12-1993

Design and Development of a Heterogeneous Parallel Computing System

Eruch R. Rustomji

Recommended Citation

Rustomji, Eruch R., "Design and Development of a Heterogeneous Parallel Computing System" (1993). *Master's Theses*. 4248.
https://scholarworks.wmich.edu/masters_theses/4248

DESIGN AND DEVELOPMENT OF A HETEROGENEOUS PARALLEL COMPUTING SYSTEM

by

Eruch R. Rustomji

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1993

ACKNOWLEDGMENTS

Firstly, I would like to thank my major advisor, Dr. Ajay Gupta, who has been my mentor and friend throughout my thesis project. His suggestions stimulated new ideas and added a creative touch to the thesis. To the members of my committee, Dr. Thomas F. Piatkowski and Dr. Elise deDoncker, I extend my sincere appreciation for their guidance and support throughout the entire thesis project.

Secondly, I thank my family, for their support in all my endeavours. I extend my sincere thanks to my colleagues, Mr. Winston Anand-Kumar, Mr. Siddharth Bhingarde, Mr. Arun Shanbhag and Mr. Omprakash Mishra for their never-ending support.

Finally, I dedicate this thesis to my Lord Ahura Mazda, for giving me strength, motivation and determination in all my work.

Eruch R. Rustomji

DESIGN AND DEVELOPMENT OF A HETEROGENEOUS PARALLEL COMPUTING SYSTEM

Eruch R. Rustomji, M.S.

Western Michigan University, 1993

A parallel and distributed processing environment can be defined as one where a set of workstations is configured in a certain topology (such as completely connected linear chain) to simulate the working of a particular parallel architecture. Such an environment provides an extremely useful means of experimenting with parallel algorithms, without the use of expensive dedicated parallel machines.

Several parallel and distributed processing environments exist, such as Parallel Virtual Machine, The Condor System and the Reactive Kernel System/Cosmic Environment System. Each of these systems has some unique features and limitations. Other systems include p4, Hermes, Linda and Express.

This thesis describes the HPCS system and its three modes of operation with examples and briefly discusses future expansion potentials .

In this thesis, we describe the design, development and features of the Heterogeneous Parallel Computing System, a parallel and distributed processing environment, which allows program development given a limited set of resources and in a reasonable amount of time. It employs three modes of operation to suit the user's needs; the central-server mode and distributed-server mode are client-server models and the client-serve mode of HPCS is a serverless model. The HPCS system has been conceptually designed to overcome limitations posed by the *well-known systems* (e.g. PVM, RK/CE, Condor).

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

- The Problem 1
- Parallel and Distributed Processing Environments 2
  - The Reactive Kernel/Cosmic Environment (RK/CE) 3
  - The Condor System 4
  - The Parallel Virtual Machine (PVM) 4
  - Heterogeneous Parallel Computing System 6
- HPCS v/s PVM, RK/CE and Condor 8

II. THE UNDERLYING MECHANISM 10

- The OSI Model and Networking Standards 10
  - The Open Systems Interconnection Model (OSI) 10
- Protocol Suites 11
  - TCP/IP - The Internet Protocols 11
  - Transport Layer - TCP and UDP 12

III. CENTRAL SERVER MODE 17

- System Setup and Configuration 17
- Server Mechanism Description 18
- Mechanism Involved 20

Table of Contents – Continued

CHAPTER

The Reduction Program 20

IV. DISTRIBUTED SERVER MODE 22

System Setup and Configuration 22

Mechanism Involved 22

Finding Fundamental Cycle Set in a Graph 23

Observations 25

V. CLIENT-SERVE MODE 26

System Setup and Configuration 26

Mechanism Involved 27

Finding Co-Occurrence Values of an Image 27

VI. HPCS SOFTWARE LIBRARY 29

Library Calls 29

Simple Example Using HPCS Primitives 30

VII. X-WINDOW GUI 33

The HPCS Process Monitor 33

VIII. FUTURE EXPANSIONS 34

Future Expansions to the GUI 34

Future Expansions to HPCS 34

IX. CONCLUSION 35

APPENDICES

A. Central-Server Mode Example Program 36
B. Distributed-Server Mode Example Program 39
C. Client-Serve Mode Example Program 45
REFERENCES 49

LIST OF TABLES

1. Comparison of Protocol Features for IDP, UDP and TCP 14

LIST OF FIGURES

1. Example Configuration of a Parallel and Distributed Environment. 2
2. Skeleton of Host and Node Programs in HPCS. 7
3. OSI Seven-Layer Model. 11
4. Layering in the Protocol Suite. 12
5. Four-Layer Model Showing TCP, UDP and IDP. 13
6. TCP/IP Communication Example Between Two Machines. 16
7. HPCS Central Server Mode Configuration. 17
8. Socket System Calls for Connection-Oriented Protocol. 19
9. HPCS Distributed Server Mode Configuration. 23
10. HPCS Client-Serve Mode Configuration. 26

CHAPTER I

INTRODUCTION

The Problem

A parallel/distributed application program is one which performs a functional task in a parallel manner (simultaneous execution) by distributing any given set of data which is divided into several parts and the functions are executed simultaneously on each individual part. The individual results obtained are then combined together to get the final result. The parallel program is usually run on a parallel machine [1, 13, 15, 16, 17].

Recently a new trend has developed called Heterogeneous Computing. It involves a parallel program and/or distributed application running on a cluster of workstations. A common computing environment consists of workstations of different architectural bases, connected together by a high-speed network. These workstations have grown in power over the past few years and represent a significant computing resource [14, 16, 17, 18, 19, 21]

The main goal is to use any given set of already available workstations (an example configuration is shown in Figure 1) as a resource pool and to speed up the computational power by pooling in the workstations together to provide a parallel and distributed environment. If p processors are available, then in parallel/distributed computation, the *ideal* goal is to execute the application on these p hetero-processors pseudo parallel machine, such that the application runs atleast p times faster than the fastest known solution on the slowest hetero-processor. The figure on the next page shows an example configuration where four different types of workstations are connected by a network along with a network manager.

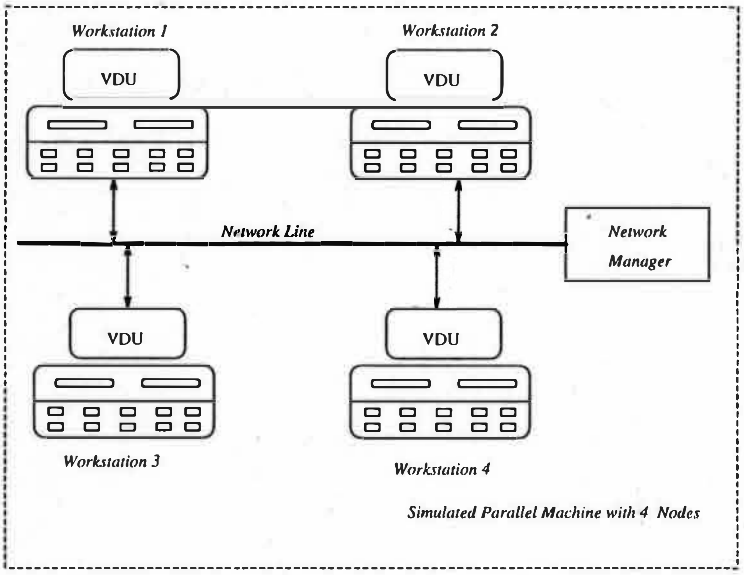

Figure 1. Example Configuration of a Parallel and Distributed Environment.

For such a configuration of a system we need to provide a set of software primitives to the user to easily develop a coarse-grain parallel program without the availability of a parallel machine. The primitives should be robust and have incorporated features such as error detection, fault tolerance, timeout etc. One of the main objectives here is to develop software library which is easily portable for a similar network of workstations, presenting a parallel and distributed processing heterogeneous environment to the user.

Parallel and Distributed Processing Environments

A Parallel Processing Environment (PPE) is a network of workstations (homogeneous or heterogeneous) configured together as a resource pool alongwith a software package which as a whole offers the equivalent power of computation, as that of a similar parallel machine. In effect, it simulates a parallel machine using the set of already available workstations. The software developed to offer this parallel and distributed power to the user is called a Parallel Simulator and the HPCS system which we describe here suits the purpose.

PPE's can be developed on any network of machines and PPE's can be implemented using a variety of system software available. This includes Berkeley Socket calls which is part of the Berkeley Socket Distribution (BSD), Transport Layer Interface (TLI) provided by IBM and SUN Remote Procedure Calls (RPC).

