UNIT 2 PRAM ALGORITHMS

Structure **Page Nos.**

2.0 Introduction 23
2.1 Objectives 23
2.2 Message Passing Programming 23
  2.2.1 Shared Memory
  2.2.2 Message Passing Libraries
  2.2.3 Data Parallel Programming
2.3 Data Structures for Parallel Algorithms 43
  2.3.1 Linked List
  2.3.2 Arrays Pointers
  2.3.3 Hypercube Network
2.4 Summary 47
2.5 Solutions/Answers 47
2.6 References 48

2.0 INTRODUCTION

PRAM (Parallel Random Access Machine) model is one of the most popular models for designing parallel algorithms. A PRAM consists of unbounded number of processors interacting with each other through shared memory and a common communication network. There are many ways to implement the PRAM model. We shall discuss three of them in this unit: message passing, shared memory and data parallel. We shall also cover these models and associated data structures here.

A number of languages and routine libraries have been invented to support these models. Some of them are architecture independent and some are specific to particular platforms. We shall introduce two of the widely accepted routine libraries in this unit. These are Message Passing Interface (MPI) and Parallel Virtual Machine (PVM).

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the concepts of message passing programming;
- list the various communication modes for communication among processors;
- explain the concepts of shared programming model;
- describe and use the functions defined in MPI;
- understand and use functions defined in PVM;
- explain the concepts of data parallel programming, and
- learn about different data structures like array, linked list and hypercube.

2.2 MESSAGE PASSING PROGRAMMING

Message passing is probably the most widely used parallel programming paradigm today. It is the most natural, portable and efficient approach for distributed memory systems. It provides natural synchronisation among the processes so that explicit synchronisation of memory access is redundant. The programmer is responsible for determining all parallelism. In this programming model, multiple processes across the arbitrary number

of machines, each with its own local memory, exchange data through *send and receive* communication between processes. This model can be best understood through the diagram shown in *Figure 1*:

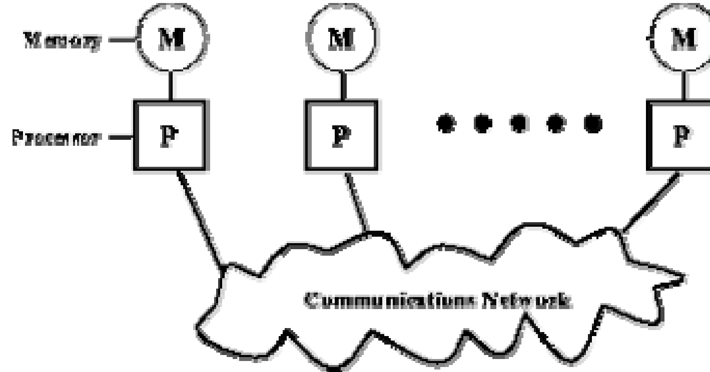


Figure 1: Message passage model

As the diagram indicates, each processor has its own local memory. Processors perform computations with the data in their own memories and interact with the other processors, as and when required, through communication network using message-passing libraries. The message contains the data being sent. But data is not the only constituent of the message. The other components in the message are:

- The identity/address of the processor that sending the message;
- Starting address of the data on the sending processor;
- The type of data being sent;
- The size of data;
- The identity/address of processor(s) are receiving the message, and
- Starting address of storage for the data on the receiving processor.

Once the message has been created it is sent through the communication network. The communication may be in the following two forms:

i) Point-to-point Communication

The simplest form of message is a point-to- point communication. A message is sent from the sending processor to a receiving processor. Message passing in point-to-point communication can be in two modes: synchronous and asynchronous. In synchronous transfer mode, the next message is sent only after the acknowledgement of delivery of the last message. In this mode the sequence of the messages is maintained. In asynchronous transfer mode, no acknowledgement for delivery is required.

ii) Collective Communications

Some message-passing systems allow communication involving more than two processors. Such type of communication may be called collective communication. Collective communication can be in one of these modes:

Barrier: In this mode no actual transfer of data takes place unless all the processors involved in the communication execute a particular block, called barrier block, in their message passing program.

Broadcast: Broadcasting may be one-to-all or all-to-all. In one-to-all broadcasting, one processor sends the same message to several destinations with a single operation whereas

in all-to-all broadcasting, communication takes place in many-to-many fashion. The messages may be personalised or non-personalized. In a personalized broadcasting, unique messages are being sent to every destination processor.

Reduction: In reductions, one member of the group collects data from the other members, reduces them to a single data item which is usually made available to all of the participating processors.

Merits of Message Passage Programming

- Provides excellent low-level control of parallelism;
- Portable;
- Minimal overhead in parallel synchronisation and data distribution; and
- It is less error prone.

Drawbacks

- Message-passing code generally requires more software overhead than parallel shared-memory code.

2.1.1 Shared Memory

In shared memory approach, more focus is on the control parallelism instead of data parallelism. In this model, multiple processes run independently on different processors, but they share a common address space accessible to all as shown in *Figure 2*.

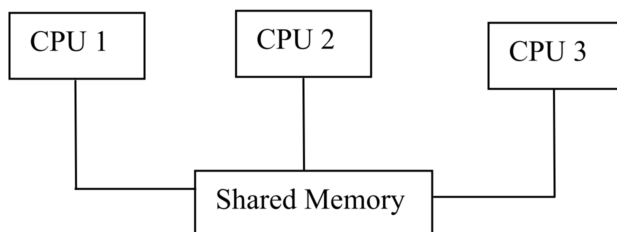


Figure 2: Shared memory

The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. Any change in a memory location effected by one processor is visible to all other processors. As shared data can be accessed by more than one processes at the same time, some control mechanism such as locks/ semaphores should be devised to ensure mutual exclusion. This model is often referred to as SMP (Symmetric Multi Processors), named so because a common symmetric implementation is used for several processors of the same type to access the same shared memory. A number of multi-processor systems implement a shared-memory programming model; examples include: NEC SX-5, SGI Power Onyx/ Origin 2000; Hewlett-Packard V2600/HyperPlex; SUN HPC 10000 400 MHz ;DELL PowerEdge 8450.

