITY, add it to the
        blocked_process array.

    case 3: message size greater than 100 bytes, sending the rest of the
        message
        if did not get space (flag = 7), then get space and set flag to
        8.
        if got space but did get first channel ( flag = 8), then get first
        channel and set flag to 9.
        set the process's time to INFINITY and add it to the
        blocked_process array.

    case 4: got space and first channel, waiting for the message delivery
        flag = 2, 5, or 9
        set the process's time to INFINITY and add it to the
        blocked_process array.
}

```

Now we can explain the use of the flag's value mentioned above. The flag value is used to save which stage the send operation is in. The send operation has three stages. Stage 1 consists of ensuring that the destination node has reserved space to receive the message. Stage 1 is followed by stage 2 which tries to place the message's probe on the first channel toward the message destination. Stage 3 is the waiting stage. After a process places a message's probe on the first channel it must wait until the probe controls all the channels required to transfer the message. Also, since the iPSC/2 provides two different communications protocols, the flag's value is used to indicate to the communication model which protocol to use.

Case 1 handles the short message's one trip protocol. As explained earlier, before sending any message the sending processor must make sure there is space at the destination node

to receive the message. For short messages the space may be in the application buffer if the matching receive was posted for the message or in the reserved pool. The sending processor checks the column that corresponds to receiver in `recv_posted` array to see if the receiver had posted a receive request for the message. If the receive was not posted, the sending processor checks if the destination node has enough space in its reserved pool to receive the message by checking the entry that corresponds to the sending processor number and the receiving processor number in the `reserved_pool_sizes` 2-dimensional array. If the above entry is not equal to zero, space is found. Finding enough space in the reserved pool, the reserved size is decreased by the size of the message.

If no space is found to receive the message, the process calling `ipsc_send_cont()` time gets set to INFINITY. Setting a process time to INFINITY prevents the processes scheduler from calling it to be evaluated. Recall that the processes scheduler calls the process with the lowest time to be evaluated. This time gets changed to the original time when a space to receive the message becomes available.

Having ensured that the destination node has a space to receive the message, the sending node tries to place the probe on the first channel towards the destination node. Getting the first channel requires checking if the channel is not `in_use` and is not busy. A channel is busy when it is used by another message at the time of checking. A channel is `in_use` when it has a probe on it. The real distinction between these two terms will become clear later.

If the first channel toward the destination is not busy or `in_use`, the sending process places the message probe on this channel and labels the channel as busy and `in_use`. This labeling is done to indicate to all the other nodes in the system that they can't use the channel at this time. Having placed the probe on the first channel, the process must wait until the probe controls all the channels required to transfer the message. We indicate that the processor is waiting (case 4) by setting its time to INFINITY and setting the entry that corresponds to its number in the `blocked_process` array to TRUE. This time will be changed when the probe takes over the required channels and the message transmission starts. Similar to getting a space, if getting the first channel failed (the first channel was busy or `in-use`) the process calling `ipsc_send_cont()` is blocked until the channel become available.

Case 2 and case 3 handle the three-trip protocol used by the `ipsc/2` for long messages. Case 2 handles sending the first 100 bytes to the destination node and case 3 handles sending the rest of the message. One can easily see the similarity between case 1 and case 2 in that both of them first get space and then get the first channel towards the destinations. We chose to make them two separate cases to stress that case 2 must be followed by case 3.

As with case 1, case 2 finds a space at the destination node then places the message probe on the first channel toward the destination. After delivering the first 100 bytes, the probe flag is set to case 3. This indicates that the first 100 bytes were delivered and it is time to send the rest of the message.

The steps performed by case 3 are the same as those performed by both case 1 and case 2 except in getting the space to receive the message at the destination node. While case 1 and case 2 look for space only in the reserved pool because they are sending 100 bytes or less, case 3 first looks for space by searching the `recv_posted` array to see if the destination node had posted a receive request for a message with the same type as the type of the message in the send operation. If such a receive request was found it gets deleted from `recv_posted` array and the process tries to place the probe on the first channel towards the destination. Otherwise, case 3 looks for space in the destination free pool but not in the reserved pool. This is because the reserved pool is designed to provide temporary storage for messages with sizes less than or equal to 100 bytes only.

After ensuring that the destination has enough space to receive the message, case 3 places the message probe on the first channel towards the destination. Doing that the process has to wait until the probe reaches the destination node, thus it gets blocked by setting its time to INFINITY.

In the above three cases, when getting a space to receive the message, an indication is placed in the probe of where that space was taken from. This is important to indicate to the communication network where to place the message when it arrives at its destination: i.e. the reserved pool (`reserved_pool_msgs`), the free pool (`free_pool_msgs`) or in the application buffer specified by the corresponding receive command (`recv_posted_msgs`).

Having placed the message probe on the first channel towards its destination, the responsibility of moving it to its destination is passed to the communication network. The PST communication model views the communication network as another process in the system that needs to be evaluated. Thus, it keeps a handler that is called to evaluate it. This handler is called `Handle_iPSC_Communications()`. When called by the process scheduler, it updates the channel with the lowest time which is `in_use`. `Handle_iPSC_Communications()` performs the following steps:

```
Handle_iPSC_Communications()
{
    1- find channel that is in_use with lowest time whose probe is not blocked.
    2- if no channel is used, return INFINITY
    3- if the probe on this channel (old_channel) has reached its destination
        call iPSC_message_deliver(). return.
    4- new_channel = find next channel using the probe information
    5-if( the new_channel is not busy)
        move the probe to the new_channel.
        set the old_channel as not in_use
        set new_channel as in_use and busy, set its time and return.
    6- else
        block the old_channels, go to step one
}
```

Step 1 searches all the system channels to find the channel with the lowest time which is in_use (has a probe on it) and whose probe is not blocked. The difference between in_use and busy used in labeling channels is now explained. The channel which has a probe on it is labeled as in_use. This indicates to Handle_iPSC_Communications() that the channel has a probe that needs to be advanced towards its destination. When a probe moves from channel A to channel B, channel A is labeled as busy meaning it is controlled by (reserved to) this probe while channel B is labeled as in_use and busy meaning it is controlled by the probe and it has a probe on it which needs to be advanced. Also, a probe is blocked if the next channel towards its destination is busy. In other words, it can't be advanced.

If no such channel is found, Handle_iPSC_Communications() returns INFINITY, step 2. INFINITY here is used to prevent the processes scheduler from choosing Handle_iPSC_Communications() to be evaluated. This is because not finding a channel which is in_use and whose probe is not blocked means that either there is no active messages in the system or that the communication network has to wait until a process releases some channels. In other words, there is nothing to evaluate on the communication network. As noted above, the scheduler chooses the process with the lowest time to be evaluated. As a result, a process with time INFINITY will not get scheduled to be evaluated.

The old_channel in step 3 is the channel with the lowest time that was found in step 1. If the probe on the old_channel has reached its destination i.e. has controlled all the channels required to transmit the message, iPSC_message_deliver() - explained below- is called to handle the message transmission.

Step 4 finds the next channel towards the probe destination (new_channel). The new_channel may be in_use by another probe or may be available to be used by the current probe. Step 6 handles the first case and step 5 handles the second. Step 5 moves the probe from the old_channel to the new_channel and labels the new_channel as in_use and busy and the old_channel as busy only. As explained earlier, a channel is busy if it is controlled by a probe but does not have a probe on it. A channel is in_use if it has a probe on it that needs to be advanced to its next channel.

After moving the probe to the new_channel, step 4 sets its available time to the maximum of the old_channel and the new_channel time plus the channel latency. The channel latency is the nearest neighbor hardware latency which is 25 microseconds on the iPSC/2. The communication model was designed to allow users of PST to change the channel latency to investigate how this may effect the program performance.

We arrive at step 6 if the probe on the channel with the lowest time is blocked. Step 6 labels this probe as blocked so that it will not be chosen again to be evaluated until a change happens on the communication network that may unblock it. Failing to update the channel with the lowest time, step 6 tries to find another channel to update by going back to step 1.

As mentioned above, when a probe reaches its destination node i.e. has controlled all the channels required to transmit the message, `iPSC_message_deliver()` is called to handle the message transmission. It performs the following steps:

```
iPSC_message_deliver()
{
    1-set the first channel's time.
    2-label the first channel as not busy and not in_use.
    3-free all the used channels and unblock all the blocked probes.
    4-remove the probe from the last channel and place the message in
      the destination buffer.
    5-unblock all the processes that were blocked because of a message
      passing operation and activate the communication network handler
      case 1: message size is less or equal to 100 bytes
          set message receive time = to the last channel's time +
          channel latency
          if the send was ISEND, msg_id
              array[id]=TRUE; msg_id_time[id]=time;
          restore the sending process and return
      case 2: message size greater than 100 bytes, sending the first 100
        bytes
          set the probe flag to 7 and return
      case 3: message size greater than 100 bytes, sending the rest of the
        message
          set message receive time = to the last time + channel latency
          if the send was ISEND,
              msg_id array[id]=TRUE and msg_id_time[id]=time
          copy the first 100 bytes from the reserved pool
          restore the sending process and return
}
```

Care is taken when finding the first channel's (toward the destination) time in step 1. This is because message transmission starts after the probe reaches the message destination's node and because there are two different protocols: one for short messages and one for long messages. In general this is the equation used to find the first channel's time after the tail of the message passes by it's end:

$$\text{1st channel's time} = \text{last channel's time} + \text{msg_size} / \text{bandwidth} + \text{channel latency}$$

If the message size is less or equal to 100 bytes, the value used for `msg_size` above is the message size. But if the message size is greater than 100 bytes and we are sending the first 100 bytes, the `msg_size` equals 100. Finally, if the message size is greater than 100 bytes and we are sending the rest of the message, the `msg_size` equals the message size - 100. Also, in the third case the last channel in the above equation is actually the first channel,

since in the three-trip protocol, a control message is sent from the message destination to the message source node to start message transmission. After this control message arrives at the message's source the message transmission starts.

After setting the time of the first channel, it gets freed by labeling it as not in_use and not busy, step 2. This will enable other nodes to use this channel to send messages. Having set the time of the first channel and freeing it, the other channels used to transfer the message times need to be set and they need to be freed in step 3. Also, the probes which were blocked (because their required next channel was busy) get unblocked so they will not get ignored by the handler that advances probes.

A channel gets freed when the tail of the message leaves toward its destination. Thus a channel available time is set to the previous channel's time plus the channel latency. Step 4 removes the probe from the last channel and places the message at the destination buffer. This buffer may be the destination's reserved pool, free pool or the application buffer. Which one of the above buffers depends on from where space was reserved to receive the message in `iPSC_send_cont()` above. If the space was taken from the reserved pool, the message gets placed in the `reserved_pool_msgs` array, if the space was taken from the free pool, it gets placed in the `free_pool_msgs` array. Finally, if a receive was posted when the corresponding send was initiated, the message gets placed in `recv_posted_msgs` array.

Since delivering a message frees the channels used in transferring it to its destination, all the processes that were blocked for message passing reasons get unblocked in step 5 above. By unblocking the processes we mean changing their time from INFINITY to the time that was kept in the temporary communication data when they were blocked. Also, these processes are removed from `blocked_process` array by changing the entry that corresponds to their numbers from TRUE to FALSE. When unblocked, these processes check to find if the reason of blocking them is still valid or not and act accordingly.

What is left in the role of `iPSC_message_deliver()` in the communication model depends on the size of message being delivered. As before we have three cases to consider. Case 1 handles short messages. After delivering the whole message, the message receive time gets set to the last channel's time plus the channel latency. If the send command was done asynchronously (`isend()`), an indication of the `isend()` completion is placed in the `msg_id` array. As explained earlier when `isend()` is initiated its message is given an id and the position that corresponds to this id in `msg_id` array is labeled as FALSE. Now, the `isend()` is completed, this position is labeled TRUE. Also, the message receive time is placed in `msg_id_time` array to indicate when the `isend()` was completed. Also, the process that was created to continue evaluating `isend()` gets removed from the scheduler list of the processes that need to be evaluated. This is what is meant by "restore the sending process" at the end of case 1 if the send was done using the `isend()` function call.

When a processor executing a `csend()` was blocked its data and handler were replaced by the temporary communication data and `iPSC_send_cont()` handler to continue the send operation. After the message gets delivered, it is time to restore the original processor

data and handler. Also, its time is set to the first channel's time minus the channel latency. This is because a `csend()` is complete when the communication network accepts the message.

Case 2 in `iPSC_message_deliver()` above sets the probe flag to 7 indicating that the first 100 byte were delivered to the destination. Since not all the message was transmitted to its destination, the message receive time is not set and the sending process data and handler did not get restored as in case 1.

The steps taken by case 3 are the same as case 1 except the step that copies the first 100 bytes. For long messages, the sending processor first sends the first 100 bytes to their destination and then it waits for a control message from the destination to transfer the rest of the message. The first 100 bytes gets saved in the destination node's reserved pool but the rest of the message gets saved in the destination's application buffer or free pool buffer. Thus, after delivering the rest of the message the operating system on the iPSC/2 combines the two parts of the message together.

Up until here, we explained what steps the message has to go through to reach the receiving node. Now, we focus our attention on the receiving processor part of the communication operation. When a processor executes an asynchronous receive (`irecv()`) function call, the `iPSC_Irecv()` handler is called to evaluate it. When called `iPSC_Crecv()` handler performs the following steps:

```
iPSC_Irecv( )
{
    1-post a receive request in post_recv_request array
    2-increments the processor's time by the time to execute irecv()
    3-give the message an id
    4-set msg_id array[id] to FALSE and msg_id_time[id] to INFINITY;
    5-set the send_type to IRECV
    6-create a probe for the message
    7-save the probe and the processor's time in a temporary communication
      data
    8-create a new process and put iPSC_recv_cont as its handler and
      the temporary communication data as its data
    9-return the message id
}
```

If a processor executes an asynchronous receive (`irecv()`) function call, it posts a receive request to the communication network (step 1) and continues to the next instruction. This is evaluated by increasing the processor's time by the time to execute `irecv()` function call (step 2) and by creating a new process (step 8) that continues the receive operation.

As explained earlier, asynchronous send and receive function calls have another two functions calls associated with them: `msgwait()` and `msgdone()`. These two function calls

are used to determine whether `isend()` or `irecv()` operations identified by the message id (given in step 3) are complete. The communication model associates a FALSE value with each message id when `isend()` or `irecv()` is initiated and a INFINITY time at its completion time, step 4. This is performed to ease evaluating `msgdone()` and `msgwait()`.

PST evaluates a synchronous receive (`crecv()`) differently from the way it evaluates `irecv()`. A process executing the `crecv()` waits (blocks) until the message arrives in the application buffer specified by the receive command. When a processor executes a `crecv()` function call, `iPSC_Crecv()` gets called to evaluate it. `iPSC_Crecv()` performs the following steps:

```
iPSC_Crecv()
{
    1-set send_type to CRECV
    2-post a receive request in post_recv_request array
    3-create a probe for the message
    4-save the probe and the processor time in a temporary communication
      data
    5-replace the processor's data and handler by the temporary communication
      data and iPSC_recv_cont()
}
```

Step 1 sets the `send_type` to CRECV indicating the function call was a `crecv()`. Similar to the `irecv()` handler, step 2 posts a receive request and step 3 creates a probe and saves in it the necessary information to continue `crecv()` evaluation. Step 4 saves the probe and the processor's time in a temporary communication data and step 5 blocks the processor by replacing its data and handler by the temporary communication data and `iPSC_recv_cont()` to continue evaluation of `crecv()`.

Notice the difference between evaluating `irecv()` and `crecv()`. For `irecv()` the message was given an id and some other information were initialized to enable evaluating `msgdone()` and `msgwait()` function calls. Since `crecv()` is a synchronous function call, `msgdone()` and `msgwait()` are not valid for it. Also, in evaluating `irecv()` a new process was created to continue evaluating it while for `crecv()` the processor's handler was replaced by `iPSC_recv_cont()`. This is because `irecv()` is an asynchronous function call while `crecv()` is a synchronous one.

Lets turn our attention to `iPSC_recv_cont()` that continues evaluating `irecv()` and `crecv()`. It performs the following steps:

```

iPSC_rcv_cont()
{
    1-If the message has already arrived at the its destination node
    2-move it to the application buffer
    3-return the space taken by the message
    4-if irecv(), set the msg_id array and msg_id_time
        remove the process from the scheduler processes
    5-if crecv(),
        set the processor time
        restore the calling processor data and handler
    6-unblock all the processes which were blocked because of message
    passing
    7- else
        set the calling process time to INFINITY
}

```

Step 1 above searches only the calling processor reserved pool if the receive was for a short message, and only the free pool and the receive posted array for long messages to determine if a message with the same type to the type specified by the receive command has arrived or not. If the message did not arrive yet, the calling process time is set to INFINITY in step 7. Setting the calling process's time to INFINITY in step 7 is done to prevent the process scheduler from calling it until a change happens on the communication network that may enable it to continue its operation. As explained above, when a message gets delivered at its destination, all the blocked processes get unblocked to see if the reason of blocking them is not valid any more. When a message arrives at its destination, the iPSC_rcv_cont() time gets changed from INFINITY to the time saved in the temporary communication data to enable it to look if the arrived message is the message it requires.

If the message has already arrived, step 2 moves it to the application buffer specified by the receive command. If the message was found in the reserved pool or the free pool step 3 returns the space that the message occupied to the respective pool so it can be used by other messages.

Now the use of send_type mentioned above can be explained. If send_type value was equals to IRECV (irecv()) the completion of irecv() is declared in msg_id array by setting the entry that corresponds to the message id to TRUE in step 4. Also, the time of the completion is saved in msg_id_time. Doing that, the role of the process that was created to complete irecv() is completed. Thus, it gets removed from PST process scheduler in step 4.

If send_type value was equal to CRECV (crecv()), the calling processor time is set to the maximum of the processor's time and the message receive time. Finishing evaluating crecv(), step 5 restores the processor's data and handler which were replaced by

iPSC_recv_cont and the temporary communication data to enable the processor to continue to the next instruction.

Having explained how the PST communication model evaluates csend(), isend(), crecv() and irecv() we turn our attention to msgdone() and msgwait() evaluation. When a processor issues a msgdone() function call, it gets evaluated by iPSC_msgdone() handler which performs the following steps:

```
iPSC_msgdone(id)
{
    1- if ((msg_id array[id] is TRUE) AND
           (the processor's time <= msg_id_time[id]))
        increase the calling processor's time by the time to execute
        msgdone
        and return TRUE
    2- else
        increase the calling processor's time by the time to execute
        msgdone
        and return FALSE
}
```

The PST communication model associates a FALSE value with each message id when isend() or irecv() is initiated and a TRUE value when they are completed in the msg_id array. Also, when the irecv() or isend() is completed, the time of completion is saved in msg_id_time array. Evaluating msgdone(id) is done by checking these two entries and acting accordingly. If the position that corresponds to the message id in question in msg_id array is TRUE and the same position in msg_id_time has a time that is less than or equal to the calling processor time, iPSC_msgdone() increments the calling processor time by the time it takes to execute the msgdone() function call and it returns TRUE, step 1. Otherwise, the calling processor time gets incremented by the time to execute msgdone() and it returns FALSE, step 2.

The msgwait() function call is evaluated in a different fashion than msgdone() because it is a synchronous function call. When a processor issues a msgwait(id), it gets blocked and its data and handler get replaced by a temporary communication data and iPSC_msgwait_cont() handler that continue evaluating the msgwait() function call (step 3 below). iPSC_msgwait_cont gets called until the value associated with the message id in question is TRUE. When this happens, the calling processor's time is set to the time when msgwait() returned and its original data and handler get restored so it can continue to the next instruction.