There have been several such development environments available to use namely, the Reactive Kernel/Cosmic Environment System [2], the Parallel Virtual Machine [12], the Condor System [6], the p4 system, the Hermes PPE language, the Linda System and Parasoft's Express. We describe briefly the first three systems.

The Reactive Kernel/Cosmic Environment (RK/CE)

The *RK/CE* system was developed by C. Seitz, Jakov Seizovic and Wen-King Su at the California Institute of Technology [2]. A program written in RK/CE can run on an actual parallel machine (such as Simult 2010, Cosmic IPCS II, Intel IPSC 8/60) without modification.

It can be configured as a ghost cube or a non-cube.[1] The host program running on the host machine of the user, makes the node program ready to run on the PE's(nodes), allocates the data to the nodes and spawns processes at every node in the cube. The results are then received by the host from the nodes when the tasks are completed. Each node has its own local memory, and nodes communicate among themselves through messages. A set of message passing primitives is given to the user as a library call. These primitives include standard C binding calls and they are archived in the RK/CE library. All paths used by the

---

[1] For a definition of the commonly used fixed interconnection architectures for parallel machines, the reader is referred to [15].

RK/CE system for the user, should have world executable access. RK/CE uses Unix Datagram Protocol (UDP) and Berkeley Socket calls as its base structure.

The Condor System

The *Condor System* developed by by M. Litzkow, M. Livny and M. Mutka at the University of Wisconsin, Madison [3, 6]. It is the result of an attempt to make use of idle cycles on workstations by monitoring the activity on all participating workstations in the *local network*. Those machines which are determined to be idle, are placed into a resource pool or “processor bank”. Machines are then allocated from the bank for the execution of jobs. The bank is a dynamic entity, workstations enter the bank when they become idle, and leave the bank when they become busy [3, 4, 5].

No special programming is required to use *Condor*. The local execution environment is preserved for remotely executing processes. It takes care of data file movements for remote execution of programs. It uses a special algorithm to locate and allocate idle workstations. If a job is stopped when a user returns to his/her workstation the *Condor* system will eventually checkpoint and restart the job on another workstation. It runs outside the UNIX kernel, and is compatible with BSD 4.2 and 4.3.

The Condor system basically has been oriented towards idle time resource utilization and effective location and allocation of jobs on such workstations.

The Parallel Virtual Machine (PVM)

Parallel Virtual Machine (PVM) was developed by A. Begeulin and C.A. Geist at Oak Ridge National Laboratory, J. Dongarra and R. Manchek at University of Tennessee and by V. Sunderam at Emory University [12]. It is a software package that enables concurrent computing on loosely coupled networks of processing elements.

PVM may be implemented on a hardware base consisting of different machine architectures, including single CPU systems, vector machines, and multiprocessors. These computing elements may be interconnected by one or more networks, which may themselves be different (e.g. Ethernet and Internet, MCInet, ATTnet and UUnet). These computing elements are accessed by applications via a library of standard interface routines. These routines allow the initiation and termination of processes across the network as well as communication and synchronization between processes.

Application programs are composed of *components* that are subtasks at a moderately large level of granularity. During execution, multiple *instances* of each component may be initiated.

Application programs view PVM as a general and flexible parallel computing resource that supports a message-passing model of computation. This resource may be accessed at three different levels: the *transparent* mode in which component instances are automatically located at the most appropriate sites, the *architecture-dependent* mode in which the user may indicate specific architectures on which particular components are to execute, and the *low-level* mode in which a particular machine may be specified. Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network. The PVM user interface is strongly typed; support for operating in a heterogeneous environment is provided in the form of special constructs that selectively perform machine-dependent data conversions where necessary.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and, further, any process may communicate and/or synchronize with any other. This is the most unstructured form of crowd computation, but in practice a significant number of concurrent applications are more structured. Two typical structures are the tree and the "regular crowd" structure. We use the latter term to denote parallel computations in which all processes are identical; frequently such applications also exhibit regular communication and synchronization patterns. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

The PVM system is client-server model, which uses both Unix Datagram Protocol (UDP) and TCP (Transmission Control Protocol) for communication purposes. PVM daemons are run on a set of host machines, each co-ordinating control and data traffic. The pvmd daemons use UDP with each other and client-to-client communication is done using TCP sockets.

Heterogeneous Parallel Computing System

The *Heterogeneous Parallel Computing System* (HPCS) was designed and developed by A. Gupta, E. Rustomji and R. Kamenine at Western Michigan University. An attempt has been made here to present a parallel and distributed processing environment to develop a parallel program on a network of workstations. The environment provides most of functionalities existing on dedicated parallel machines such as nCube, IPSC 860, BBN Butterfly.

A parallel program can be viewed as a set of tasks, each task running on a workstation (node) of the parallel machine. Furthermore, at any instant of time, during the execution of the task, the node is either performing some computation or is engaged in communication with another node. We view the parallel program development in a MIMD (Multiple Instruction Multiple Data) environment. Any user program implemented on a parallel machine may be abstractly viewed to consist of two sets of sub-programs, a *host* program and a *node* program. An example setup is shown in Figure 2. The host program is the user-level data

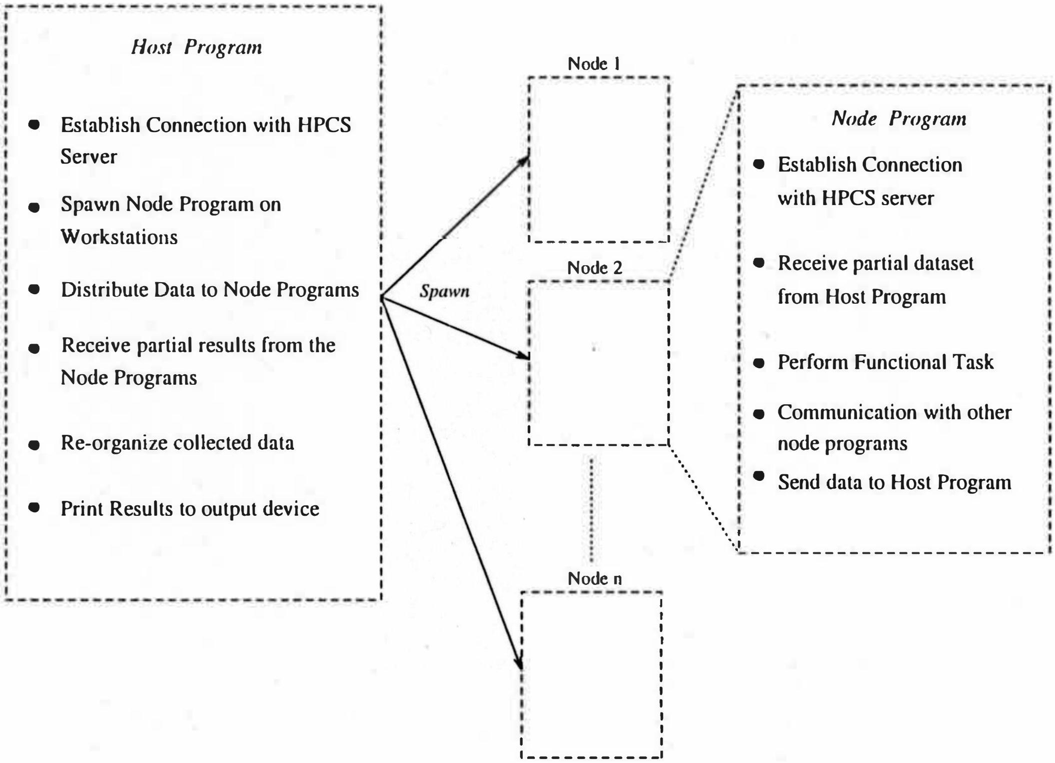
Host Program
Establish Connection with HPCS Server
Spawn Node Program on Workstations
Distribute Data to Node Programs
Receive partial results from the Node Programs
Re-organize collected data
Print Results to output device
Spawn
Node 1
Node 2
Node n
Node Program
Establish Connection with HPCS server
Receive partial dataset from Host Program
Perform Functional Task
Communication with other node programs
Send data to Host Program

Figure 2. Skeleton of Host and Node Programs in HPCS.

controlling program, which spawns a copy of the node program on several processors of the parallel machine, breaks up the original data into several subsets and passes each of the subsets to the spawned node programs. The node programs now work on their individual subsets in parallel, compute the results and pass the results back to the host program. The host program then combines these results to calculate the overall solution to the problem. HPCS provides primitives to spawn the node program on the several workstations. These primitives are used in the host program prepared by the user. HPCS also provides communication primitives to distribute the data to the spawned node programs, and to receive the data(results) from them.