Shared memory programming has been implemented in various ways. We are introducing some of them here.

Thread libraries

The most typical representatives of shared memory programming models are thread libraries present in most of the modern operating systems. Examples for thread libraries

are, *POSIX threads* as implemented in Linux, *SolarisTM threads* for solaris , *Win32 threads* available in Windows NT and Windows 2000 , and *JavaTM threads* as part of the standard JavaTM Development Kit (JDK).

Distributed Shared Memory Systems

Distributed Shared Memory (DSM) systems emulate a shared memory abstraction on loosely coupled architectures in order to enable shared memory programming despite missing hardware support. They are mostly implemented in the form of standard libraries and exploit the advanced user–level memory management features present in modern operating systems. Examples include Tread Marks System, IVY, Munin, Brazos, Shasta, and Cashmere.

Program Annotation Packages

A quite renowned approach in this area is OpenMP, a newly developed industry standard for shared memory programming on architectures with uniform memory access characteristics. OpenMP is based on functional parallelism and focuses mostly on the parallelisation of loops. OpenMP implementations use a special compiler to evaluate the annotations in the application's source code and to transform the code into an explicitly parallel code, which can then be executed. We shall have a detailed discussion on OpenMP in the next unit.

Shared memory approach provides low-level control of shared memory system, but they tend to be tedious and error prone. They are more suitable for system programming than to application programming.

Merits of Shared Memory Programming

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between processes is both fast and uniform due to the proximity of memory to CPUs.
- No need to specify explicitly the communication of data between processes.
- Negligible process-communication overhead.
- More intuitive and easier to learn.

Drawbacks

- Not portable.
- Difficult to manage data locality.
- Scalability is limited by the number of access pathways to memory.
- User is responsible for specifying synchronization, e.g., locks.

2.1.2 Message Passing Libraries

In this section, we shall discuss about message passing libraries. Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications. We shall discuss only two worldwide accepted message passing libraries namely; MPI and PVM.

Message Passing Interface (MPI)

The **Message Passing Interface** (**MPI**) is a universal standard for providing communication among the multiple concurrent processes on a distributed memory system. Most, if not all, of the popular parallel computing platforms offer at least one implementation of MPI. It was developed by the MPI forum consisting of several experts

from industry and academics. MPI has been implemented as the library of routines that can be called from languages like, Fortran, C, C++ and Ada programs. MPI was developed in two stages, MPI-1 and MPI-2. MPI-1 was published in 1994.

Features of MPI-1

- Point-to-point communication,
- Collective communication,
- Process groups and communication domains,
- Virtual process topologies, and
- Binding for Fortran and C.

Features added in MPI-2

- Dynamic process management,
- Input/output,
- One-sided operations for remote memory access, and
- Binding for C++.

MPI's advantage over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware it runs on).

Building and Running MPI Programs

MPI parallel programs are written using conventional languages like, Fortran and C .One or more header files such as "mpi.h" may be required to provide the necessary definitions and declarations. Like any other serial program, programs using MPI need to be compiled first before running the program. The command to compile the program may vary according to the compiler being used. If we are using *mpcc* compiler, then we can compile a C program named "program.c" using the following command:

mpcc program.c -o program.o

Most implementations provide command, typically named *mpirun* for spawning MPI processes. It provides facilities for the user to select number of processes and which processors they will run on. For example to run the object file "program" as n processes on n processors we use the following command:

mpirun program -np n

MPI functions

MPI includes hundreds of functions, a small subset of which is sufficient for most practical purposes. We shall discuss some of them in this unit.

Functions for MPI Environment:

int MPI_Init (int *argc, char ** argv)

It initializes the MPI environment. No MPI function can be called before MPI_Init.

int MPI_Finalize (void)

It terminates the MPI environment. No MPI function can be called after MPI_Finalize.

Every MPI process belongs to one or more groups (also called communicator). Each process is identified by its rank (0 to group size –1) within the given group. Initially, all

processes belong to a default group called MPI_COMM_WORLD group. Additional groups can be created by the user as and when required. Now, we shall learn some functions related to communicators.

int MPI_Comm_size (MPI_Comm comm, int *size)

returns variable *size* that contains number of processes in group *comm*.

int MPI_Comm_rank (MPI_Comm comm, int *rank)

returns **rank** of calling process in group *comm*.

Functions for Message Passing:

MPI processes do not share memory space and one process cannot directly access other process's variables. Hence, they need some form of communication among themselves. In MPI environment this communication is in the form of message passing. A message in MPI contains the following fields:

msgaddr: It can be any address in the sender's address space and refers to location in memory where message data begins.

count: Number of occurrences of data items of message datatype contained in message.

datatype: Type of data in message. This field is important in the sense that MPI supports heterogeneous computing and the different nodes may interpret *count* field differently. For example, if the message contains a strings of 2n characters (count =2n), some machines may interpret it having 2n characters and some having n characters depending upon the storage allocated per character (1 or 2). The basic datatype in MPI include all basic types in Fortran and C with two additional types namely MPI_BYTE and MPI_PACKED. MPI_BYTE indicates a byte of 8 bits .

source or *dest* : Rank of sending or receiving process in communicator.

tag: Identifier for specific message or type of message. It allows the programmer to deal with the arrival of message in an orderly way, even if the arrival of the message is not orderly.

comm.: Communicator. It is an object wrapping context and group of a process .It is allocated by system instead of user.

The functions used for messaging passing are:

int MPI_Send(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* can be reused immediately.

int MPI_Recv(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* contains requested message.

MPI message passing may be either point-to-point or collective.