```

iPSC_msgwait()
{
    1-create a probe and save the message id in it
    2-save the probe and the processor time in a temporary communication
    data
    3-replace the calling processor handler and data by iPSC_msgwait_cont
    and the temporary communication data
}

```

When iPSC_msgwait_cont() gets called by the processes scheduler it performs the following steps:

```

iPSC_msgwait_cont()
{
    1- if (msg_id array[id] is TRUE)
        set the processor time to:
            max(msg_id_time[id] and the processor time)
            + time to execute msgwait
        restore the processor data and handler
        return TRUE
    2- else
        set the calling process time to INFINITY
}

```

Step 1 above checks the value stored in the position that corresponds to the message id in set msg_id array. If this value was FALSE the calling process time is set to INFINITY (get blocked). Otherwise, the processor time is set to the maximum of the time stored in msg_id_time[id] and the processor time added to it the time to execute msgwait() function call. Then step 1 above restores the processor original data and handler that was replaced in iPSC_msgwait() handler.

CM-5 and Delta Extensions

Delta

The communication model for the DELTA is similar to that of the iPSC/2. Only the differences between the two models are highlighted here.

The major difference between the communication protocol on the DELTA and that on the iPSC/2, is that on the DELTA messages get sent packet by packet rather than as a whole done on the iPSC/2. Because of this, the DELTA adopts one communication protocol for long and short messages rather than two on the iPSC/2.

Also, the use of the free pool and reserved pool on the DELTA differs slightly from that on the iPSC/2. On the DELTA, a packet is saved in the reserved pool only when the free pool is full. Whilst the free pool on the DELTA provides temporary storage for all message sizes, the iPSC/2's free pool is used only for long messages.

As with the iPSC/2, the routing algorithm on the DELTA is deterministic. On the DELTA, a packet moves horizontally until it reaches the destination's column, then it moves vertically to its destination. Compare this to the e-cube routing algorithm used on the iPSC/2.

As with the iPSC/2, the communication module for the DELTA starts by initializing the communication network channels, the system buffers and other data needed by the model. Since both systems keep a free and reserved pool, `Delta_Communications_init()` performs the same steps performed by `iPSC_Communications_init()`. But, due to the fact that the Delta's architecture is different from that of the iPSC/2, step 1 in `iPSC_Communications_init()` has a different meaning. Step 1 in `Delta_Communications_init()` initializes the Delta's 2-dimension mesh. It creates for each node in the mesh 4 links which connect it to its neighbors (up, down, left and right). Even though the nodes at the edges of the mesh have only three connections and the nodes at the four corners of the mesh have only 2 links, the extra links will not be used.

After creating all the links in the mesh, step 1 labels them as not busy and not in_use the same way we labeled the channels for the iPSC/2. This labeling indicates that these links are not in_use (do not have a probe on them) at the moment.

As with the iPSC/2, let's follow a message starting from when the send was issued until the message is received and highlight the differences between the iPSC/2 and the DELTA communication models. When a `csend()` function call is issued by a processor, `Delta_Csend()` handler gets called which performs the following steps:

```
Delta_Csend()
{
    1-set send type to 1
    2-create a probe for the message
    3-break the message into packets and place them in the probe
    4-save the probe and the processor's time in a temporary communication
      data.
    5-replace the calling processor data and handler by the temporary
      communication data and Delta_send_cont().
}
```

The role of steps 1, 2, 4 and 5 above are the same as their corresponding steps in `iPSC_Csend()` explained in the previous section. Step 3 breaks the message into 512 bytes packets and places these packets into the probe. The probe now has to keep track of the number of packets in the message, which have been sent, and which still have to be sent.

As with the iPSC/2, the send operation can be performed by an `isend()` function call which gets evaluated by calling `Delta_Isend()` handler. Again, the steps performed by `Delta_Isend()` are the same as the steps performed by `iPSC_Isend()` except in the step that breaks the message into packets. The reader is referred to the discussion presented for `iPSC_Isend()` in the previous section for more details.

Lets see how the `Delta_send_cont()` handler continues the evaluation of both `csend()` and `isend()` operations. When the `Delta_send_cont()` handler gets called by the processes scheduler it performs the following steps:

```
Delta_send_cont( )
{
    1-if did not get space (flag = 0), then get space and set flag to 1.
    2-if did get space, but did not get first channel (flag = 1), then get first
      channel and set flag to 2 and activate the communication network
      handler
    3-set the process's time to INFINITY, add it to the blocked_process array.
}
```

The `Delta_send_cont()` routine is much shorter than `iPSC_send_cont()`. The reason behind this is that the DELTA uses only one communication protocol for both short and long messages while the iPSC/2 uses one protocol for short messages and one for long messages. Notice also, the steps performed by `Delta_send_cont()` are exactly the same as the steps performed by case 1 in `iPSC_send_cont()`.

After `Delta_send_cont()` places a message's probe on the first link towards its destination, the responsibility of moving it to its destination is passed to the communication network handler. The communication handler is called `Handle_Delta_Communications()` for the DELTA. Because it performs exactly the same steps performed by `Handle_iPSC_Communications()` it will not be discussed here. Needless to say, when the probe reaches its destination node `Delta_packet_deliver()` handler is called instead of `iPSC_message_deliver()`.

When a probe reaches its destination node i.e. has controlled all the links required to transmit the packet, `Delta_packet_deliver()` is called to handle the packet transmission. It perform the following steps:

Delta_packet_deliver()

```
{
    1-set the first link's time.
    2-label the first link as not busy and not in_use.
    3-free all the used links and unblock all the blocked probes.
    4-remove the probe from the last link and place the packet in
      the destination buffer. Move to the free pool if possible
    5-unblock all the processes that were blocked because of a message
      passing
      operation and activate the communication network handler
      case 1: the packet was last packet of the message
        set message receive time = to the last channel's time +
        channel latency if the send was ISEND,
        msg_id array[id]=TRUE;msg_id_time[id]=time;
        restore the sending process and return
      case 2: the packet was not last packet of the message
        set flag to 0 and return
}
```

Steps 1, 2, 3, and 5 above perform exactly the same functions performed by their corresponding steps in `iPSC_message_deliver()` keeping in mind that we are now dealing with a mesh architecture rather than a hypercube.

The Delta packets get delivered to their application buffer if a receive was posted to the message when the matching send started. Otherwise, they are delivered to the reserved. If the free pool had space when a packet is delivered to the reserved pool, the packet is moved to the free pool. Moving a packet from the reserved pool to the free pool requires returning the space that the packet occupied back to the reserved pool and decrementing the free pool's available space by one packet. Also, since the message is sent packet by packet, some information is kept in the packets to enable reconstruction of the message.

What is left in the role of `Delta_packet_deliver()` in the communication model depends on if the packet just delivered was the last packet of the message or not. If the packet was not the last packet of the message that need to be delivered, case 2 above changes the flag value to zero so `Delta_send_cont()` handler will start looking for a space for the next packet of the message. On the other hand, if the packet just delivered was the last packet of the message (case 1), the message's receive time is set. Also, if the send command was done asynchronously (`isend()`), an indicating of the `isend()` completion is placed in the `msg_id` array. Finally, the message's receive time is placed in `msg_id_time` array to indicate when the `isend()` was completed.

The Delta's `irecv()`, `crecv()`, `msgdone()` and `msgwait()` are evaluated exactly in the same way their corresponding function calls were evaluated in the `iPSC/2` communication model so they are not repeated here.

The reader may notice that the Delta's communication model was easily obtained by a slight modification of the communication model for the iPSC/2. Now we will see how the communication model evaluates message passing operations on the CM-5.

CM-5

Although the communication protocol on the CM-5 is completely different from that of the DELTA, the CM-5 communication model is easily obtained by a slight modification of the communication model for the DELTA.

The communication network (the Data Network) on the CM-5 does not guarantee the order of delivery of each message's packet (even from single source to single destination). This is because packets traveling up (to least common ancestor level) make random choices among available channels. Finally, on the CM-5, there is no lock-down path created from the message's source to its destination.

Needless to say, the handler that moves the probe from one channel to another channel on the Delta's mesh must also be modified to move it on the fat tree instead. Also, the CM-5 routing algorithm must be kept in mind when moving a probe from one channel to another. Remember, packets traveling up (to least common ancestor level) make random choices among available channels while when traveling down they have a deterministic path.

Recall that the CM-5 does not allow messages to be transmitted before their matching receive is posted. This simplifies the communication model for the CM-5 since the reserved and free pool sizes and messages manipulation is not needed any more.

As with the DELTA, the communication module for the CM-5 starts by initializing the communication network channels and other data needed by the model. When called CM5_Communications_init() performs the following steps:

```
CM5_Communications_init()
{
    1-initialize the communication network channels
    2-initialize the recv_posted_msgs array to NULL
    3-initialize the recv_posted array to not_posted
    4-initialize the msg_id array to FALSE
    5-initialize the msg_id_time array to INFINITY
    6-initialize the message id counter to 0
    7-initialize the blocked_process array to FALSE
}
```

CM5_Communications_init() performs fewer steps than the Delta_Communications_init(). This is because the CM-5 does not keep a free and a reserved pool like the DELTA so

they are not initialized. Also notice that only step 1 above will require a different explanation from that on the DELTA.

Step 1 above built the CM-5 fat tree with the required number of nodes. The fat tree nodes are then given a number and a type. Each node is given a type that corresponds to its level in the fat tree. For example, the leaf nodes are given a type 0. Notice that type 0 nodes have 0 down links and 2 up links (see Figure 9). Types 1 and 2 nodes have 4 down links and 2 up links and types 3 and above nodes have 4 down links and 4 up links. As with the DELTA channels, all the fat tree's links are labeled as not busy and not in_use and given a 0 as their available time when created.

As with the DELTA, let's follow a message starting from when the send started until the message is received and highlight the differences between the DELTA and the CM-5 communication models.

Because CM5_Csend() and CM5_Isend() handlers perform the same steps performed by Delta_Csend() and Delta_Isend() handlers we omit repeating them here and focus our discussion on CM5_send_cont() handler.

```
CM5_send_cont()
{
    1-if the matching receive is not posted ( flag = 0)
        set the calling process's time to INFINITY and return
    2-if the matching receive is posted (flag = 1)
        place one packet on the communication network and
        increment the process time by the time to do that
    3-if the packet was the last packet in the message
        restore the process data and handler
        if the send was isend()
            set msg_id array[id] to TRUE
            and msg_id_time[id]=time
    4-else return
}
```

The major difference between the CM5_send_cont() and the Delta_send_cont() come from the fact that the CM-5 communication protocol does not lock all the links from the message's source to its destination before placing a packet on the first link towards its destination. In other words, the sending process does not wait until a packet is received by the destination node before sending the next packet.

Also, on the CM-5 a space is available to receive the message only when the message's corresponding receive is posted by the receiving node. This is because the CM-5 does not allow message transmission to start until the receiver has issued its corresponding receive. So, step 1 above searches the rcv_posted array to see if the message's destination node has issued a receive for this message or not. If the matching receive was found, step 2 tries

to place one packet on the communication network. Otherwise, step 1 sets the sending process's time to INFINITY to block it until the required receive is posted.

As explained earlier, the CM-5 communication protocol first moves the packet up the tree until it reaches the desired height, then it moves it down to the destination. Because each leaf node has two up links, packets are placed on the link with the lowest time that is not busy or in_use (step 2).

If the packet was the last packet of the message, the send operation is complete and CM5_send_cont() is not needed anymore. The sending processor's data and handler get restored so it can advance to the next instruction. Recall from the discussion on the DELTA communication model that restoring the processor's data and handler is done only for csend() function call evaluation. If the send was done using an isend(), the process that was created to complete the send operation is removed from the process scheduler when CM5_send_cont() is completed.

Having placed a packet probe on the first channel towards its destination, the responsibility of moving it to its destination is passed to the communication network. The communication network handler for the CM-5 is called Handle_CM5_Communications(). It performs the same steps performed by Handle_Delta_Communications() with only one exception. On the CM-5 when a probe is moved from channel A to channel B, channel A is freed by labeling it as not busy and not in_use and channel A is labeled as busy and in_use. This is because the CM-5 does not lock all the channels (links) from the packet's source to its destination.

When a packet reaches its destination node, CM5_packet_deliver() is called which performs the following steps:

```
CM5_message_deliver()
{
    1- remove the probe from the last channel and place the packet in the
    destination buffer.
    2- if the packet was the last packet in the message
        set message receive time = to the last channel's time + channel
        latency
}
```

Step 1 removes the probe from the last channel and places the packet it carries in its destination buffer. Removing the probe from a channel, the channel gets freed by labeling it as not busy and not in_use. If the packet was the last packet in the message, the message's receive time is set in step 2.

At the receiving end, the CM-5 communication model performs the same steps performed by the DELTA communication model. In other words, the CM-5 communication model

evaluates the receive operations the same way the DELTA does. The reader is referred to the discussion presented in the previous section for more details.

Performance Summary

PST has successfully illustrated several aspects of parallel systems. Cache effect have been witnessed in several experiments, shared memory "bus" contention and deadlock. have all been seen.

With the Sequent architecture using only one processor, cache effects can be witnessed in simple image processing applications. Specifically, comparing an image processing routine that ensures processed regions fall evenly on memory pages with one that does not shows a noticeable performance difference. When page size and the mapping of an image onto memory pages are not considered, there is a loss in performance. In addition, when the same program is adapted to a multiple processor algorithm, cache effects become even more important since any overlapping memory that one processor uses may affect another processor's access to data in the same block. An ideal four processor algorithm would yield a system that consumes only one quarter the amount of time a non-parallel algorithm consumes. When cache effects are not considered, the parallel version often approached the same execution time as the non parallel system.

The cache size and block size both affect system performance based on how long and how frequently blocks must be swapped. For programs that use a small set of data that are spread out in memory, large block size means that a large amount of time is spent reading memory that will never be used. In these cases a smaller cache block size is desirable. On the other hand, if the block size is too small, the cache will constantly be swapping.

The CLIP model has successfully illustrated the importance of locking bitplanes. In small examples, locking bitplanes increases execution time, while other, larger examples have shown an improvement in performance when the correct bitplanes are locked. In addition, the difficulty associated with manipulating an entire image is apparent by inspecting the code used to implement simple operations such as convolution.

Communications effects have also been observed on the CM-5, Delta and iPSC. On the CM-5, when csend and crecv are not exactly matched, PST reports an INFINITE execution time, as it should since the CM-5 requires a matching crecv before csend can complete. The other models only report INFINITE execution times when a crecv is not matched with a csend. The execution times for these models all show a proper dependence on the times messages are sent and received. That is, when one processor posts a crecv early, and a matching csend occurs much later, the receiving processor does indeed wait for the receipt of the entire message before it continues its processing.

The communication model for the iPSC/2 was coded and tested to prove the theoretical predictions. The effect of varying channel latency, channel bandwidth, message size and

path length on the message latency were fully investigated and found to agree with the theoretical predictions. Similar experiments were also found to agree with theory for the Delta and CM-5 machines.

Future Work

It is proposed that in the future the tool will be extended further. In particular, the design and implementation of PST enables us to consider several visualization techniques which could be added to the tool. For example, page swapping or message passing could be shown physically on the screen as the application is simulated, making it even easier for a novice to appreciate the problems and issues associated with parallel computing. Other additions to the tool will include:

- Completing a full set of evaluations on PST in terms of its performance and functionality
- Testing the tool with the students at Clarkson University enrolled in the Parallel and Distributed systems graduate course. In particular, students would be encouraged to develop tutorials for the system.
- Add more parallel architectures to the tool. MasPar and pipeline architectures would be candidates for this.
- Add some more higher-level programming constructs to the architectures already modeled.

References

- [1] Nugent, S. 'The iPSC/2 Direct-Connect Communication Technology'. Conference on Hypercube Concurrent Computers and Applications, 1988, 3, pp 51-60.
- [2] Littlefield, R. 'Characterizing and Tuning Communications Performance on the Touchstone DELTA and iPSC/860'. Intel Scientific Computers pp 1-5.
- [3] Pierce, P. 'The NX/2 Operating System'. Conference on Hypercube Concurrent Computers and Applications, 1988, 3, pp 348-390.
- [4] 'Concurrent Supercomputer Consortium Proceedings of the First Delta New User Training Class Notes'. Edited by M. Maloney & P. Olsen. CCSF-24-92 ; July 1992.

[5] 'Concurrent Supercomputer Consortium
Proceedings of the First Intel Delta Application Workshop'. Edited
by Tina Mihaly and Paul Messina. CCSF-14-92 ; February 1992.

[6] 'Concurrent Supercomputer Consortium
Proceedings of the Delta Advanced User Training Class Notes'.
Edited by M. Maloney & P. Olsen. CCSF-25-92 ; July 1992.

[7] 'Connection Machine CM-5 Technical Summary'. Thinking Machines Corporation
Cambridge, Massachusetts. November 1992.

[8] iPSC/860 System Manuals

iPSC/860 System Technical Documentation Guide

iPSC/860 C System Calls Reference Manual

iPSC/860 System User's Guide

iPSC/2 and iPSC/860 C Language Reference Manual

Appendices

1. List of Other Documentation

User Manual

Programmers Guide

PST Programmer's Manual

The Programmer's Manual will describe how the PST source files are organized, how the makefile is arranged and how support for new features can be added.

File Structure and Installation

The file PST.01.src.tar.Z is the complete source tar'ed and compressed into one file. Use "zcat PST.01.src.tar.Z | tar -xvf -" to expand the file. This command will create a directory "PST", which is the root of the PST source structure.

File Structure

The file organization is outlined in the following tree:

PST -----	README	- brief description of file structure
-----	Makefile	- "make all" makes executable "make clean" removes temporary files
-----	*.ACH	- example architecture files
-----	*.RAP	- example application files
-----	UIM	- User Interface Module source
-----	AM	- Architecture Module source
-----	LM	- Language Module source
-----	EM	- Evaluation Module source and language module token handlers
-----	sequent	- sequent model source
-----	clip	- clip model source
-----	ipsc	- iPSC (hypercube) model source
-----	delta	- delta (2-d mesh) model source
\-----	cm5	- CM-5 (fat tree) model source

Makefile

The file "Makefile" has the following rules:

- make all - builds the entire program, results in the executable "PST" (see paragraph below)
- make clean - removes all the intermediate files
- make scour - removes all files created by "make all" including any executables
- make EMtest -make a command-line version of PST that is intended for development purposes

The Makefile uses the CC variable to define the compiler make uses. PST was developed on an ANSI compatible compiler, and as such, requires an ANSI compatible compiler. One **warning** about the makefile: **the makefile has no explicit dependencies on header files**. For this reason, a change on a header file used by more than one source file should be followed by a "make clean all" to rebuild the entire system. There are specific rules in the makefile to generate object code from C source (.c.o rule), Lex source (.l.o) and Yacc source (.y.o). The Lex and Yacc rules use intermediate "PREFIX" files that are used to allow renaming of appropriate functions in order to allow multiple parsers to be linked together.

There is a list of directories assigned to variables that are used in the makefile. If a new directory is added, another entry should be added. If there are include files, the directory can be added to the INCLUDES variable, and the object file can be added to OBJECTFILES. There is no dependency on header files, so if any header file is changed, a "make clean" should be done before continuing.

UIM

The UIM directory contains all the source for the User Interface Module (UIM). This source is written to run under X-windows and relies on the Motif widget set

The user interface module is divided into several key C files: uim-main.c, record.c, graph.c, and buttons.c. All of these files include uim-main.h, the header file that includes all of the necessary X-Window files, and defines our widely-used NewRecStruct structure. The uim-main.c file sets up the main window, initializes global variables, and contains the few functions that aren't separated into the other .c files (such as Tutorial). The record.c file contains all functions pertaining to the creation and modification of a record; NewRecord() creates a new record widget, initializes a unique NewRecStruct, and sets up the callbacks for all of its buttons. graph.c contains the functions pertaining to the

graph module. Lastly, buttons.c defines the functions that insert buttons specific to a particular architecture into the record widget.

NewRecStruct is a structure that contains all of the widgets and variables inherent to each individual instance of a record. When NewRecord() is called during a callback function, it creates a new NewRecStruct (nrs is a pointer to a NewRecStruct that gets passed to nearly every callback). When nrs is passed to callback functions, w usually holds the parent record widget defined in NewRecord. temp_widget is widget, set to 0 initially; it points to widgets that are to be destroyed at the end of a callback. For example, when you pop up a file selection dialog to choose an architecture, temp_widget gets set to the FileSelection widget you created; when the user presses OK or Cancel on the dialog, XtDestroyWidget() is called on temp_widget. rec is a pointer to a RecordTable structure (defined in the simulator code). This struct gets updated every time the user picks a new architecture or application. After a run, several values of rec return performance information (such as cache hits, cache misses, and total time). The rest of NewRecStruct contains variables that determine whether or not windows are opened or closed, label widgets that get updated, etc.

Most of our window layouts are straightforward, with the possible exception of the Record window. This window begins with a form-shell, which contains a main-window widget, which contains a paned-window widget with four panes. The main-window widget is used for the menu bar. The first (top) pane has a form, inside of which are the buttons and labels that pertain to any architecture. The second pane has a form, with nothing in it (initially). When a Sequent architecture file is selected, the SequentButtons() function is called, which puts a second form inside of the first, and fills that form with buttons specific to the Sequent architecture. When a different architecture is selected, XtDestroyWidget() is called on the second form, thereby allowing forms to be inserted into the first form later on. The third pane contains a form, in which there are "speed buttons" that merely duplicate the functions found on the menu bar. Lastly, the fourth pane contains a form and several labels that report the results of a simulation run.

When the Graph button is pressed it pops-up the Graph window, and creates a new NewRecStruct with all of the setting of the original passed in. The Graph window consists of a form widget with a paned-window widget of two panes. The top pane contains a form which has three drawing area widgets in it: the left and bottom rulers, along with the main graphing area. It also holds all of the labels showing the current settings of the module. The second pane consists of a form with three buttons in it. The Change button calls up the menu by which the settings can be changed, and the Run button calls the GraphRun() function. This function runs the simulation "Number Of Runs" times. It does not reset anything in, nor affect in any way, the Record window because of the new NewRecStruct

created at the start of the Graph module. Note that the GraphModule won't run if you don't have an architecture and application selected. Error checking may be added, but is non-existent as of right now.

How to add architecture-specific buttons

The buttons.c file contains all of the functions pertaining to specific architectures. The function names are prefixed with the name of the architecture, followed by the word Buttons. (Ex.: SequentButtons, IPSCButtons, etc.) You can mostly cut-and-paste the widget set-up code found in SequentButtons; you'll probably need to change the names of the buttons and labels (such as GlobalMemoryLabel), and add the new ones into the NewRecStruct definition. The callback functions are similar - you can cut-and-paste them for the most part, with changes needed to be made to variable identifiers, and to whatever "control" info that differs. SequentButtons() is a good example to look at.

AM

The Architecture Module (AM) directory contains Lex (IAM.l) and Yacc (yAM.y) source. These files are fairly straightforward. The rules for recognizing specific tokens are very explicit. The parameter names are all Lex/Yacc tokens. The memory units and time units are stored in a table to allow easy modification. In yAM.y there is a function ParseArchitecture, which parses the file specified in the Record_Table that is passed to it. Many parameters are checked for validity and various architecture specific parameters are computed. The main outcome of ParseArchitecture is that the Record_Table entries are filled in. No new data is created.

One difficulty in the AM is the fact that PST has two parsers, one for the Language Module and one for the Architecture Module. The AM parser has been modified to have a prefix "AM_PST". This is accomplished through a script file "gnPREEFIX" which creates a prefix file that gets used by the Lex and Yacc makefile rules to change the names of functions that would otherwise be multiply defined.

LM

The Language Module (LM) directory contains Lex (IRAP.l) and Yacc (yRAP.y) source. These files use many tables to make implementing the C-like RAP parser easier. The Lex source recognizes very few tokens. For example, variable names and function names are just considered names, and whether or not they are defined is determined when parsing is complete.