All communication takes place through the underlying mechanism which controls data flow and connection control. The HPCS uses TCP/IP. It has been implemented using Berkeley Socket Calls.

The HPCS system is configurable into three modes of operation namely, Single Server Mode, Distributed Server Mode, and Client-Serve Mode. The ensueing chapters describe these three modes and how the host task and node tasks behave in each particular configuration.

An X-window GUI has been developed for process monitoring. Future expansion potentials are discussed in the later chapters.

HPCS v/s PVM, RK/CE and Condor

From the above sections, we have observed the various features, each of the parallel and distributed environments offer us. In this section, we compare our newly developed HPCS system with PVM, RK/CE and Condor.

We note that the RK/CE environment mainly views the parallel machine with the hypercube model in mind. It operates only on homogeneous environments.

The PVM system uses a mix of the UDP and TCP protocols, and it provides its own mechanisms of reliability when using UDP. This type of reliability mechanism is system dependent and comes under layer 6 and 7 of the 7-layer OSI model. The Condor system is actually an idle workstation hunter, which is its primary job.

Our goal here is to present, a PPE, which provides system level reliability using strictly TCP (a very reliable, efficient protocol and widely available protocol) using a given set of workstations as a resource pool which should be configurable with any parallel model in mind and has the potential to perform load balancing (idle workstation hunting).

The HPCS system is oriented towards presenting such a system and is in the process of continously adding more features to it, in order to surpass not equal the capabilities of other similar systems we may encounter.

CHAPTER II

THE UNDERLYING MECHANISM

The OSI Model and Networking Standards

In this chapter, factors have been presented which contribute to the development of the HPCS system. Firstly, the goal of all protocol designs is to find the fundamental abstractions that can be reused in multiple applications. In most UNIX based systems, the complete network system follows the Open Systems Interconnection model (OSI).

There are many different organizations involved in the development of networking standards. Both the organizations and standards are referred to by acronyms in most networking literature. The International Standards Organization (ISO), was founded in 1946 and develops standards on a wide variety of subjects. The members of ISO, are the national standards organizations from the member countries. The American National Standards Institute (ANSI), is the U.S. member of ISO. The Open Systems Interconnection model was adopted by ISO in 1984 [7, 8].

The Open Systems Interconnection Model (OSI)

The OSI model provides a detailed standard for describing a network. Most computer networks are described today using the OSI model as a reference. From the seven-layer OSI model shown in Figure 3, the lowest of the seven layers which provides reliable data transfer between any two systems is the transport layer. Error detection, checksums, retransmission and flow control are handled by the transport layer.

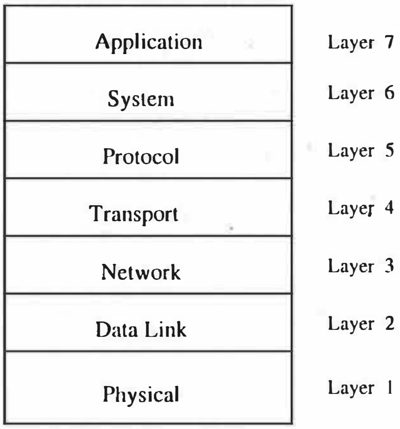

Figure 3. OSI Seven-Layer Model.

Protocol Suites

There are many protocol suites used in most major networking systems today. Some of the them are listed here : (*a*) TCP/IP protocol suite (the Internet protocols), (*b*) Xerox Networking Systems (Xerox NS or XNS), (*c*) IBM's Systems Network Architecture (SNA), (*d*) IBM's NetBIOS, (*e*) The OSI protocols and (*f*) Unix-to-Unix Copy (UUCP).

Figure 4 demonstrates the relationship of the protocols in the Internet protocol suite. In this chapter we will mainly examine the TCP and UDP protocols of the Internet Protocol family.

TCP/IP - The Internet Protocols

The family of Internet protocols consist of many members. We examine two members which are referred to as the Transmission Control Protocol/Internet Protocol (TCP/IP) and Unix Datagram Protocol (UDP). OSI layers 5-7 comprise of the user process, which include those developed by the programmer. *The HPCS system lies within these layers.* OSI layer 4, which is the Transport layer, consists of TCP or UDP. OSI layer 3 which is the Network layer consists of Internet Control

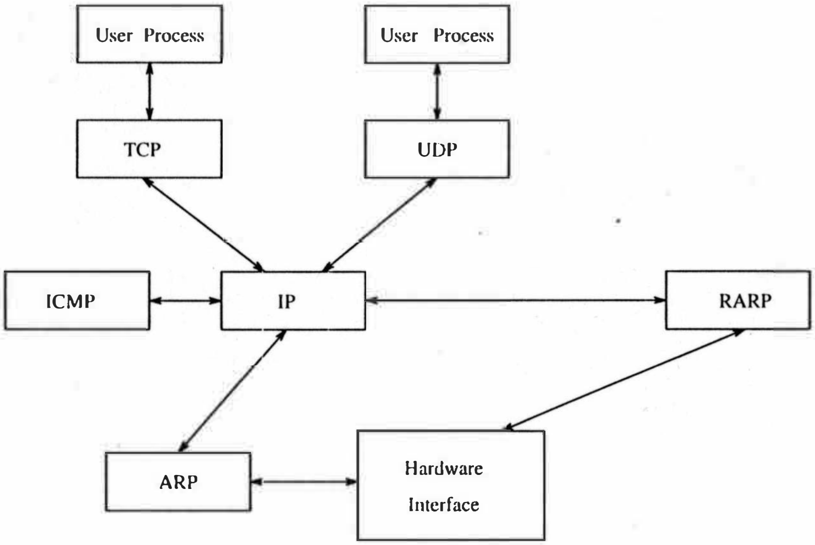

Figure 4. Layering in the Protocol Suite.

Message Protocol(ICMP), the protocol to handle error and control information between gateways and hosts. ICMP messages are transmitted using IP datagrams.

The IP is the protocol that provides a packet delivery service for TCP, UDP and ICMP. The Address Resolution Protocol (ARP) maps an Internet address into a hardware address. The Reverse Address Resolution Protocol (RARP) maps the hardware address into an Internet address. The ARP and RARP are not used on all networks. OSI layer 1 and 2 are the Physical and Data link layers, which is together considered as the hardware interface is shown in Figure 4.

Transport Layer - TCP and UDP

The Application layer user processes communicate with each other by sending and receiving TCP data or UDP data. Let us simplify the relationship into a 4-layer model showing TCP, UDP and IP, as shown in Figure 5. These two protocols are sometimes referred to as TCP/IP and UDP/IP. TCP provides a

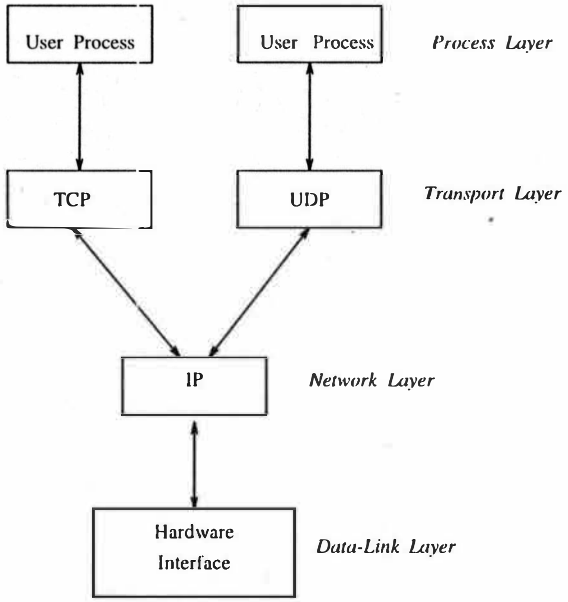

Figure 5. Four-Layer Model Showing TCP, UDP and IDP.

connection-oriented service. It also provides reliable message broadcasting, full-duplex communication lines and a byte-stream service to an application program.

On the other hand, UDP provides an unreliable, connection-less datagram service. Since HPCS assumes that the communication between its clients (host and node programs) are reliable and secure, and these clients have to be connected, TCP/IP was chosen as the protocol and the Internet Address Family (AF_INET) was chosen as the means of communication for the channel through which HPCS server communicates.