Point-to-point Message Passing

In point-to-point message passing, one process sends/receives message to/from another process. There are four communication modes for sending a message:

i) **Buffered mode:** Send can be initiated whether or not matching receive has been initiated, and send may complete before matching receive is initiated.

ii) **Synchronous mode:** Send can be initiated whether or not matching receive has been initiated, but send will complete only after matching receive has been initiated.

iii) **Ready mode:** Send can be initiated only if matching receive has already been initiated.

iv) **Standard mode:** May behave like either buffered mode or synchronous mode, depending on specific implementation of MPI and availability of memory for buffer space.

MPI provides both blocking and non-blocking send and receive operations for all modes.

Functions for various communication modes

Mode	Blocking	Non-Blocking
Standard	*MPI_Send*	*MPI_Isend*
Buffered	*MPI_Bsend*	*MPI_Ibsend*
Synchronus	*MPI_Ssend*	*MPI_Issend*
Ready	*MPI_Rsend*	*MPI_Irsend*

MPI_Recv and *MPI_Irecv* are blocking and nonblocking functions for receiving messages, regardless of mode.

Besides send and receive functions, MPI provides some more useful functions for communications. Some of them are being introduced here.

MPI_Buffer_attach used to provide buffer space for buffered mode. Nonblocking functions include *request* argument used subsequently to determine whether requested operation has completed.

MPI_Wait and *MPI_Test* wait or test for completion of nonblocking communication.

MPI_Probe and *MPI_Iprobe* probe for incoming message without actually receiving it. Information about message determined by probing can be used to decide how to receive it.

MPI_Cancel cancels outstanding message request, useful for cleanup at the end of a program or after major phase of computation.

Collective Message Passing

In collective message passing, all the processes of a group participate in communication. MPI provides a number of functions to implement the collective message passing. Some of them are being discussed here.

MPI_Bcast(msgaddr, count, datatype, rank, comm):
This function is used by a process ranked *rank* in group *comm* to broadcast the message to all the members (including self) in the group.

MPI_Allreduce

***MPI_Scatter*(Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype, Rank, Comm):**

Using this function process with rank rank *in group comm sends personalized message to all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the send buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.*

MPI_Gather (Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype,Rank, Comm):
Using this function process with rank rank in group comm receives personalized message from all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the receive buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.

MPI_Alltoall()

Each process sends a personalized message to every other process in the group.

MPI_Reduce (Sendaddr , Receiveaddr , count, datatype, op, rank, comm):

This function reduces the partial values stored in Sendaddr of each process into a final result and stores it in Receiveaddr of the process with rank rank. op specifies the reduction operator**.**

***MPI_Scan** (Sendaddr,, Receiveaddr , count, datatype, op, comm):*

It combines the partial values into p final results which are received into the Receiveaddr of all p processes in the group comm.

MPI_Barrier(cmm):
This function synchronises all processes in the group comm.

Timing in MPI program

MPI_Wtime ( ) returns elapsed wall-clock time in seconds since some arbitrary point in past. Elapsed time for program segment is given by the difference between *MPI_Wtime* values at beginning and end of process. Process clocks are not necessarily synchronised, so clock values are not necessarily comparable across the processes, and care must be taken in determining overall running time for parallel program. Even if clocks are explicitly synchronised, variation across clocks still cannot be expected to be significantly less than round-trip time for zero-length message between the processes.

Now, we shall illustrate use of these functions with an example.

Example 1:

```
#include <mpi.h>
int main(int argc, char **argv) {
int i, tmp, sum, s, r, N, x[100];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
If(r==0)
 {
  printf( “Enter N:”);
  scanf (“%d”, &N);
  for (i=1; i<s; i++)
    MPI_Send(&N, 1, MPI_INT,i, i, MPI_COMM_WORLD);
  for (i=r, i<N; i=i+s)
    sum+= x[i];
```

```
 for (i=r, i<s; i++)
 {
   MPI_Recv(&tmp, 1, MPI_INT,i, i,  MPI_COMM_WORLD, &status);
   Sum+=tmp;
}
printf( "%d", sum);
}
else {
        MPI_Recv(&N, 1, MPI_INT,0, i, MPI_COMM_WORLD, &status);
        for (i=r, i<N; i=i+s)
             sum+= x[i];
        MPI_Send(&sum, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
}
MPI_Finalize( );
}
```

Merits of MPI

- Runs on either shared or distributed memory architectures;
- Can be used on a wider range of problems than OpenMP;
- Each process has its own local variables; and
- Distributed memory computers are less expensive than large shared memory computers.

Drawbacks of MPI

- Requires more programming changes to go from serial to parallel version.
- Can be harder to debug, and
- Performance is limited by the communication network between the nodes.

Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a portable message-passing programming system, designed to link separate heterogeneous host machines to form a "virtual machine" which is a single, manageable parallel computing resource. Large computational problems such as molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, can thus be solved more cost effectively by using the aggregate power and memory of many computers.

PVM was developed by the University of Tennessee, The Oak Ridge National Laboratory and Emory University. The first version was released in 1989, version 2 was released in 1991 and finally version 3 was released in 1993. The PVM software enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. It transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures. The programming interface of PVM is very simple .The user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronisation between the tasks. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast and barrier synchronization.

Features of PVM:

- Easy to install;

- Easy to configure;
- Multiple users can each use PVM simultaneously;
- Multiple applications from one user can execute;
- C, C++, and Fortran supported;
- Package is small;
- Users can select the set of machines for a given run of a PVM program;
- Process-based computation;
- Explicit message-passing model, and
- Heterogeneity support.

When the PVM is starts it examines the virtual machine in which it is to operate, and creates a process called the PVM demon, or simply pvmd on each machine. An example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing electronic mail on a computer, pvmd provides inter-host point of contact, authenticates task and executes processes on machines. It also provides the fault detection, routes messages not from or intended for its host, transmits messages from its application to a destination, receives messages from the other pvmd's, and buffers it until the destination application can handle it.

PVM provides a library of functions, libpvm3.a, that the application programmer calls. Each function has some particular effect in the PVM. However, all this library really provides is a convenient way of asking the local pvmd to perform some work. The pvmd then acts as the virtual machine. Both pvmd and PVM library constitute the PVM system.