Most of the action in the LM parser is in the Yacc source. If a new machine is added to PST, a search should be made for existing machines to see where the new machine needs to be added. There are several switch statements that make the parser behave differently based on what application type it is intended for. For example, the Sequent has global memory, so global variables have to be allowed, CLIP has image primitives that PST will evaluate. There is a section that determined what types are legal in various machines.

Before a new file is parsed, the global tables that the LM parser uses are cleaned up. Then the LM parses the file. Finally, the resulting tables are checked for errors and then copied into the current record table.

EM

The Evaluation Module (EM) constructs the model of the current system, then evaluates it by executing the parsed code, calling on the architecture models as needed. When a new machine is added, a search should be made in EM.c on an existing machine type to see where the new machine needs to be added. Again, switch statements are used to allow machine specific routines to be called to construct the system model and report performance.

For performance reasons, tokens (token_types) that the LM generates are consecutive so that the functions associated with them can be stored in an array of function pointers which can be indexed by a simple subtraction of the first from the desired one. If any changes to the language are desired, is new tokens are added, there must be token handlers added as well. It is imperative that these tables be kept current. The files EM.c, yRAP.y, and record.h all need to be updated if any new token handlers are added. When adding new tokens, always add them to the end, and always include the number of the token relative to the first (REDUCE) token, as all the previous tokens are.

Models

A modular approach was desired for all the models. The memory models are created in such a way that new memory models can easily be added, or new arrangements of memory can be tried. For example, a memory can be directly connected to a processor, or a cache can be inserted, or even two caches, one of which might be used to imitate register optimization or virtual memory support. These models are designed so that they are all called in exactly the same manner.

The modeling of all concurrent systems, such as multiple processors, and communications networks, were designed in a similar manner. The EM scheduler just calls a process handler, without regard for what the process is.

The handler takes care of it's job, whether it is to update the communications network, or allow a processor to evaluate more code.

Machine Files

"sequent.c", "CLIP.c", "ipsc.c", "delta.c" and "cm5.c" are all the most important files in regards to defining a new machine. These files contain the routines that construct the machine model. If a machine has a communications model, one of these existing models can be used, or a new one can be added. If a new machine is added, any models that are unique to it can be added in it's file. In general, the format of the existing files should be followed.

The CLIP model, for example, has bit-plane models, and is a SIMD machine, which means that only one processor is modeled, and image types are used. Since images are unique to CLIP and affect the Language Module, and bitplanes are a type of memory, the LM parser had to be modified to support images, and only in CLIP mode. The file "memories.c" also had to be modified to add support for image types.

The Sequent model does not have a communications network, but does use shared memory, so the LM has code to handle shared and local memory variables. If a new machine were to be added that had shared memory and a communications network, it would be simple to add a communications network the existing sequent model by enabling the communications system calls (csend, crecv, etc.) in the new machine, and either copying the communications source from an existing file, making global the existing communications code or writing a new communications model.

Adding an Architecture

The following is a list of existing features that new architectures can take advantage of with little (removing "static" or block copying code) or no change:

- | | |
|----------------|---|
| Memories | - FIFO WT cache, shared memory and local memory |
| CLIP | - CLIP bit memory routines |
| Communications | - The iPSC (hypercube), Delta (2-d mesh), CM-5 (fat tree) networks |
| Language | - The language parser supports local and shared class variables, as well as image types |

To add a new architecture, a general understanding of how the current system works is essential. To add a new machine, an old machine that is closest to the

new one should be copied first. The new machine should be exactly the same as the existing one. Once this copied system is functioning properly, and a general understanding of how the system works has been developed, the new model can be modified to match the desired architecture.

It is suggested that modifications be made in the following order: 1) add new architecture parameters to the AM (may involve adding parameters to the Record_Table in record.h. See warning above), 2.) modify LM to recognize new language constructs, keeping in mind that these constructs may not be desirable in other language modes, and should be invalid in other modes, 3) develop new token handlers to implement the new language constructs (make sure that the programs execute, properly, without the new architecture models. i.e. ensure the language processing is correct), 4) develop any new architecture models and 5)integrate all the changes by now allowing the token handlers to call on these new models.

Contents

Contents	
Introduction	
System Requirements	
Terms and Features	
Screen Layout	
Quick Start	
User Interface	
General Record Information	
Graph Module	
Tutorial	
Reference Guide	
Language Guide	
General Features	
RAP	
CM5	
Message Passing Protocol	
CLIP	
DELTA	
Message Passing Protocol	
Sequent	
Hypercube	
Message Passing Protocol	
Architecture	
Architecture Module	
Supported Machines	
Delta Architecture	
Sequent Architecture	
CM5	
CLIP	
iPSC Hypercube	
Software Module	
Supported Modes	
RAP	
Sequent	
CM5, Delta and iPSC Hypercube	
CLIP	
Sample Files	

Introduction

PST is a tool designed to illustrate how different algorithms and applications perform on different parallel computer systems as well as to point out where bottlenecks and system slow-downs occur. PST allows the user to interactively change the architecture parameters as well as to graph the performance of an algorithm over a range of parameter values.

System Requirements

PST was developed in the UNIX environment in C and requires X-Windows and the Motif tool kit.

Terms and Features

PST has three main tools:

1. **Record Windows** - This is where the user selects an application and architecture specification. System evaluation for a specific architecture/application combination and relevant tutorial information will also be shown in this window.
2. **Graph Windows** - This is where the user can select which performance parameter to graph and which parameter to vary. Multiple graphs are supported to allow the user to compare either different machines or the same machine with a different parameter varied.
3. **Tutorial Windows** - This window will allow the user to select tutorial files which walk the user through sample sessions and point out strengths and weaknesses of different architecture and application combinations.

Screen Layout

When PST is started, a simple vertical menu will appear on the screen. This is the Main Window. From here you can invoke New Record, Tutorial, Quit. Each option pops up a new window.

To activate or select an option, simply click on it by positioning the mouse over the button then pressing and releasing the left mouse button. Along the top of the windows are menus. These buttons invoke pull-down menus which list some additional options.

Quick Start

Start PST by executing the main program PST. A window will pop up with a simple vertical menu. Select New Record. This will cause a new Record Window to appear.

Now that we have a Record Window open, click the Arch button in the Record Window. This action will cause a list of architecture files (.ACH files). Select the entry that reads Seq1.ACH. This will cause PST to parse the Seq1.ACH file into memory. If there are any errors, they are printed to the standard error device, and a dialog window warning that the file was invalid will appear. Since this file is correct, the title bar of the Record Window will now contain the label Sequent, the selected architecture will show the Seq1.ACH file name and memory parameter buttons will appear.

To select an applications, click on the App button. A file selection window will appear as before, but this time application source (.RAP) files will be listed. Select Seq1.RAP. As with the architecture file, the selected file will be parsed into memory. If there are any errors, they are printed to the standard error device, and a dialog window warning that the file was invalid will appear. Since this file is also correct, the selected application line will contain the "Seq1.RAP" file name.

Any of the visible architecture parameters can be changed and examined. The bottom of the Record Window shows several measures of performance. These values are updated when the Run button is pressed.

User Interface

When PST is started, you will see a menu with three choices: **New Record**, **Tutorial**, and **Quit PST**. Each click on New Record will bring up a new record window. The record window controls how each individual simulation will run. Tutorial provides on-line help, stored in the form of text files. Quit PST will end all parts of the simulator and exit to the operating system.

General Record Information

Pressing New Record causes a Record Window to pop up (see Figure 1). The top of the window contains a menu bar with two pull-down menus: **File** and **Debug**. File provides **Architecture**, **Application**, **Run**, **Graph**, and **Close** buttons. Architecture brings up a file selection window, allowing you to choose an architecture to simulate. Architecture files must end with .ACH. Similarly, Application allows you to choose a program to run on the current architecture. Applications must end with .RAP. Run will simulate the selected application on the selected architecture. Run will not work if you

have not selected one (or both) of these, nor if the application and architecture are of incompatible types. Graph calls up the corresponding graph module for this particular record. Close will get rid of the record, and all corresponding windows. The debug pull-down menu allows you to turn debugging on or off. When on, debug information is sent to standard error.

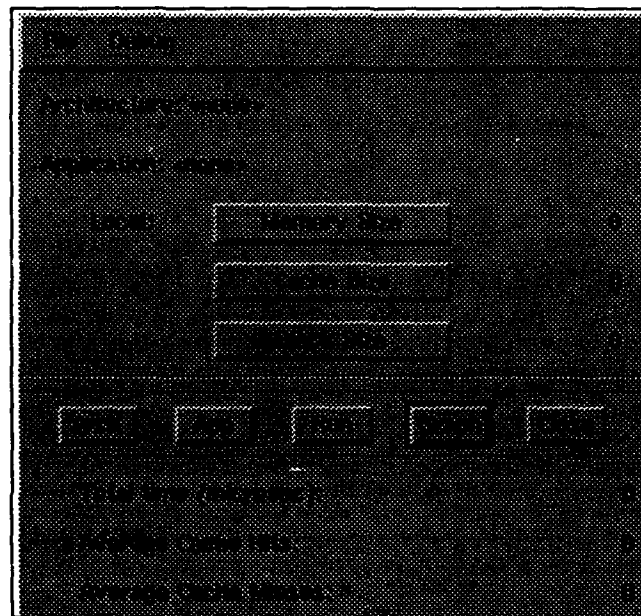


Figure 1

The upper pane of the window contains the architecture file name and the application file name, along with three buttons controlling local memory options, **Memory Size**, **Cache Size**, and **Block Size**. The only limitations on the values are: block size must divide evenly into cache size, and cache size must divide evenly into memory size. When applicable, there will be global memory buttons with the same restrictions.

The middle pane of the window contains quick buttons having the same effect as the buttons on the File menu of the menu bar. The last pane provides the results of the simulation: **Total Time** (in micro-seconds), **Average Cache Hits**, and **Average Cache Misses**.

Graph Module

The graph module will allow the user to run the simulation multiple times by changing one parameter and graphing the result against another. The upper pane of the window contains the area where the graph will appear, along with labels to tell the user what the graph means (see Figure 2). The lower pane contains three buttons: **Change**, **Run**,

and Close.

The Change button pops up a menu of buttons (see Figure 3) used to modify the Graph Module: **X Label**, **Y Label**, **Runs**, **X Start Value**, **Y Start Value**, **X Increment**, **Y Maximum**, and **Close**. XLabel sets the x axis parameter. This will be the value which will be varied. Y Label sets the y axis parameter, which will be compared against the x axis. Runs sets the number of simulations the graph will show. X Start Value lets the user change the value to start the simulation's first x value. Y Start Value is the lowest value recorded on the y axis of the graph. X Increment is the amount the x axis parameter will increment between each test run. Y Maximum is the largest value recorded on the y axis of the graph. Excluding X Label and Y Label, a window with editable text will pop up if one of these buttons is pushed. Click in the text window, and type the new value desired. Then press OK for the value to change in the Graph Module. Lastly, Close will close the button menu.

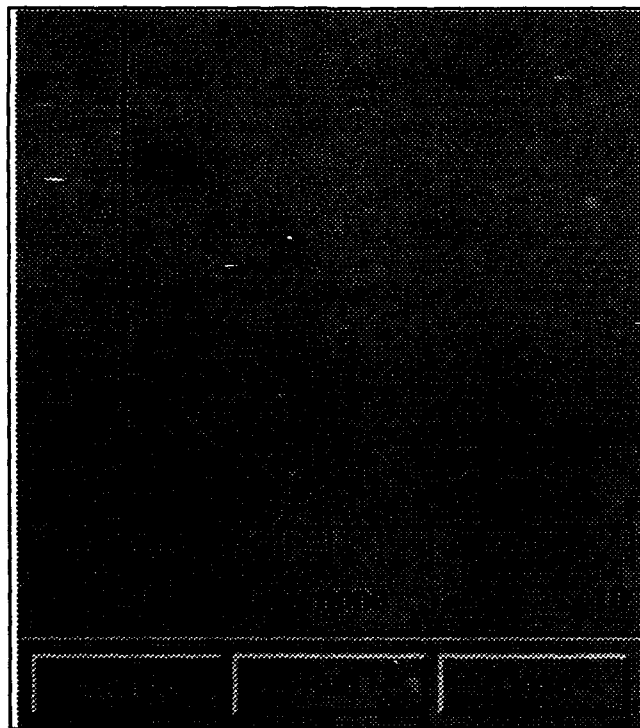


Figure 2

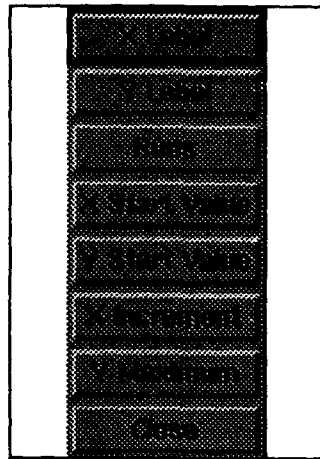


Figure 3

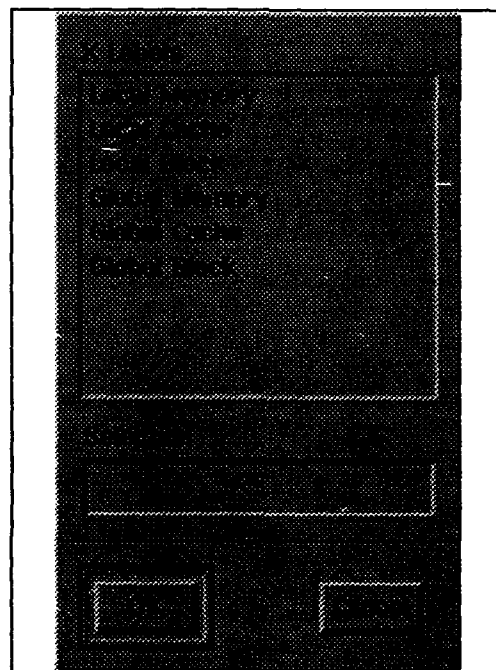


Figure 4

When the X Label or Y Label button is pressed a selection window will appear (see Figure 4), from which there are limited choices. Simply click on one of the choices twice, or click on it once and press the OK button. **Warning:** There is very little error checking in this section. Invalid runs will not be skipped.

The Run button of the Graph Module will run the simulation, and graph the results in the upper pane. The results of the Graph Module will not be posted on the Record Window. The Close button will close the Graph Module window.

Tutorial

The tutorial button in the main window allows you to have one or more help files present on the desktop while you are running a simulation. Simply click on Tutorial to bring up a new one. From the file selection list, pick a file that corresponds to the topic of interest. A window with the tutorial text will appear. You can resize the window and scroll through the text using the horizontal and vertical scrollbars, or close the window by pressing the close button. The tutorials are self-explanatory.

Reference Guide

- | | |
|------------------------------|--|
| Create a new Record - | Click on the New Record button in the Main Window. |
| Change Application - | In the Record Window, press the App button. Next, choose an application from the list of stored application files. |
| Change Architecture - | In the Record Window, press the Arch button. Next, choose an architecture from the list of stored architectures files. |
| Open a new Graph - | In the Record Window, click on the Graph button. Click on the Change button. This will pop up a list of parameters that need to be set before the Run button is pressed in the Graph Window. |
| View Tutorial - | To start the tutorial, click on the Tutorial button in the Main Window. |

Language Guide

To write a RAP program (file name must be of form *fname.RAP*), the user must first decide which language mode is to be used. To specify the mode, the first non-comment in the file must be "language = *language-mode*". For example:

```
/* First define language mode */  
#language CM5
```

This makes RAP interpret some supported CM5 functions, csend, crecv, etc.

Once a language mode has been chosen, the program can be written. The following is a brief description of features supported in all modes, followed by a description of each mode and its unique features.

General Features

All modes are "C-like". The semicolon (";") is required between statements.

Strings are supported.

+, -, /, * and % (remainder) are supported.

& (bitwise AND), | (bitwise OR) and ^ (bitwise XOR) are supported.

&& (Logical AND) and || (Logical OR) are supported.

<, >, <=, >=, ==, != are supported.

= (assignment) is supported.

int and integer arrays are the only general data types available.

image is only available in CLIP mode.

function calls are the same as for C. To call fred with a parameter x, use "fred(x)"

if, if .. else, and for are supported as in C.

break, continue, return, and return *expression* are supported.

RAP

The PST source code is RAP code. There are several modes for RAP code. These modes are used by PST to parse the source code correctly based on the type of machine the code is intended for.

CM5

CM-5 only supports local non-shared variables. The communications system calls are supported as listed below.

Message Passing Protocol

csend - Blocks until a corresponding **crecv** has been posted and the message sent.

crecv - blocks until a message is received

isend - does not block sending processor, message is sent while program executes

irecv - does not block, message is received while program executes

msgwait - wait for completion of message

msgdone - returns 0 if message send/receive is not complete, non-zero otherwise

my_pid() - logical node number

numnodes() - number of nodes in program

CLIP

The CLIP supports the standard integer and integer arrays. The CLIP architecture only operates on image types, so image types are also supported:

`image x[length][width][height];`

This declares variable `x` to be an image of `length` bitplanes, of size `width` by `height`. The CLIP supports many of the primitive system calls:

LDA - load bit register A with the given bitplane
LDB - load bit register B with the given bitplane
PST_local - execute a local bit process, store result in specified bitplane
PST_pointwise - execute a pointwise bit process, store result in specified bitplane
im_lock - mark a given image as locked
im_unlock - mark an image as unlocked

DELTA

Delta only supports local non-shared variables. The communications system calls are supported as listed below.

Message Passing Protocol

csend - Blocks until message has been completely delivered to the communications network

crecv - blocks until a message is received

isend - does not block sending processor, message is sent while program executes

irecv - does not block, message is received while program executes

msgwait - wait for completion of message

msgdone - returns 0 if message send/receive is not complete, non-zero otherwise

my_pid() - logical node number in partition

numnodes() - number of nodes in partition

Sequent

The Sequent has shared memory, so allows local and shared variables. All global variables are treated as shared. Additionally, the Sequent has a library of routines used to implement parallel programming:

m_fork - create a new process in parallel

my_pid() - return process id

numnodes() - return number of child processes

Hypercube

Hypercube only supports local non-shared variables. The communications system calls are supported as listed below.

Message Passing Protocol

csend(), crecv() - provides message buffering:

csend - Blocks until message has been completely delivered to the communications network

crecv - blocks until a message is received

isend - does not block sending processor, message is sent while program executes

irecv - does not block, message is received while program executes

msgwait - wait for completion of message

msgdone - returns 0 if message send/receive is not complete, non-zero otherwise

my_pid() - logical node number

numnodes() - number of nodes in program

Architecture

To write an architecture file (file name must be of form *fname.ACH*), the user must first decide which machine is to be defined. To specify the machine, the first non-comment in the file must be "architecture = *machine-name*". For example:

```
/* First define architecture */  
architecture = Delta
```

Supported *machine-names* are iPSC, Delta, CLIP, Sequent and CM5.

All architecture parameters are specified by equating the parameter name with the numbers necessary to specify the parameter. Most common units are supported for given parameters. For example:

```
main_memory = 16 MBytes
```

This specifies that there is 16MBytes of main memory.

Some parameters are defined by type and operation. The following example defines the time required for image addition on the CLIP:

```
load = 8 cycles  
PST = 10 cycles  
SST = 12 cycles
```

Architecture Module

The architecture module is responsible for interpreting architecture files (*fname.ACH*) for simulation. There are several machines ready for simulation, including the CM5, Sequent, Delta, Clip and iPSC machines. These files specify the type of machine, all the relevant machine parameters, and the language parameters, if the machine has its own characteristic language.

Supported Machines

Delta Architecture

The following is a description of the Delta architecture:

- System Hardware configuration
 - 32 compute nodes maximum MIMD mode

Compute Node

- 33 MIs(Integer)
- 16 MBytes Main Memory (expandable to 64 MBytes)
- 160 MBytes/sec peak DRAM access rate

Delta System Interconnect

- System-Wide Communications Fabric
 - handles all inter-node communication
 - handles all I/O communication
- Automatically routes messages without interrupting intermediate compute nodes
- Programmer can ignore details of how messages move along the interconnect network
- Interconnect supports 28 MB/S node-node bandwidth

Note: The above is based on the Proceeding of the Delta New User Training Class Notes, July 1992.

Sequent Architecture

The following is a description of the Sequent architecture:

Hardware:

- there are two models.
 - Balance 8000 include from 2 to 12 processors
 - Balance 21000 include from 4 to 30 processors,
(32 max for evaluation)
- all processors share one bus to global memory

- inter-processor communication through main memory
- main memory ranges from 4 to 28 MBytes
- each processor has 8KByte on-chip cache
- each processor has an 8KByte local memory

CM5

The following is a description of the CM-5 architecture:

Hardware

- 32 Processing Nodes (PN) maximum
- three networks connect all nodes
 - Control Network (CN) for concurrent operations
 - Data Network (DN) for bulk data transfer
- specific hardware and software support improve speed of many special cases

Processing Nodes

- general purpose (RISC) computer
- 8, 16 or 32 MBytes of memory (32 maximum for experiments)
- 64KByte cache for instructions and data

Network

- each PN has its own Network Interface (NI)
- once the Data Network accepts a message, it takes on all the responsibility of delivering the message.
- Data can be transferred between I/O devices without involving the Processing Nodes
- the Control network handles special global operations
 - broadcasting
 - reduction
 - parallel prefix
 - synchronization
 - error signaling
- guaranteed network bandwidth
 - DN: 5 MBytes/sec
 - CN: 20 MBytes/sec
- the networks are completely scaleable

CLIP

The following is a description of the CLIP architecture:

- 96 x 96 SIMD processor array
- each processor has two boolean processors

- local access takes same time as neighbor access
- 80ms per bit plane input
- UNIX host
- lock, unlock - keeps a bit plane in memory if possible

iPSC Hypercube

The following is a description of the iPSC architecture:

- n-dimensional array of processors (n=5 maximum = 32 processors)
- 7 bi-directional channels on each node (maximum 4 used when n=4)

Software Module

The software module is responsible for interpreting the application code files. These files are written in a new language called RAP. RAP has several modes of operation that allow the user to write code in a generic pseudo-C, or in a simplified version of the language widely supported on specific machine. All the language modes support parallel programming. All the languages modes assume only integer data types in parallel structures. Non-parallel integer variables' values are tracked, if possible.

Supported Modes

RAP

RAP mode supports a simple C-like language that allows the user to evaluate standard C-like programs with little modification. There are sample .RAP files below.

Sequent

Sequent mode supports library routines that are used to implement parallel programs. Again, the .ACH files hold the appropriate information for these functions.

CM5, Delta and iPSC Hypercube

These machines all have a library of routines for message passing. Their details differ slightly, but the names of the routines and their usage are all the same.

CLIP

The CLIP supports CLIP primitive system calls as described above.

Sample Files

Sequent Architecture Parameters

```

/*****
/*   Hardware definition for the Sequent                               */
machine = sequent             /* define what machine this is */
num_processors = 4           /* typically range from 2 - 30 */

/*****
/*   Language-specific definitions
/*   Parameter definition of general operations
/*
int + int = 2.5 us
int - int = 2.5 us
int * int = 11.6 us
int / int = 14.4 us

/* loop-overhead, i.e. for ( i=1; i<100; i++);                      */
loop_overhead = 3.5 us

/* call without parameters or statements                            */
subroutine_call_n_return = 11 us

/** Parameter definition of machine-specific functions */
/* memory specs */
local_memory = 8 kb
local_access_time = 1 us
local_cache_size = 96 bytes
local_cache_access_time = 0.1 us
local_cache_hit = 1 us
local_cache_miss = 5 us
local_cache_block_size = 32 bytes

shared_memory = 16 kb
shared_access_time = 10 us
shared_cache_size = 32 bytes
shared_cache_access_time = 0.1 us
shared_cache_hit = 1 us
shared_cache_miss = 5 us
shared_cache_block_size = 4 bytes

```

Sequent Application file

```
/* This is an example RAP Sequent program */
#language Sequent

int xyzzy[64],abc[64];          /* These are shared & global*/
int joe,in[8][16],out[8][16];
int fullx, halfx,fully,halfy;

/* globals are in shared memory */
main()
{
    int      i,z; /* example variable declarations */

    /* break image into 4 quadrants, add joe to each element*/
    m_fork(4,&doit);

}

doit(void)
{
    int      my_id;
    int      xu,xl,yu,yl;
    int      x,y;

    my_id=my_pid();
    if(my_id == 0)
    {
        xl=0;
        xu=halfx;
        yl=0;
        yu=halfy;
    }/* do upper left      */
    else if(my_id == 1)
    {
        xl=halfx;
        xu=fullx;
        yl=0;
        yu=halfy;
    }
    else if(my_id == 2)
    {
        xl=halfx;
        xu=fullx;
        yl=halfy;
        yu=fully;
    }
    else if(my_id == 3)
    {
```

```
    xl=halfx;  
    xu=fullx;  
    yl=halfy;  
    yu=fully;  
}  
  
for(x=x1; x<xu; x=x+1)  
{  
    for(y=y1; y<yu; y=y+1)  
    {  
        out[x][y]=in[x][y]+joe;  
    }  
}  
}
```


CM5 and iPSC Architecture Parameters

```
/* Notice that CM5 and iPSC have identical format */
/*****
/*   Hardware definition for the delta */
machine = delta      /* define what machine this is */
num_processors = 8

/*****
/*   Language-specific definitions
/*   Parameter definition of general operations
*/
int + int = 2.5 us
int - int = 2.5 us
int * int = 11.6 us
int / int = 14.4 us

/* loop-overhead, ie. for ( i=1; i<100; i++);
*/
loop_overhead = 3.5 us

/* call without parameters or statements */
subroutine_call_n_return = 11 us

/** Parameter definition of machine-specific functions*/
/* Communications parameters */
packet_size = 100 Bytes

/* memory specs */
local_memory = 8 KB
local_access_time = 0.5 us
local_cache_size = 8 KB
local_cache_access_time = 0.1 us
local_cache_hit = .1 us
local_cache_miss = 1 us
local_cache_block_size = 1 kb
```

Delta Architecture Parameters

```
/* Notice that CM5 and iPSC have identical format */
/*****
/*   Hardware definition for the delta */
machine = delta      /* define what machine this is */
num_processors = 2 by 4

/*****
/*   Language-specific definitions
/*   Parameter definition of general operations
/*
int + int = 2.5 us
int - int = 2.5 us
int * int = 11.6 us
int / int = 14.4 us

/* loop-overhead, ie. for ( i=1; i<100; i++);
/*
loop_overhead = 3.5 us

/* call without parameters or statements */
subroutine_call_n_return = 11 us

/** Parameter definition of machine-specific functions*/
/* Communications parameters */
packet_size = 100 Bytes

/* memory specs */
local_memory = 8 KB
local_access_time = 0.5 us
local_cache_size = 8 KB
local_cache_access_time = 0.1 us
local_cache_hit = .1 us
local_cache_miss = 1 us
local_cache_block_size = 1 kb
```

CM5 and iPSC application code

```
/* This is a sample file that uses communications system calls */
#language cm5
main()
{
    int a;
    a = my_pid();
    if( a == 0 ){
        a();
    }
    else if( a==1){
        b();
    }
    else if( a==3){
        c();
    }
    a=2+a*a;
}
a(void)
{ /* 3->0->1 */
    int i,j;

    i=9;
    j=irecv(300,100,100);
    isend(0,100,100,1,0);
    msgwait(j);
}
b(void)
{ /* 0->1->3 */
    int j,k;
    j=0;
    k=irecv(000,100,100);
    for(j=0; j<100; j=j+1){
        j=j-k;}
    j=isend(100,100,100,3,0);
/*    msgwait(j);    */
/*    msgwait(k);    */
}
c(void)
{ /* 1->3->0 */
    int j;

    j=8;
    j=irecv(100,100,100);
    msgwait(j);
    j=isend(300,100,100,0,0);
}
}
```

CLIP: (Cellular Logic Image processor) architecture file

```

/*****
/*   Hardware definition for the Sequent                               */
machine = CLIP           /* define what machine this is */
num_processors = 96 by 96

/*****
/*   Language-specific definitions
/*   * Parameter definition of general operations */
int + int = 2.5 us
int - int = 2.5 us
int * int = 11.6 us
int / int = 14.4 us

bitplanes = 32
bitplane_access = 80 ms
lda = 12 ms
ldb = 12 ms
pst_local = 12 ms
pst_pointwise = 12 ms

/* loop-overhead, ie. for ( i=1; i<100; i++);      */
loop_overhead = 3.5 us

/* call without parameters or statements          */
subroutine_call_n_return = 11 us

/** Parameter definition of machine-specific functions */

/* memory specs for CLIP's host system*/
local_memory = 8 KB
local_access_time = 0.5 us
local_cache_size = 8 KB
local_cache_access_time = 0.1 us
local_cache_hit = .1 us
local_cache_miss = 1 us
local_cache_block_size = 1 kb

/* these are just placeholders */
shared_memory = 16 MB
shared_access_time = 1.2 us
shared_cache_size = 8 KB
shared_cache_access_time = 0.1 us
shared_cache_hit = .1 us
shared_cache_miss = 1 us
shared_cache_block_size = .5 KB

```

CLIP: (Cellular Logic Image processor) application file

```
/* This is a sequent mode RAP program */
#language CLIP

image a[4][96][96],b[8][96][96],c[4][96][96];

main()
{ int i;

  image A[4][96][96],B[8][96][96],C[4][96][96],D[8][96][96];
  image x[16][96][96],y[16][96][96];

  im_lock(x);
  fred();
  for(i=0; i<16; i=i+1)
  {
    LDA(C[i]);
    LDB(D[i]);
    PST_local(y[i]);
  }
  fred();
  fred();
}

fred(void)
{
  int i,j;

  for(i=0; i<8; i=i+1)
  {
    for(j=0; j<4; j=j+1)
    {
      LDA(c[i]);
      LDB(b[j]);
      PST_pointwise(a[i+j]);
    }
  }
}
```

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.