Table 1 compares TCP, UDP and IP. Note that the IP layer provides an unreliable, connectionless message delivery service but it is the TCP module which contains the necessary logic to provide a reliable virtual circuit to the user process. TCP handles the establishment and termination of connections between processes, the sequencing of data, the end-to-end reliability (checksums, timeouts, acknowledgements) and end-to-end flow control. On the other hand, UDP provides only

Table 1

Comparison of Protocol Features for IDP, UDP and TCP

Protocol Feature	IP	UDP	TCP
connection oriented ?	no	no	yes
message boundaries ?	yes	yes	no
data checksum ?	no	opt	yes
positive ack. ?	no	no	yes
timeout and rexmit ?	no	no	yes
duplicate detection ?	no	no	yes
sequencing ?	no	no	yes
flow control ?	no	no	yes

two features that are not provided by IP: port numbers and checksum verification for the contents of the UDP datagram. Thus if a user process uses UDP instead of IP directly, then it can operate in a connection-less, unreliable environment presented here by UDP. TCP/IP was the chosen one, as it is connection-oriented, provides re-transmission control, data checksum, duplicate detection, sequencing, flow control and positive acknowledgement.

The port number service provided by UDP and TCP allow a client process to contact and identify a server process residing on a particular host system. Port numbers distinguish different servers and given the host IP address and the port number through which a particular server accepts requests, a client can establish a connection with the server. A client-server application like HPCS can specify a particular port number used, from the list of ephemeral port numbers available for program development. Note that the X-window system which is a Network Transparent Window System, also makes use of these port-numbers. A similar layout is shown in the Figure 6. It is also noted that TCP ports are independent of UDP ports. TCP port 1890, for example is independent of UDP port 1890.

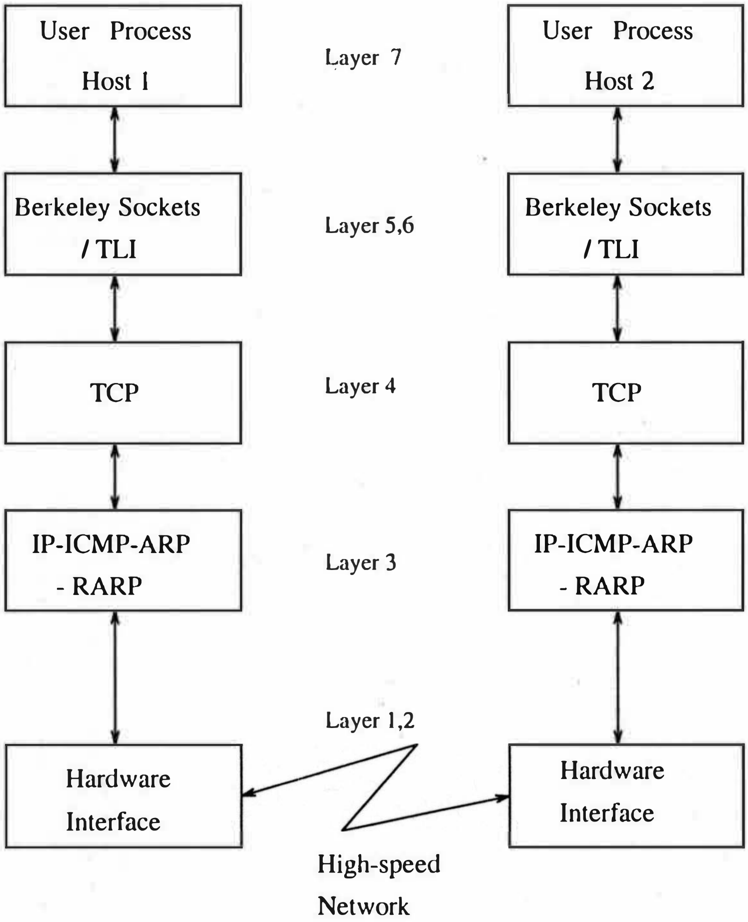

Figure 6. TCP/IP Communication Example Between Two Machines.

CHAPTER III

CENTRAL SERVER MODE

System Setup and Configuration

The HPCS system is a client-server model, which can be configured in three modes of operation, the Central Server Mode, the Distributed Server Mode and the Client-Serve Mode. The first mode of operation is called the Central Server mode. In this mode, the host program spawns the node program on several nodes (workstations) using the Central Server running on the host node. A typical program usually establishes a connection with the Central Server and spawns the node programs on however many nodes as required. To support these operations the PPE should allow spawning of processes on nodes, handle communication between different processes etc. The HPCS supplies the user with a set of primitives

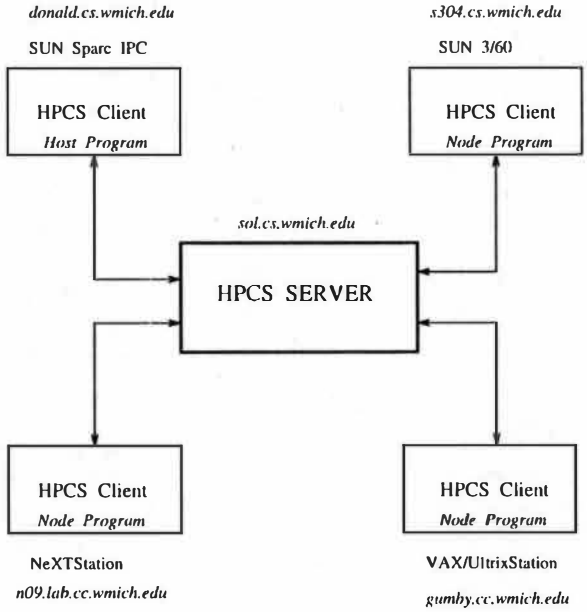

Figure 7. HPCS Central Server Mode Configuration.

to perform such functions. The important point to be noted about this mode of operation is that the Central Server (or one of supporting child processes) is involved in all the communication. This can be viewed as a fully interconnected architecture. In this chapter using the reduction problem as en example, we have shown how we can apply the HPCS primitives in this mode of operation and get the end results. Figure 7, shows that there exists one single HPCS server and client programs (both host and node programs of the user) runnning on a given set of nodes and which communicate via the central server. The HPCS central server is first setup to run as a daemon process[1] on one of the workstations. Then the server listens for connections on an available port, via the select() system call, where the port number is set up at the time of installation.

Server Mechanism Description

On a connection request, the server stores the client's information (e.g. client's IP address, client's name, client's file descriptor). It then proceeds to fork a slave process, for the connection request, which handles all I/O associated with that client, using the client's fd, as the I/O channel (See Figure 8.)

In this manner, the server forks several slave processes, which independently handle the client connections. Before a client exits, the user program uses a HPCS primitive to send a message to this slave process to indicate termination which in turn terminates. In this manner, the parent server process, basically, awaits connection requests, processes client information and forks a slave process, and returning back to await the next connection request.

---

[1] For further details about daemon processes, the reader is referred to [7, 8]