The PVM system supports functional as well as data decomposition model of parallel programming. It binds with C, C++, and Fortran . The C and C++ language bindings for the PVM user interface library are implemented as functions (subroutines in case of FORTRAN) . User programs written in C and C++ can access PVM library by linking the library libpvm3.a (libfpvm3.a in case of FORTRAN).

All PVM tasks are uniquely identified by an integer called *task identifier* (TID) assigned by local pvmd. Messages are sent to and received from tids. PVM contains several routines that return TID values so that the user application can identify other tasks in the system. PVM also supports grouping of tasks. A task may belong to more than one group and one task from one group can communicate with the task in the other groups. To use any of the group functions, a program must be linked with `libgpvm3.a.`

Set up to Use PVM

PVM uses two environment variables when starting and running. Each PVM user needs to set these two variables to use PVM. The first variable is PVM_ROOT, which is set to the location of the installed `pvm3` directory. The second variable is PVM_ARCH , which tells PVM the architecture of this host. The easiest method is to set these two variables in your`.`cshrc file. Here is an example for setting PVM_ROOT:

```
setenv PVM_ROOT $HOME/pvm3
```

The user can set PVM_ARCH by concatenating to the file `.cshrc`, the content of file `$PVM_ROOT/lib/cshrc.stub`.

Starting PVM

To start PVM, on any host on which PVM has been installed we can type

```
% pvm
```

The PVM console, called pvm, is a stand-alone PVM task that allows the user to interactively start, query, and modify the virtual machine. Then we can add hosts to virtual machine by typing at the console prompt (got after last command)

```
pvm> add hostname
```

To delete hosts (except the one we are using) from virtual machine we can type

```
pvm> delete hostname
```

We can see the configuration of the present virtual machine, we can type

```
pvm> conf
```

To see what PVM tasks are running on the virtual machine, we should type

```
pvm> ps -a
```

To close the virtual machine environment, we should type

```
pvm> halt
```

Multiple hosts can be added simultaneously by typing the hostnames in a file one per line and then type

```
% pvm hostfile
```

PVM will then add all the listed hosts simultaneously before the console prompt appears.

Compiling and Running the PVM Program

Now, we shall learn how to compile and run PVM programs. To compile the program , change to the directory pvm/lib/archname where archname is the architecture name of your computer. Then the following command:

```
cc  program.c -lpvm3 -o prgram
```

will compile a program called program.c. After compiling, we must put the executable file in the directory pvm3/bin/ARCH. Also, we need to compile the program separately for every architecture in virtual machine. In case we use dynamic groups, we should also add -lgpvm3 to the compile command. The executable file can then be run. To do this, first run PVM. After PVM is running, executable file may be run from the unix command line, like any other program.

PVM supplies an architecture-independent make, aimk, that automatically determines PVM_ARCH and links any operating system specific libraries to your application. To compile the C example, type

```
% aimk master.c
```

Now, from one window, start PVM and configure some hosts. In another window change directory to $HOME/pvm3/bin/PVM_ARCH and type

```
% master
```

It will ask for a number of tasks to be executed. Then type the number of tasks.

Programming with PVM

The general method for writing a program with PVM is as follows:
A user writes one or more sequential programs in C, C++, or Fortran 77 containing embedded PVM function (or subroutine) calls. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem.

PVM library routines

In this section we shall give a brief description of the routines in the PVM 3 user library. Every PVM program should include the PVM header file "`pvm3.h`" (in a C program) or "fpvm3.h" (in a Fortran program).

In PVM 3, all PVM tasks are identified by an integer supplied by the local pvmd. In the following descriptions this task identifier is called TID. Now we are introducing some commonly used functions in PVM programming (as in C. For Fortran, we use prefix pvmf against pvm in C).

Process Management

- int **pvm_mytid(void)**

Returns the *tid* of the calling process. **tid** values less than zero indicate an error.

- **int pvm_exit(void)**

Tells the local pvmd that this process is leaving PVM. **info** Integer status code returned by the routine. Values less than zero indicate an error.

- **pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)**

start new PVM processes. **task**, a character string is the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. **Argv is a** pointer to an array of arguments to **task**. If the executable needs no arguments, then the second argument to pvm_spawn is NULL. **flag** Integer specifies spawn options. **where** , a character string specifying where to start the PVM process. If *flag* is 0, then *where* is ignored and PVM will select the most appropriate host. **ntask** ,an integer, specifies the number of copies of the executable to start. **tids** ,Integer array of length *ntask* returns the tids of the PVM processes started by this pvm_spawn call. The function returns the actual number of processes returned. Negative values indicate error.

- **int pvm_kill(int tid)**

Terminates a specified PVM process. **tid** Integer task identifier of the PVM process to be killed (not itself). Return values less than zero indicate an error.

- **int pvm_catchout(FILE *ff)**

Catch output from child tasks. **ff** is file descriptor on which we write the collected output. The default is to have the PVM write the *stderr* and *stdout* of spawned tasks.

Information

- **int pvm_parent(void)**

Returns the tid of the process that spawned the calling process.

- **int pvm_tidtohost(tid)**

Returns the host of the specified PVM process. Error if negative value is returned.

- **int pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)**

```
struct pvmhostinfo {
int hi_tid;
char *hi_name;
char *hi_arch;
int hi_speed;
};
```

Returns information about the present virtual machine configuration. **nhost** is the number of hosts (pvmds) in the virtual machine. **narch** is the number of different data formats being used. **hostp** is pointer to an array of structures which contains the information about each host including its pvmd task ID, name, architecture, and relative speed(default is 1000).