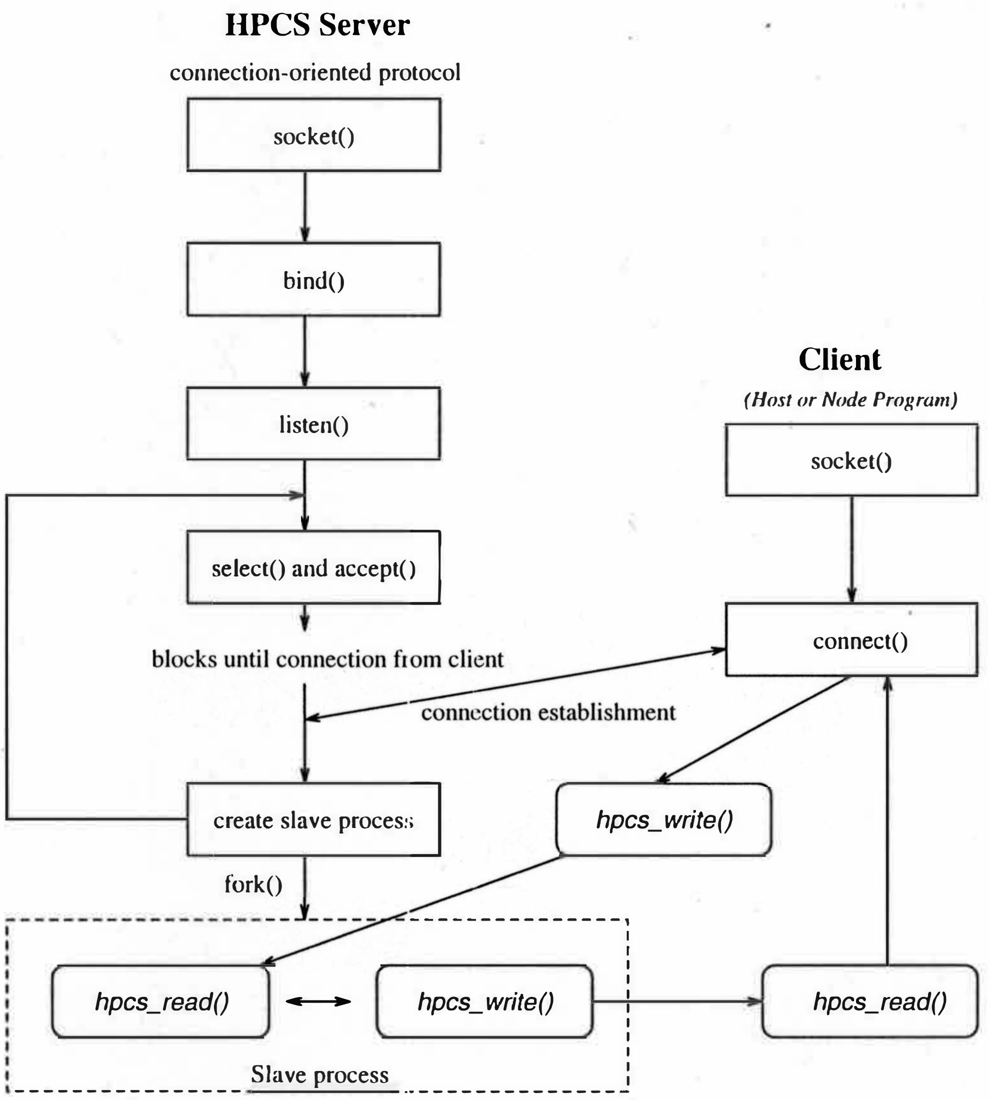

Figure 8. Socket System Calls for Connection-Oriented Protocol.

Mechanism Involved

The mechanism used by the server process to await a connection request is made very simple using Berkeley socket calls. The step-by-step process is described below (See Figure 8.):

1. Server creates a communication socket, using AF_INET (Address Family - Internet for TCP/IP). This socket returned is the listening port.

2. Server binds its Internet address and an available port number to the socket.

3. The server listens for connections using the listen() system call.

4. Upon a connection request, the server accepts the connection by issuing an accept() call, to setup a unique communication channel.

5. The server creates a slave process, passing the client's communication channel fd, returned by accept(), to allow it to handle any I/O associated with that particular connection.

In this manner the server follows the design of a connection-oriented (TCP/IP), concurrent (slave processes) server. It can set itself up as a daemon process which is a non-tty attached process. This allows the server to dis-associate itself from the terminal and become login session leader.

The Reduction Program

In order to further understand this first mode of operation, let us take an example which can show us how to develop a program under HPCS and use its provided software primitives.

A good example for this mode of operation is the Reduction problem. Given is a set of n numbers and we need to find the number which has the maximum value among this set of numbers. We are also given a set of p hetero-processors under HPCS. In this example the host program divides the n numbers into p parts and sends each individual part to each node program executing on each node (a.k.a. workstation). Now each node program receives the data and performs its own internal sort and computes the maximum value from its individual dataset. Each node sends that maximum value to the host program via the central server and the host accumulates these individual max values into an array. The host calculates the max value among the values it received and outputs the result.

The code for the above example is given in Appendix A. A set of HPCS software primitives used in the code are briefly described here:

1. hpcs_init() - Establishes connection with the central server.

2. hpcs_spawn() - Spawns node programs on workstations.

3. hpcs_read() - Reads received data into a buffer.

4. hpcs_write() - Sends the contents of the buffer to the designated node program.

5. hpcs_close() - Closes the communication channel and inform the central server that connected node program has ended.

The reader is referred to Chapter VI for further details of these primitives.

CHAPTER IV

DISTRIBUTED SERVER MODE

System Setup and Configuration

In this mode, instead of having a single server running on a single workstation, several servers will be running; one on each workstation. Each node program on individual machines communicate only with the local server. If the program on some node *i* wants to communicate with another program on node *j*, it merely talks to its local server. The local server then establishes a connection with the local server on node *j* and sends it the data which is passed to the node program on that node (See Figure 9). The main advantage of this mode of operation is that there is no congestion on a single port. Since all the communication overhead is distributed among all nodes in the network, the time spent for such communication is significantly reduced. However the intertask communication between the local server and the local node program, within a machine is increased. In the first mode of operation if the Central Server node crashes, the whole system comes to a halt. Another important advantage of this mode of operation is that even if some nodes crash, the system can still function. This does, however, require that status tables should all be kept up to date and consistent on all nodes.

Mechanism Involved

With reference to Figure 9, each client maintains a set of fd's (file descriptors) for each connection it makes with each server residing on a workstation. In this manner, the I/O is not concentrated in the central server as described in the first mode of operation. The servers in this case, each maintain a set of fd's, similar to the central server in the first mode of operation. Testing can be done for

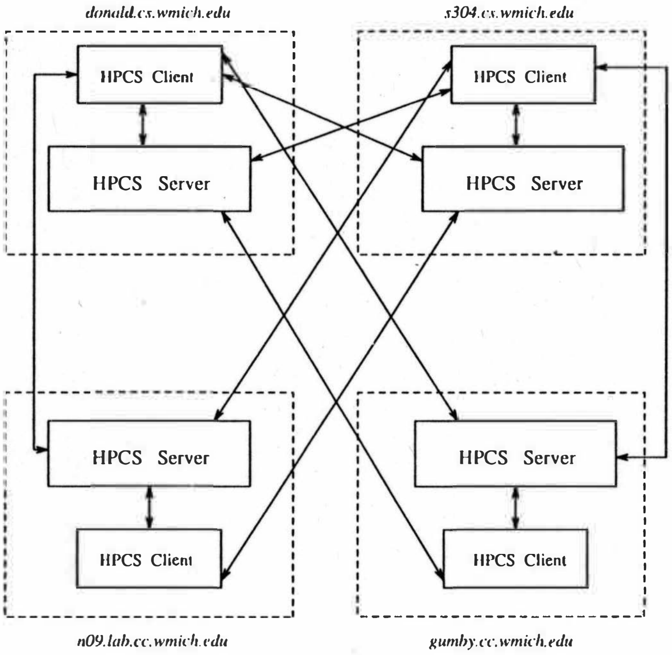

Figure 9. HPCS Distributed Server Mode Configuration.

this mode, to observe whether it reduces congestion of data flow over the Ethernet lines between workstations using Ethernet communication control programs.

Finding Fundamental Cycle Set in a Graph

Let us take an example suited to this mode of operation. We will use this example to show how to use the same set of HPCS software primitives to find the fundamental cycle set in a graph. A fundamental cycle exists in a graph or sub-graph if there exists a path in that vertice-set.

Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges, without self-loops or parallel edges. For completeness sake, we next define a few graph theory terminologies.

Subgraph: A subgraph of G is a graph whose vertices and edges are in G.

Connected Component: A graph is connected of there is a path between any pair of vertices in G. A maximal connected subgraph of G is called connected component.

Tree: A tree is a connected graph without cycles. A subgraph of a connected graph G is a *spanning tree* of G if it is a tree containing all the vertices of G. A *Spanning Forest* of a disconnected graph G is a collection of spanning trees, one for each connected component.

CoTree: A cotree in a connected graph G with respect to a spanning tree $T = (V, E_T)$ is the subgraph with a vertex-set V and the edge set $E - E_T$.

Fundamental Cycle Set: Adding an edge of a cotree to T creates a *fundamental cycle*. The set of all $m - n + 1$ cycles forms a fundamental cycle set in a connected graph with n vertices and m edges.

Bridge: An edge of a connected graph G is a bridge if its removal leads to the disconnection of G.

Bipartite Graph: A graph is said to be Bipartite if its vertex-set V can be partitioned into two subsets V_1 and V_2 such that none of the edges in E has both of its end-vertices in either V_1 or V_2.

We first construct a co-tree of G edges, such that each of its edges forms a fundamental cycle. In effect the forming of the co-trees from the output obtained from the Spanning_Forest_Global procedure described in [30, 31], makes us find the fundamental cycle by merely checking if a path exists from that vertice-set, using the following steps:

1. Using the procedure Spanning Forest Global, we identify the co-tree edges.

2. The host program divides the list of edges into p parts and sends each individual p part to each node program running on each node.

3. Next each node program examines its list of edges and finds if a particular edge belongs to the co-tree, if it does it finds its associated fundamental cycle.

4. Each node program then sends its result to the host program.

5. The host program accumulates these results sent by the individual node programs and outputs the total result.

Observations

The time complexity of the algorithm used in [30, 31] is calculated as $O(n(\log p + m/p))$. These set of algorithms use simple data structures, such as an unordered list of edges and they achieve optimal speedups for dense as well as sparse graphs. All these type of algorithms are optimally scalable up to a large number of processors depending on the graph density.

The code for the above example is shown in Appendix B. The primitives used in the code are described in Chapter VI.

CHAPTER V

CLIENT-SERVE MODE

System Setup and Configuration

This mode of operation is unique in the existing Parallel Processing Environments in that the host spawns a process to act as a temporary server and also spawns the node programs on different workstations. The slave process now awaits connections from all node programs. Each spawned node program establishes a connection with the slave server an also with other node programs. After storing all the parameters for the connection in a global table, the slave server kills itself. At this point we have a fully interconnected system. This mode has the advantage that no intertask communication takes place between a machine itself, however, the disadvantage is that the user program is burdened with taking care of fault tolerance. The configuration diagram is shown in Figure 10.

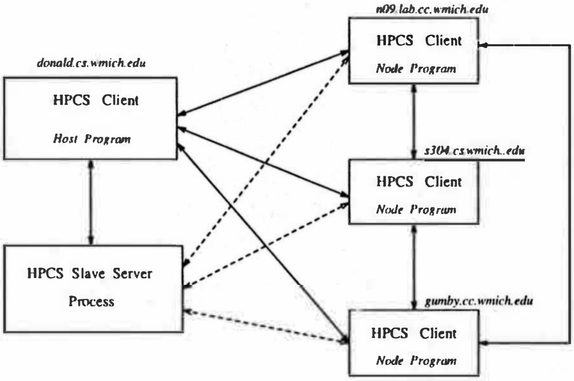

Figure 10. HPCS Client-Serve Mode Configuration.

The dashed lines, indicate that it is a temporary connection, whose parameters are passed back to the calling program (host program). The host program then uses these connection parameters, and uses it for communication purposes (shown by solid lines).

Mechanism Involved

The step-by-step process is described below:

1. The host program initiates the server slave process via a fork() call, and this server waits till all clients have made their connections.

2. The host then spawns, the client program on the workstations and the client programs in turn establish connection with the server slave process.

3. After all connections are stored in a global fd table, the server slave process exits.

4. At the same time, the client programs connect with each other.

5. The host program then distributes the data to each client.

6. The node programs in turn perform the algorithm to their individual data set and send their results to the host program.

7. The host program gathers the received data and outputs the overall results.

Finding Co-Occurrence Values of an Image

The third mode of operation is a serverless model allowing node programs to communicate directly with each other, if needed. To suit this mode of operation let us take an example which deals with large amounts of data.

Let us take the example of finding the co-occurrence values of a given image where the image has been defined in a N x N array. Each node program will receive one row of the array, assuming there are P processors where $P = N$.

The host program gets the data stored in a file into a N x N array and sends each row of the array to each node program running on each node.

The node program in turn stores the received data into a 1-dimensional array and finds the co-occurrence values. The node program then sends the calculated co-occurence values back to the host in a similar manner it receives it.

The host accumulates the received co-occurrence values, assimilates them and outputs the result.

The code for the above example is shown in Appendix C. The primitives used in the code are described in Chapter VI.

CHAPTER VI

HPCS SOFTWARE LIBRARY

The HPCS interface library is a set of primitives provided which can enable a user to develop his/her parallel program. In this chapter a detail description of the HPCS software primitives is given.

Library Calls

1. `hpcs_init(hostname)` - used in all three modes, where the hostname of the server to be contacted is supplied as an input parameter. Returns a fd to the caller which is used as a reference I/O channel.

2. `hpcs_spawn(program_name, hostname, no_nodes)` - In Central Server mode (i.e. mode 1), this primitive spawns node 'program_name', with 'hostname' specified as server machine to connect, 'no_nodes' times. In Distributed Server mode (i.e. mode 2), this primitive spawns 'program_name' on workstation, and node program establishes connection with server on local workstation. The Client-Serve mode (i.e. mode 3) is similar to the Central Server mode, except that temporary server is the same as host program which accepts connections through its slave server process.

3. `hpcs_read(fd, buffer)` - Used in all three modes and for client programs, to perform data I/O. The fd is the I/O fd (channel) obtained from the hpcs_init(hostname) call. This call reads the incoming data into 'buffer'. Maximum buffer length is set to a default of 255 bytes which can be changed at installation time.

4. `hpcs_write(fd, buffer)` - Used in all three modes and for client programs, to perform data I/O. The fd is the I/O fd (channel) obtained from the

hpcs_init(hostname) call. This call writes the data in 'buffer'. Maximum buffer length is 255 bytes, which can be changed at installation time.

5. `hpcs_close(fd)` - Called by client programs, to send a message to associated server, to shutdown connection. This message allows the slave processes in single-server mode and distributed-server mode to exit, thus terminating the connection, with the associated client.

6. `hpcs_connect_all(num_of_clnts)` - Used only in mode 3. It connects caller client, with all other clients, if present in the resource data file.

Note: The buffer message for hpcs_read() and hpcs_write() consists of:

$$< dest\_id, message >$$

where,

o In central server mode (mode 1) and client-serve mode (mode 3), if the dest_id = -1, then the server sends the message back to the sender, else it checks the client id in the client fd table, and sends it to the respective client.

o In the case of distributed server mode (mode 2), the server either sends it to the caller if dest_id = -1, else it sends the message to the client on its workstation.

Simple Example Using HPCS Primitives

Let us take a simple example to show the control flow of a parallel program written using HPCS primitives.

Assume that the hpcs server is running in on the workstation described as 'localhost'. The *host* program initially, establishes a connection with the hpcs server. It then spawns the *node* program using 'hpcs_spawn', giving the total number of node programs to be spawned. The 'hpcs_spawn' program in turn read the resource pool data file (which contains the workstation names, usernames and architecture type) and remotely executes the corresponding binary copy of the *node* program on the workstation specified.

The *host* program then writes to each of the *node* program some data in the "buffer". It then awaits a data receive from each of the *node* programs, till all data is received.

The *node* program in turn reads the data received from the *host* program into its local buffer. It then performs its functional task, and sends its results back to the *host* program.

```
Sample Host Program

#include <hpcs/hpcs_defs.h>
main(argc, argv)
int argc;
char *argv[];
{
    /* Establish Connection with HPCS server. */
    fd = hpcs_init(localhost);
    /* Spawn node programs */
    hpcs_spawn(program_name, localhost, number_of_nodes);
    for(i=0; i < number_of_nodes; i++) {
        sprintf(snd_buf,"%d  %s", i, buffer);
        hpcs_write(fd, snd_buffer);
    }
    for(i=0; i < number_of_nodes; i++)
        hpcs_read(fd, recv_buffer);
}

==============================================================
Sample Node Program

#include <hpcs/hpcs_defs.h>
main(argc, argv)
int argc;
char *argv[];
{
    /* Establish Connection with HPCS server. */
    myfd = hpcs_init(host_name);
    hpcs_read(myfd, buffer);
    /* Perform functional task */
    ...
    ...
    hpcs_write(myfd, buffer);
}
```

The details of the primitives used in the above code are described in the previous section of this chapter.

CHAPTER VII

X-WINDOW GUI

The HPCS Process Monitor

The HPCS process monitor was designed to allow the user to have a view of the activity that takes place for his/her program running on the set of workstations.

Currently, it uses the UNIX 'uptime' command to show the load average, and eventually later, will evolve into a performance meter with added tools.

It has been written using Xview, and Xt toolkits available on SUN Sparc's. In the future, a NeXTStep and Motif based monitor will be implemented for use on NeXTStations and DEC UltrixStations.

The menu system is quite simple. One can open a process monitoring connection, by specifying the node IP address and it in turn opens up a tty sub-window with the 'uptime' command running in it. Closing a connection will kill that sub-window specified.

CHAPTER VIII

FUTURE EXPANSIONS

Future Expansions to the GUI

Some suggestions include:

1. Create a time-dependent graph which shows disk I/O, CPU usage, Buffered I/O, Non-buffered I/O usage.

2. Create a Graph Log Analysis option, to allow user to later analyze the performance times of his/her program on the set of workstations.

3. Be able to start the user program through the GUI, observe changes in overall performance times using Line, Bar Graphs.

Future Expansions to HPCS

1. Assign a logical id, to the set of workstations, and configure them with a particular parallel machine model in mind, e.g. Hypercube, Butterfly Network, Token Ring network, 2-D Mesh, de Bruijn Network.

2. Employ proximity analysis and configure workstations to the above parallel models. Here the IP address of the workstations is taken and sorted. The logical id's are then assigned in a serial order to the sorted workstations. Over a Ethernet configuration where the workstations were physically located on the ethernet in the order of their IP address, this method provides a useful means of reducing traffic over the Ethernet communication lines.