- **int info = pvm_tasks(int where, int *ntask, struct pvmtaskinfo **taskp)**

```
struct pvmtaskinfo {
                int ti_tid; int ti_ptid;
                int ti_host;
                int ti_flag; char *ti_a_out; } taskp;
```

Returns the information about the tasks running on the virtual machine. **where** specifies what tasks to return the information about. The options are:

0

for all the tasks on the virtual machine

pvmd tid

for all tasks on a given host

tid

for a specific task

ntask returns the number of tasks being reported on.

taskp is a pointer to an array of structures which contains the information about each task including its task ID, parent tid, pvmd task ID, status flag, and the name of this task's executable file. The status flag values are: waiting for a message, waiting for the pvmd, and running.

Dynamic Configuration

- **int pvm_addhosts(char **hosts, int nhost, int *infos)**
  Add hosts to the virtual machine. **hosts** is an array of strings naming the hosts to be added. **nhost** specifies the length of array hosts. **infos** is an array of length *nhost* which returns the status for each host. Values less than zero indicate an error, while positive values are TIDs of the new hosts.
  Signaling

- **int pvm_sendsig(int tid, int signum)**

Sends a signal to another PVM process. **tid** is task identifier of PVM process to receive the signal. **signum** is the signal number.

- **int info = pvm_notify(int what, int msgtag, int cnt, int *tids)**
  Request notification of PVM event such as host failure. **What** specifies the type of event to trigger the notification. Some of them are:

PvmTaskExit
    Task exits or is killed.
PvmHostDelete
    Host is deleted or crashes.
PvmHostAdd
    New host is added.

msgtag is message tag to be used in notification. **cnt** For *PvmTaskExit* and *PvmHostDelete*, specifies the length of the *tids* array. For *PvmHostAdd* specifies the number of times to notify.

tids for *PvmTaskExit* and *PvmHostDelete* is an array of length *cnt* of task or pvmd TIDs to be notified about. The array is not used with the *PvmHostAdd* option.

Message Passing

The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse, and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. There are options in PVM 3 that request that data be transferred directly from task to task. In this case, if the message is large, the sender may block until the receiver has called a matching receive.

A nonblocking receive immediately returns with either the data or a flag that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer. In addition to these point-to-point communication functions, the model supports the multicast to a set of tasks and the broadcast to a user-defined group of tasks. There are also functions to perform global max, global sum, etc. across a user-defined group of tasks. Wildcards can be specified in the receive for the source and the label, allowing either or both of these contexts to be ignored. A routine can be called to return the information about the received messages.

The PVM model guarantees that the message order is preserved. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both the messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

- **int bufid = pvm_mkbuf(int encoding)**

creates a new message buffer. **encoding** specifying the buffer's encoding scheme.

Encoding data options

PVM uses SUN's XDR library to create a machine independent data format if you request it. Settings for the encoding option are:

PvmDataDefault: Use XDR by default, as the local library cannot know in advance where you are going to send the data.

PvmDataRaw: No encoding, so make sure you are sending to a similar machine.

PvmDataInPlace: Not only is there no encoding, but the data is not even going to be physically copied into the buffer.

- **int info = pvm_freebuf(int bufid)**

Disposes of a message buffer. **bufid** message buffer identifier.

- **int pvm_getsbuf(void)**

returns the active send buffer identifier.

- **int pvm_getrbuf(void)**

returns the active receive buffer identifier.

- **int pvm_setsbuf(int bufid)**

sets the active send buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **int pvm_setrbuf(int bufid)**

sets the active receive buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **intpvm_initsend(int encoding)**

Clear default sends buffer and specifies the message encoding. **Encoding** specifies the next message's encoding scheme.

- **int pvm_send(int tid, int msgtag)**

Immediately sends the data in the active message buffer. **tid** Integer task identifier of destination process. **msgtag** Integer message tag supplied by the user. msgtag should be nonnegative.

- **int pvm_recv(int tid, int msgtag)**

Receive a message. **tid** is integer task identifier of sending process supplied by the user and **msgtag** is the message tag supplied by the user(should be non negative integer). The process returns the value of the new active receive buffer identifier. Values less than zero indicate an error. It blocks the process until a message with label *msgtag* has arrived from *tid*. pvm_recv then places the message in a new *active* receive buffer, which also clears the current receive buffer.

Packing and Unpacking Data

- pvm_packs - Pack the active message buffer with arrays of prescribed data type:

- **int info = pvm_packf(const char *fmt, ...)**
- **int info = pvm_pkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_pkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_pkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_pkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_pkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_pkint(int *ip, int nitem, int stride)**
- **int info = pvm_pkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_pkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_pkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_pklong(long *ip, int nitem, int stride)**
- **int info = pvm_pkshort(short *jp, int nitem, int stride)**
- **int info = pvm_pkstr(char *sp)**

fmt Printf-like format expression specifying what to pack. **nitem** is the total number of *items* to be packed (not the number of bytes). **stride** is the stride to be used when packing the items.

- pvm_unpack - Unpacks the active message buffer into arrays of prescribed data type. It has been implemented for different data types:

- **int info = pvm_unpackf(const char *fmt, ...)**
- **int info = pvm_upkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_upkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_upkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_upkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_upkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_upkint(int *ip, int nitem, int stride)**
- **int info = pvm_upkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_upkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_upkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_upklong(long *ip, int nitem, int stride)**
- **int info = pvm_upkshort(short *jp, int nitem, int stride)**
- **int info = pvm_upkstr(char *sp)**

Each of the pvm_upk* routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, *nitem* which is the total number of items to unpack, and *stride* which is the stride to use when unpacking. An exception is pvm_upkstr() which by definition unpacks a NULL terminated character string and thus does not need *nitem* or *stride* arguments.

Dynamic Process Groups

To create and manage dynamic groups, a separate library `libgpvm3.a` must be linked with the user programs that make use of any of the group functions. Group management work is handled by a group server that is automatically started when the first group function is invoked. Any PVM task can join or leave any group dynamically at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not members. Now we are giving some routines that handle dynamic processes:

- **int pvm_joingroup(char *group)**

Enrolls the calling process in a named group. **group** is a group name of an existing group. Returns instance number.Instance numbers run from 0 to the number of group members minus 1. In PVM 3, a task can join multiple groups. If a process leaves a group and then rejoins it, that process may receive a different instance number.

- **int info = pvm_lvgroup(char *group)**

Unenrolls the calling process from a named group.

- **int pvm_gettid(char *group, int inum)**

Returns the tid of the process identified by a group name and instance number.

- **int pvm_getinst(char *group, int tid)**

Returns the instance number in a group of a PVM process.

- **int size = pvm_gsize(char *group)**

Returns the number of members presently in the named group.

- **int pvm_barrier(char *group, int count)**

Blocks the calling process until all the processes in a group have called it. **count** species the number of group members that must call pvm_barrier before they are all released.

- **int pvm_bcast(char *group, int msgtag)**

Broadcasts the data in the active message buffer to a group of processes. **msgtag** is a message tag supplied by the user. It allows the user's program to distinguish between different kinds of messages .It should be a nonnegative integer.

- **int info = pvm_reduce(void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootginst)**

Performs a reduce operation over members of the specified group. **func** is function defining the operation performed on the global data. Predefined are PvmMax, PvmMin, PvmSum and PvmProduct. Users can define their own function. **data** is pointer to the starting address of an array of local values. **count** species the number of elements of *datatype* in the data array. **Datatype** is the type of the entries in the data array. **msgtag** is the message tag supplied by the user. msgtag should be greater than zero. It allows the user's program to distinguish between different kinds of messages. **group** is the group name of an existing group. **rootginst** is the instance number of group member who gets the result.

We are writing here a program that illustrates the use of these functions in the parallel programming:

Example 2: Hello.c

```
#include "pvm3.h"
main()
{
   int cc, tid, msgtag;
   char buf[100];

   printf("%x\n", pvm_mytid());

   cc = pvm_spawn("hello_other", (char**)0, 0, "", 1,
&tid);

   if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
   } else
      printf("can't start hello_other\n");

   pvm_exit();
}
```

In this program, pvm_mytid( ), returns the TID of the running program (In this case, task id of the program hello.c). This program is intended to be invoked manually; after

printing its task id (obtained with `pvm_mytid()`), it initiates a copy of another program called *hello_other* using the `pvm_spawn()` function. A successful spawn causes the program to execute a blocking receive using `pvm_recv`. After receiving the message, the program prints the message sent by its counterpart, as well its task id; the buffer is extracted from the message using `pvm_upkstr`. The final `pvm_exit` call dissociates the program from the PVM system.

hello_other.c

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the ``slave'' or spawned program; its first PVM action is to obtain the task id of the ``master'' using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

☞ **Check Your Progress 1**

1) Write a program to give a listing of the "slave" or spawned program.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
…………

2.2.3 Data Parallel Programming

In the data parallel programming model, major focus is on performing simultaneous operations on a data set. The data set is typically organized into a common structure, such as an array or hypercube. Processors work collectively on the same data structure. However, each task works on a different partition of the same data structure. It is *more restrictive* because not all algorithms can be specified in the data-parallel terms. For these reasons, data parallelism, although important, is not a universal parallel programming paradigm.

Programming with the data parallel model is usually accomplished by writing a program with the data parallel constructs. The constructs can be called to a data parallel subroutine

library or compiler directives. Data parallel languages provide facilities to specify the data decomposition and mapping to the processors. The languages include data distribution statements, which allow the programmer to control which data goes on what processor to minimize the amount of communication between the processors. Directives indicate how arrays are to be aligned and distributed over the processors and hence specify agglomeration and mapping. Communication operations are not specified explicitly by the programmer, but are instead inferred by the compiler from the program. Data parallel languages are more suitable for SIMD architecture though some languages for MIMD structure have also been implemented. Data parallel approach is more effective for highly regular problems, but are not very effective for irregular problems.

The main languages used for this are Fortran 90, High Performance Fortran (HPF) and HPC++. We shall discuss HPF in detail in the next unit. Now, we shall give a brief overview of some of the early data parallel languages:

- ***Computational Fluid Dynamics:*** CFD was a FORTRAN-like language developed in the early 70s at the Computational Fluid Dynamics Branch of Ames Research Center for ILLIAC IV machines, a SIMD computer for array processing. The language design was extremely pragmatic. No attempt was made to hide the hardware peculiarities from the user; in fact, every attempt was made to give the programmers the access and control of all of the hardware to help constructing efficient programs. This language made the architectural features of the ILLIAC IV very apparent to the programmer, but it also contained the seeds of some practical programming language abstractions for data-parallel programming. In spite of its simplicity and ad hoc machine-dependencies, CFD allowed the researchers at Ames to develop a range of application programs that efficiently used the ILLIAC IV.

- ***Connection Machine Fortran:*** Connection Machine Fortran was a later SIMD language developed by Thinking Machines Corporation. Connection Machine Fortran included all of FORTRAN 77, together with the new array syntax of Fortran 90. It added various machine specific features, but unlike CFD or DAP FORTRAN these appeared as *compiler directives* rather than special syntax in Fortran declarations or executable statements. A major improvement over the previous languages was that, distributed array dimensions were no longer constrained to exactly fit in the size of the processing element array; the compiler could transparently map dimensions of arbitrary extent across the available processor grid dimensions. Finally the language added an explicitly parallel looping construct called FORALL. Although CM Fortran looked syntactically like standard Fortran, the programmer had to be aware of many nuances. Like the ILLIAC IV, the Connection

Machine allowed the Fortran arrays to either be distributed across the processing nodes (called *CM arrays*, or distributed arrays), or allocated in the memory of the front-end computer (called *front-end arrays*, or sequential arrays). Unlike the control unit of the ILLIAC, the Connection Machine front-end was a conventional, general-purpose computer--typically a VAX or Sun. But there were still significant restrictions on how arrays could be manipulated, reflecting the two possible homes.