3. Employ queues at each server in Central Server Mode (mode 1) and Distributed Server (mode 2), to allow non-blocking sending and receiving of data.

CHAPTER IX

CONCLUSION

HPCS is totally built with a Heterogeneous set of platforms in mind. In this chapter we list briefly the features of HPCS and the advantages in using it:

1. HPCS is based on a connection-oriented protocol (TCP/IP), thus allowing the host and node programs of the user to reside on any workstation in the world, with Internet access.

2. HPCS employs concurrent server mechanisms for the Central Server and Distributed Server modes, increasing response time for each client's request.

3. Employing TCP/IP, allows a HPCS user to assume, that the underlying protocol is a reliable connection-oriented, stream-byte service, which uses timeouts, re-transmission, checksums, error detection mechanisms.

4. All accounts used on the set of workstations, have access defined to allow the host program to spawn the node program on the set of workstations specified.

5. Provides three modes of configuration which can be setup depending on the users needs. The central server and distributed server modes, are client-server models. The client-serve mode is a serverless model.

The Client-Serve mode has the feature of a serverless operation, and clients can communicate *directly* with each other, without any external controlling mechanism.

Appendix A

Central Server Mode Example Program

```
/* HPCS Sample host program for the Reduction problem. */
#include <hpcs/hpcs_defs.h>
main(argc, argv)
int argc;
char *argv[];
{
     /* Establish connection with the HPCS central server */
     int array[16], in_array[4];
     int fd, i, j=0;
     fd = hpcs_init(argv[1]);
     hpcs_spawn("hpcs_node", argv[1], 4);
     ...  /* initialize array contents */
     ...

     /* Data distribution part by host program */
     sprintf(buffer, "%d ", j);
     for(i=0; i < 4; i++) {
         if(i % 4 == 0 || i !=0  ) {
            hpcs_write(fd, buffer);
            j++;  /* Send next partition to next node id */
            sprintf(buffer, "%d ", j);
         }
     strcat(buffer, (char *)&array[i]);
     }

     /* Data collection part by host program */
     for(i=0; i < 4; i++) {
    hpcs_read(fd, buffer);
    in_array[i] = atoi(buffer);
     }
     ...
     ...
     /* re-arrangement of data, find max among four numbers */
     /* Max value is now in in_array[0], print results on screen */
     printf("max = %d \n", in_array[0]);
}

================================================================

/* HPCS Sample node program for the Reduction problem. */
#include <hpcs/hpcs_defs.h>

main(argc, argv)
```

```
int argc;
char *argv[];
{
     /* Establish connection with the HPCS central server */
     int in_array[4];
     int my_fd, k,
     char buffer[MAX_BUF]; /* MAX_BUF defined in hpcs_defs.h */
     my_fd = hpcs_init(argv[1]);

     hpcs_read(fd, buffer);
     /* Scan data in buffer, to in_array[] */
     ...
     ...
     /* Perform internal sort, on in_array[]  */
     ...
     ...

     /* Send in_array[0], containing internal max value,
     back to host program */

     hpcs_write(fd, (char *)&in_array[0]);

     hpcs_close(fd);  /* Close connection with server */
}
```

Appendix B

Distributed-Server Mode Example Program

```
/* Program name: cycle_host */
/* Algorithm: Find a Fundamental Cycle Set in a Graph
              given by its Minimum Cost Spanninf Forest
*/
#include <stdio.h>
#include <hpcs/hpcs_defs.h>
#define CUBE_DIM 2

main(argc, argv)
int argc;
char *argv[];
{
   int fd;
   FILE *fp, *fopen();
   int number_of_edges, edge_number, total_edges;
   char buffer[MAX_BUF];
   int i,k;
   int RET[5][MAX_BUF];
   int *DATA, *START, *END, *LISTA;

   if( (fp = fopen(argv[2], "r")) == NULL)
   {
     printf(stderr, "ERROR: Input file not found");
     return(-1);
   }
   fd = hpcs_init(argv[1]);

Step1:   /* Get Spanning Forest Output from the data file */
   fscanf(fp, "%d %d\n", &number_of_edges, &total_edges);
   DATA = (int *)malloc(2 * sizeof(int) );
   LISTA = (int *)malloc(number_of_edges * sizeof(int) );
   START = (int *)malloc(number_of_edges * sizeof(int) );
   END = (int *)malloc(number_of_edges * sizeof(int) );

   for(k=0; k < number_of_edges; k++)
   {
     fscanf(fp,"%d %d %d\n", &vert1, &vert2, &edge_number);
     START[k] = vert1;
     END[k] = vert2;
     LISTA[k] = edge_number;
   }
   DATA[0] = number_of_edges;
   DATA[1] = total_edges;
```

```
Step2:   /* Send the data to the PE's */
   /* Assuming 2 * CUBE_DIM  PE's */
   for(i=0; i < 2 * CUBE_DIM ; i++)
   {
   sprintf(buffer, "%d ", i);
   strcat(buffer, (char *)&DATA) ;
   hpcs_write(fd, DATA);
   sprintf(buffer, "%d ", i);
   strcat(buffer, (char *)&START) ;
   hpcs_write(fd, START);
   sprintf(buffer, "%d ", i);
   strcat(buffer, (char *)&END) ;
   hpcs_write(fd, END);
   sprintf(buffer, "%d ", i);
   strcat(buffer, (char *)&LISTA) ;
   hpcs_write(fd, LISTA);
   }

Step3:   /* Get results back from the PE's */
   for(k=0; k < 2 * CUBE_DIM ; k++)
     hpcs_read(fd, (char *)&RET[k]);
}

====================================================================

/* Program: cycle_node */
/* HPCS node program for fundamental cycle set algorithm */
#include <stdio.h>
#include <math.h>
#include <hpcs/hpcs_defs.h>

int number_of_edges=0;
int ADJ_MAT[SIZE][SIZE];
#define CUBE_DIM  2

main(argc, argv)
int argc;
char *argv[];
{
   int fd;
   int *ret_val;
   char buffer[MAX_BUF];
   int COTREE[SIZE];
   int i, k;
   int number_of_procs,total_edges;
   int ret, index, length;
   int path[2], size;
   int *DATA, *START, *END, *LISTA;
   DATA = (int *)malloc(2 * sizeof(int) );
   LISTA = (int *)malloc(number_of_edges * sizeof(int) );
   START = (int *)malloc(number_of_edges * sizeof(int) );
   END = (int *)malloc(number_of_edges * sizeof(int) );

Step1:   /* Connect to Server */
   fd = hpcs_init(localhost);

Step2:   /* Read data sent by HOST program */
   hpcs_read(fd, (char *)&DATA) ;
   hpcs_read(fd, (char *)&START) ;
   hpcs_read(fd, (char *)&END) ;
   hpcs_read(fd, (char *)&LISTA) ;

   number_of_procs = 2 * CUBE_DIM;
   number_of_edges = DATA[0];
   total_edges = DATA[1];

Step3:   /* Form the adjacency matrix from data received */
   init_array(COTREE, SIZE);
   init_adj_mat();
   length = sizeof(START)/sizeof(int) - 1;
   for(k=0; k < length; k++)
   {
     put_in_matrix(START[k], START[k]);
     put_in_matrix(END[k], END[k]);
     put_in_matrix(START[k], END[k]);
     put_in_matrix(END[k], START[k]);
   }

Step4:   /* Identify the local co-tree edges */
   ret = (int)ceil( (double)total_edges/(double)number_of_procs);
   for(i=index * ret; i < index * ret + ret; i++)
     COTREE[i] = YES;

   for(i=0; i < number_of_edges; i++)
   {
      if( (index * ret + 1) <= LISTA[i] &&
      LISTA[i] <= (index * ret + ret) )
      COTREE[LISTA[i] - 1] = NO;
   }

Step5:   /* Find the fundamental cycle */
   for(i= index * ret; i < index * ret + ret; i++)
   {
     if(COTREE[i] == YES)
     {
       find_path(i, path, &size);
       print_path(i+1, path, size);
     }
   }
   ret_val = (int *)malloc(sizeof(int));
   ret_val = (int *)&path;

Step6:   /* Send results back to HOST program */
   sprintf(buffer, "0  %s", (char *)&ret_val);
   hpcs_write(fd, buffer);
   exit(0);

}

print_path(id, path, size)
int id, *path, size;
{
   if(size != 0)
      fprintf(stderr, "Edge %d : Forms a F-cycle", id);
   else
   fprintf(stderr, "Edge %d : No existing edge or no path", id);
}

find_path(vt1, path, size)
int vt1, *path, *size;
{
   int h;
   for(h=0; h < number_of_edges; h++)
   {
   if(ADJ_MAT[vt1][h] == TRUE)
   {
      path[0] = vt1 + 1;
      path[1] = h + 1;
```

```
        *size = 2;
        break;
     }
     else *size = 0;
     }
}

init_adj_mat()
{
   int m, n;
   for(m=0; m < SIZE; m++)
    for(n=0; n < SIZE; n++)
      ADJ_MAT[m][n] = 0;

}

put_in_matrix(vert1, vert2)
int vert1, vert2;
{
    ADJ_MAT[vert1-1][vert2-1] = TRUE;
}

print_array(array, size)
int *array, size;
{
   int k;

   for(k=0; k < size; k++)
     printf("elem[%d] = %d", k, array[k]);
}

init_array(array, size)
int *array, size;
{
   int k;

   for(k=0; k < size; k++)
     array[k] = 0;
}
```