Glypnir ,IVTRAN and *LISP are some of the other early data parallel languages.

Let us conclude this unit with the introduction of a typical data parallel programming style called **SPMD.**

Single Program Multiple Date

A common style of writing data parallel programs for MIMD computers is SPMD (single program, multiple data): all the processors execute the same program, but each operates

on a different portion of problem data. It is easier to program than true MIMD, but more flexible than SIMD. Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style. In this style, although there is no central controller, the worker nodes carry on doing *essentially* the same thing at *essentially* the same time. Instead of central copies of control variables stored on the control processor of a SIMD computer, control variables (iteration counts and so on) are usually stored in a replicated fashion across MIMD nodes. Each node has its own local copy of these global control variables, but every node updates them in an identical way. There are no centrally issued parallel instructions, but communications usually happen in the well-defined collective phases. These data exchanges occur in a prefixed manner that explicitly or implicitly synchronize the peer nodes. The situation is something like an orchestra without a conductor. There is no central control, but each individual plays from the same script. The group as a whole stays in lockstep. This loosely synchronous style has some similarities to the Bulk Synchronous Parallel (BSP) model of computing introduced by the theorist Les Valiant in the early 1990s. The restricted pattern of the collective synchronization is easier to deal with than the complex synchronisation problems of a general concurrent programming.

A natural assumption was that it should be possible and not too difficult to capture the SPMD model for programming MIMD computers in data-parallel languages, along lines similar to the successful SIMD languages. Various research prototype languages attempted to do this, with some success. By the 90s the value of portable, standarised programming languages was universally recognized, and there seemed to be some consensus about what a standard language for SPMD programming ought to look like. Then the High Performance Fortran (HPF) standard was introduced.

2.3 DATA STRUCTURES FOR PARALLEL ALGORITHMS

To implement any algorithm, selection of a proper data structure is very important. A particular operation may be performed with a data structure in a smaller time but it may require a very large time in some other data structure. For example, to access i^{th} element in a set may need constant time if we are using arrays but the required time becomes a polynomial in case of a linked list. So, the selection of data structure must be done keeping in mind the type of operation to be performed and the architecture available. In this section, we shall introduce some data structures commonly used in a parallel programming.

2.3.1 Linked List

A linked list is a data structure composed of zero or more nodes linked by pointers. Each node consists of two parts, as shown in *Figure 3*: *info* field containing specific information and *next* field containing address of next node. First node is pointed by an external pointer called *head*. Last node called tail node does not contain address of any node. Hence, its next field points to null. Linked list with zero nodes is called null linked list.

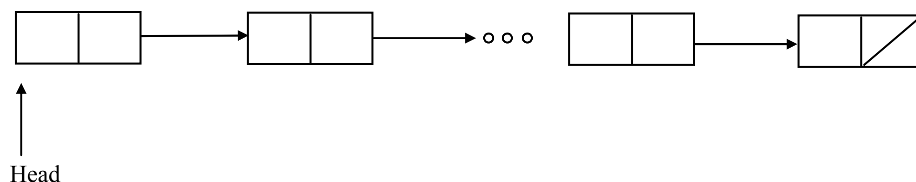


Figure 3: Linked List

A large number of operations can be performed using the linked list. For some of the operations like insertion or deletion of the new data, linked list takes constant time, but it is time consuming for some other operations like searching a data. We are giving here an example where linked list is used:

Example 3:
Given a linear linked list, rank the list elements in terms of the distance from each to the last element.

A parallel algorithm for this problem is given here. The algorithm assumes there are p number of processors.

Algorithm:

```
Processor j, 0≤ j<p, do
  if next[j]=j then
    rank[j]=0
   else rank[j] =1
 endif
 while rank[next[first]]≠0 Processor j, 0≤ j<p, do
      rank[j]=rank[j]+rank[next[j]]
      next[j]=next[next[j]]
 endwhile
```

The working of this algorithm is illustrated by the following diagram:

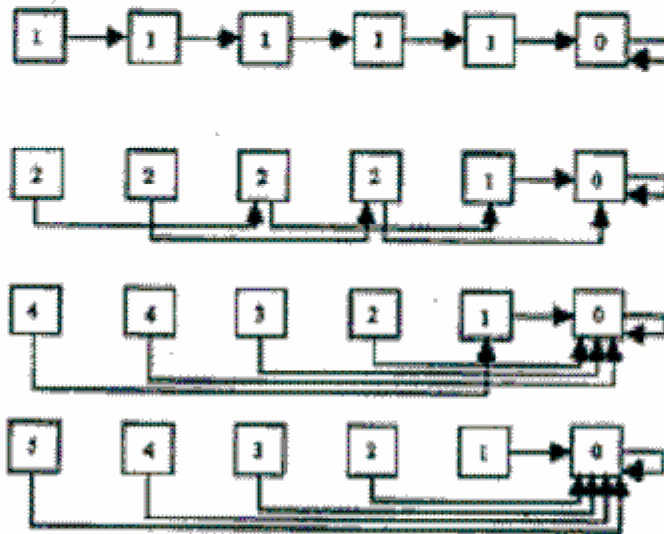


Figure 4 : Finding rank of elements

2.3.2 Arrays Pointers

An array is a collection of the similar type of data. Arrays are very popular data structures in parallel programming due to their easiness of declaration and use. At the one hand, arrays can be used as a common memory resource for the shared memory programming, on the other hand they can be easily partitioned into sub-arrays for data parallel programming. This is the flexibility of the arrays that makes them most

frequently used data structure in parallel programming. We shall study arrays in the context of two languages Fortran 90 and C.

Consider the array shown below. The size of the array is 10.

5	10	15	20	25	30	35	40	45	50

Index of the first element in Fortran 90 is 1 but that in C is 0 and consequently the index of the last element in Fortran 90 is 10 and that in C is 9. If we assign the name of array as A, then i^{th} element in Fortran 90 is A(i) but in C it is A[i-1]. Arrays may be one-dimensional or they may be multi-dimensional.

General form of declaration of array in Fortran 90 is

```
type, DIMENSION(bound) [,attr] :: name
```

for example the declaration

```
INTEGER, DIMENSION(5): A
```

declare an array A of size 5.

General form of declaration of array in C is

type array_name [size]

For example the declaration A

int A[10]

declares an array of size 10.

Fortran 90 allows one to use particular sections of an array. To access a section of an array, you need the name of the array followed by the two integer values separated by a colon enclosed in the parentheses. The integer values represent the indices of the section required.
For example, a(3:5) refers to elements 3, 4, 5 of the array, a(1:5:2) refers to elements 1, 3, 5 of the array , and b(1:3, 2:4) refers to the elements from rows 1 to 3 and columns 2 to 4. In C there is only one kind of array whose size is determined statically, though there are provisions for dynamic allocation of storage through pointers and dynamic memory allocation functions like *calloc* and *malloc* functions. In Fortran 90, there are 3 possible types of arrays depending on the binding of an array to an amount of storage : *Static arrays* with fixed size at the time of declaration and cannot be altered during execution ; *Semi-dynamic arrays* or *automatic arrays*: the size is determined after entering a subroutine and arrays can be created to match the exact size required, but local to a subroutine ; and *Dynamic arrays* or *allocatable arrays* : the size can be altered during execution.

In these languages, array operations are written in a compact form that often makes programs more readable.
Consider the loop:

```
s=0
do i=1,n
   a(i)=b(i)+c(i)
   s=s+a(i)
end do
```

It can be written (in Fortran 90 notation) as follows:

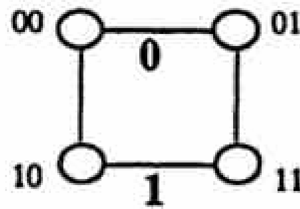
```
a(1:n) = b(1:n) +c(1:n)
s=sum(a(1:n))
```

In addition to Fortran 90 , there are many languages that provide succinct operations on arrays. Some of the most popular are APL, and MATLAB. Although these languages were not developed for parallel computing, rather for expressiveness, they can be used to express parallelism since array operations can be easily executed in parallel. Thus, all the arithmetic operations (+, -, * /, **) involved in a vector expression can be performed in parallel. Intrinsic reduction functions, such as the sum above, also can be performed in a parallel .

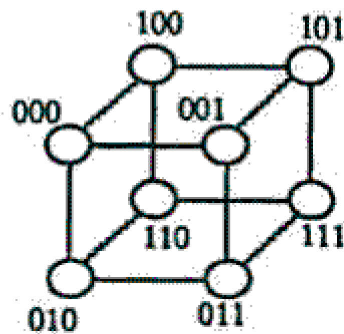
2.3.3 Hypercube Network

The hypercube architecture has played an important role in the development of parallel processing and is still quite popular and influential. The highly symmetric recursive structure of the hypercube supports a variety of elegant and efficient parallel algorithms. Hypercubes are also called n-cubes, where n indicates the number of dimensions. An-cube can be defined recursively as depicted below:

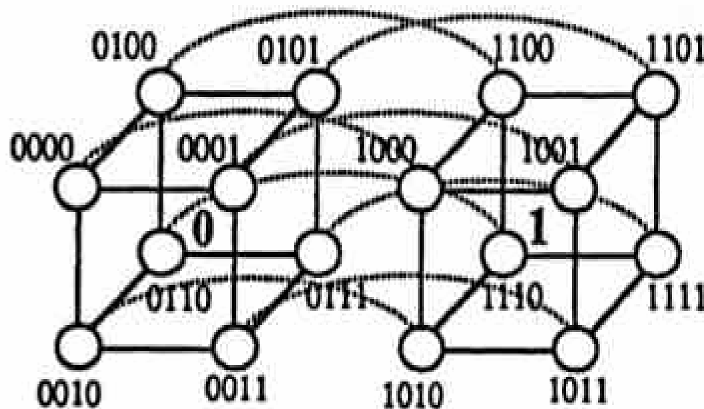
1-cube built of 2 0-cubes
Figure 5(a): 1-cube



2-cube built of 2 1-cubes
Figure 5(b): 2-cube



3-cube built of 2 2-cubes
Figure 5(c): 3-cube



4-cube built of 2 3-cubes

Figure 5(d): 4-cube

Properties of Hypercube:

- A node p in a n-cube has a unique label, its binary ID, that is a n-bit binary number.
- The labels of any two neighboring nodes differ in exactly 1 bit.
- Two nodes whose labels differ in *k* bits are connected by a shortest path of length *k*.
- Hypercube is both node- and edge- symmetric.

Hypercube structure can be used to implement many parallel algorithms requiring all-to-all communication, that is, algorithms in which each task must communicate with every other task. This structure allows a computation requiring all-to-all communication among P tasks to be performed in just log P steps compared t polynomial time using other data structures like arrays and linked lists.

2.4 SUMMARY

In this unit, a number of concepts have been introduced in context designing of algorithms for PRAM model of parallel computation. The concepts introduced include message passing programming, data parallel programming, message passing interface (MPI), and parallel virtual machine. Also, topics relating to modes of communication between processors, the functions defined in MPI and PVM are discussed in sufficient details.

2.5 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1) ***hello_other.c***

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the "slave" or spawned program; its first PVM action is to obtain the task id of the ``master" using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and

architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

2.6 REFERENCES/FURTHER READINGS

1) Kai Hwang: *Advanced Computer Architecture*: Parallelism, Scalability, Programmability (2001), Tata McGraw Hill, 2001.

2) Henessy J. L. and Patterson D. A. *Computer Architecture*: A Qualititative Approach, Morgan Kaufman (1990)

3) Rajaraman V. and Shive Ram Murthy C. *Parallel Computer: Architecture and Programming*: Prentice Hall of India

4) Salim G. *Parallel Computation, Models and Methods*: Akl Prentice Hall of India