Appendix C

Client-Serve Mode Example Program

```
/* Program: HPCS host program for computing the
horizontal Co-occurence values for an image, from
left to right
*/
#include <stdio.h>
#include <hpcs/hpcs_defs.h>

#define NPROCS 8  /* Number of PE's */
#define SZIMG 8 /* Size of Image 8 x 8 */

main(argc, argv)
int argc;
char *argv[];
{
     int fd;
     char buffer[MAX_BUF];
     FILE *fopen(), *fp;
     int image[SZIMG][SZIMG];

     if( (fp = fopen(argv[2], "r")) == NULL) {
          fprintf(stder,"HOST: Input file not found\n");
          exit(-1);
     }

Step1:   /* Connect to local slave server */
     fd = hpcs_init(localhost);

Step2:   /* Spawn node programs on nodes */
     hpcs_spawn("hpcs_node", argv[1], NPROCS);

Step3:   /* Read Image file into image_array[][] */
     for(i=0; i < SZIMG; i++)
          for(j=0; j < SZIMG; j++)
               fscanf(fp,"%d", &image_array[i][j]);

Step4:   /* Send image array to PE's */
     for(j=0; j < NPROCS; j++) {
          sprintf(buffer,"%d %s", j+1, (char *)&image_array[j]);
          hpcs_write(fd, buffer);
     }

Step5:   /* Read co-occurence values from PE's into array */
     for(j=0; j < NPROCS; j++)
```

```
          hpcs_read(fd, (char *)&image_array[j]);

Step6:   /* Write to standard output results */
     for(i=0; i < SZIMG; i++) {
          for(j=0; j < SZIMG; j++)
               fprintf(stdout, "%d ", image_array[i][j]);

          fprintf(stdout, "\n");
     }
}
```

```
============================================================
```

```
/* Program: HPCS node program for computing the
horizontal Co-occurence values for an image, from
left to right
*/
#include <stdio.h>
#include <hpcs/hpcs_defs.h>

#define NPROCS 8  /* Number of PE's */
#define SZIMG 8 /* Size of Image 8 x 8 */

main(argc, argv)
int argc;
char *argv[];
{
     int fd;
     char buffer[MAX_BUF];
     FILE *fopen(), *fp;
     int image[SZIMG];

Step1:   /* Connect with host slave server running on argv[1] */
     fd = hpcs_init(argv[1]);

Step2:   /* Connect with all other clients, gets FD_TABLE[] */
     hpcs_connect_all((atoi(argv[2]));
     hpcs_read(fd, (char *)&image);

Step3:   /* Compute Co-occurence values for each pixel */
     for(i=0; i < SZIMG; i++) {
          for(j=0; j < SZIMG; j++)
               image[i] += image[j];
```

```
            image[i] = image[i] / SZIMG;
        }
Step4:   /* Send Co-occurence values values to host */
        hpcs_write(fd, (char *)&image);
        hpcs_close(fd);
}
```

REFERENCES

[1] Akl, S., *The Design and Analysis of Parallel Algorithms,* Prentice Hall, 1989.

[2] Seitz, C., Jakov Seizovic and Wen-King Su, *"The C Programmer's Abbreviated Guide to Multicomputer Programming",* Technical Report Caltech-CS-TR-88-I, Department of Computer Science, California Institute of Technology, Pasadena, CA, Jan 1988

[3] Mutka, M. and Livny, M., *Profiling Workstations' Available Capacity For Remote Execution,* Proceedings of Performance-87, The 12th IFIP W.G.7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation Brussels, Belgium, December 1987

[4] Litzkow, M., *Remote Unix - Turning Idle Workstations Into Cycle Servers,* Proceedings of the Summer 1987 Usenix Conference Phoenix, Arizona, June 1987

[5] Mutka, M., *Sharing in a Privately Owned Workstation Environment,* Ph.D. Dissertation, University of Wisconsin, Madison, May 1988

[6] Litzkow, M., Livny, M. and Mutka, M., *Condor - A Hunter of Idle Workstations,* Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif. June 1988

[7] Stevens, R., *UNIX Network Programming,* Prentice-Hall, 1989

[8] Stevens, R., *Advanced UNIX Network Programming,* Prentice-Hall, 1991

[9] Stevens, R., Comer, P., *Internetworking with TCP/IP, Volume I,* Prentice-Hall, 1988

[10] Stevens, R., Comer, P., *Internetworking with TCP/IP, Volume II,* Prentice-Hall, 1990

[11] Stevens, R., Comer, P., *Internetworking with TCP/IP, Volume III,* Prentice-Hall, 1992

[12] Begeulin, A., Dongarra, J., Geist, C., Manchek, R., Sunderam, V., *The Parallel Virtual Machine,* Proceedings of the Conference on Distributed Computing, Baton Rouge, Lousiana, 1992

[13] Bertsekas, D. and Tsitsiklis, J., *Parallel and Distributed Computation : Numerical Methods,* Prentice-Hall, 1989

[14] Mano, M., *Computer Systems Architecture,* Prentice-Hall, 1993

[15] Leighton, T., *Introduction to Parallel Algorithms and Architectures : Arrays, Trees and Hypercubes,* Morgan Kaufman Publishers, 1992

[16] JaJa, J., *Introduction to Parallel Algorithms,* Addison-Wesley, 1992

[17] Akl, S. and Lyons, K., *Parallel Computational Geometry,* Prentice-Hall, 1993

[18] Lakshmivarahan, S. and Dhall, S., *Analysis and Design of Parallel Algorithms : Arithmetic and Matrix Problems,* McGraw-Hill, 1990

[19] Harrison, R., *Portable Tools and Applications for Parallel Computers* Prentice-Hall, 1991

[20] Birrel, A., and Nelson, B., *Implementation Remote Procedure Calls* ACM Transactions on Computer Systems, Vol 2, No. 1, Feb. 1984

[21] Zahn, L., et al. *Network Computing Architecture* Prentice-Hall, 1990

[22] Kruskal, J., *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem,* Proceedings of American Mathematics Society, Vol. 7, No. 1, pp. 48-50, 1956.

[23] Nath, D., S.N. Maheshwari and Bhatt, P.C.P., *Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees,* IEEE Trans. on Computing., IEEE, New York, Vol. C-32. No. 6, pp. 569-581, June 1983.

[24] Sollin, M., *An Algorithm Attributed to Sollin in Programming, Games and Transportation Networks,* by C. Berge, and A. Choulia-Houri, Wiley, New York, 1965.

[25] Prim, R., *Shortest Connection Networks and Some Generalizations,* Bell Systems Technical Journal, Vol. 36, pp. 1389-1401, Nov. 1957.

[26] Dijkstra, E.W., *A Note on Two Problems in Connection With Graphs,* Numerisch Math, Vol. 1, No. 5, pp. 269-271, 1981.

[27] Nath, D. and Maheshwari, S. N., *Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems,* Information Processing Letters, Vol. 14, No. 1, pp. 7-11, March 1982.

[28] Tsin, Y., and Chin, F., *Efficient Parallel Algorithms for a Class of Graph Theoretic Problems,* SIAM Journal on Computing, Vol. 13, pp. 580-599, Aug. 1984.

[29] Chin, F., Lam, J. and Chen, I., *Efficient Parallel Algorithm For Some Graph Problem,* Communications of ACM, Vol. 25, pp. 659-665, Sept. 9, 1982.

[30] Quinn, M. J., *Designing Efficient Algortihms for Parallel Computers,* McGraw-Hill Publications, 1987

[31] Das, S. K., Deo, N., Prasad, S., *Parallel Graph Algorithms for Hypercube Computers,* Journal of Parallel Computing, 1990