Practical Workbook
CS-451
Parallel Processing
(BCIT)

Name :

Year :

Batch :

Roll No :

Department:

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology,

INTRODUCTION

Parallel processing has been in common use for decades. To deal with all types of grand challenges we must need High Performance Computing / Parallel Processing. Vary broadly; Parallel Processing can be achieved by two approaches, Hardware Approach and Software Approach and this Lab manual of Parallel Processing has been designed accordingly.

In Software approach, a parallel resource administration and management software is configured on systems / computers connected through LAN. All the hardware and software resources are combined logically to present a single system image to the programmer and user. Clusters and Grids are the examples of this approach. This environment is also refereed to as Distributed Memory environment. All nodes / processors have their own local memories. The processors can communicate and share the resources using Message Passing. This approach is cheap and scalable but difficult to program, manage and secure.

In Hardware approach, all the processors or execution units are placed on the same motherboard sharing a common memory and other resources on the board. This approach is referred to as Shared Memory Architecture. This approach is expensive but much faster and easy to program. SMPs and Multi Core processors are examples of this system.

Programming a parallel system is not as easy as programming single processor systems. There are many considerations like details of the underlying parallel system, processors interconnection, use of the correct parallel programming model and selection of parallel language which makes the parallel programming more difficult. This lab manual is focused on writing parallel algorithms and their programming on distributed and shared memory environments.

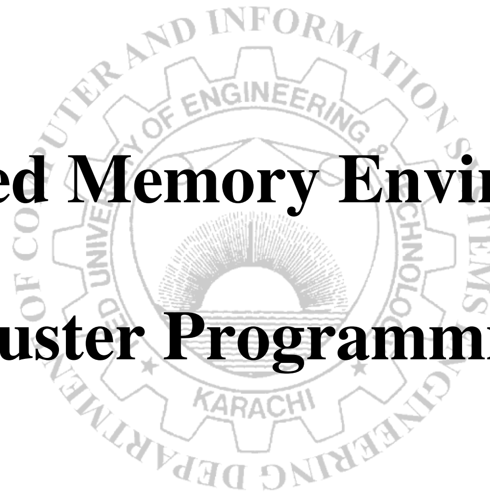
Part one of this lab manual is based on Cluster Programming. A four node Linux based cluster using MPICH is used for programming. First Lab starts with the basics of MPI and MPICH. The next two labs proceed with the communication among the parallel MPI processes. Third, fourth and fifth labs deal with the MPI collective operations. In the final lab some non blocking parallel operations are explored.

Part two of this lab manual deal with SMP and Multi-Core systems programming. Intel Dual processors systems and Intel Quad core systems are the targeted platforms. This section starts with the introduction of Shared Memory Architectures and OpenMP API for windows. Rest of laboratory sessions are based on the theory and implementation of OpenMP directives and their clauses. Environment variables related to the OpenMP API are also discussed in the end.

CONTENTS

Lab Session No.	Object	Page No
	Part One: Distributed Memory Environments / Cluster Programming	
	Introduction	1
1	Basics of MPI (Message Passing Interface)	5
2	To learn Communication between MPI processes	10
3	To get familiarized with advance communication between MPI processes	15
4	Study of MPI collective operations using 'Synchronization'	21
5	Study of MPI collective operations using 'Data Movement'	23
6	Study of MPI collective operations using 'Collective Computation'	30
7	To understand MPI Non-Blocking operation	37
	Part Two: Shared Memory Environments / SMP Programming	
	Introduction	44
8	Basics of OpenMP API (Open Multi-Processor API)	49
9	To get familiarized with OpenMP Directives	55
10	Sharing of work among threads using Loop Construct in OpenMP	61
11	Clauses in Loop Construct	65
12	Sharing of work among threads in an OpenMP program using 'Sections Construct'	74
13	Sharing of work among threads in an OpenMP program using 'Single Construct'	78
14	Use of Environment Variables in OpenMP API	82

Part One



Distributed Memory Environments / Cluster Programming

Introduction

As parallel computers started getting larger, scalability considerations resulted in a pure distributed memory model. In this model, each CPU has local memory associated with it, and there is no shared memory in the system. This architecture is scalable since, with every additional CPU in the system, there is additional memory local to that CPU, which in turn does not present a bandwidth bottleneck for communication between CPUs and memory. On such systems, the only way for tasks running on distinct CPUs to communicate is for them to explicitly send and receive messages to and from other tasks called Message Passing. Message passing languages grew in popularity very quickly and a few of them have emerged as standards in the recent years. This section discusses some of the more popular distributed memory environments.

1. Ada

Ada is a programming language originally designed to support the construction of long-lived, highly reliable software systems. It was developed for the U.S. Department of Defense for real-time embedded systems. Inter task communication in Ada is based on the rendezvous mechanism. The tasks can be created explicitly or declared statically. A task must have a specification part which declares the entries for the rendezvous mechanisms. It must also have a body part, defined separately, which contains the accept statements for the entries, data, and the code local to the task. Ada uses the select statement for expressing non determinism. The select statement allows the selection of one among several alternatives, where the alternatives are prefixed by guards. Guards are boolean expressions that establish the conditions that must be true for the corresponding alternative to be a candidate for execution. Another distinguishing feature of Ada is its exception handling mechanism to deal with software errors. The disadvantages of Ada are that it does not provide a way to map tasks onto CPUs.

2. Parallel Virtual Machine

Parallel Virtual Machine, or PVM, was the first widely accepted message passing environment that provided portability and interoperability across heterogeneous platforms. The first version was developed at Oak Ridge National Laboratory in the early 1990s, and there have been several versions since then. PVM allows a network of heterogeneous computers to be used as a single computational resource called the parallel virtual machine. The PVM environment Consists of there parts on all the computers in the parallel virtual machine, a library of PVM interface functions, and a PVM console to interactively start, query and modify the virtual machine. Before running a PVM application, a user needs to start a PVM daemon on each machine thus creating a parallel virtual machine. The PVM application needs to be linked with the PVM library, which contains functions for point to point communication, collective communication, dynamic task spawning, task coordination, modification of the virtual machine etc. This application can be started from any of the computers in the virtual machine at the shell prompt or from the PVM console. The biggest advantages of PVM are its portability and interoperability. Not only the same PVM program run on any platforms on which it is supported, tasks from the same program can run on different platforms at the same time as part of the same program. Furthermore,

different vendor's PVM implementations can also talk to each other, because of a well-defined inter-PVM daemon protocol. Thus, a PVM application can have tasks running on a cluster of machines, of different types, and running different PVM implementations. Another notable point about PVM is that it provides the programmer with great flexibility for dynamically changing the virtual machine, spawning tasks, and forming groups. It also provides support for fault tolerance and-Load balancing.

The main disadvantage of PVM is that its performance is not as good as other message passing systems such as MPI. This is mainly because PVM sacrifices performance for flexibility. PVM was quickly embraced by many programmers as their preferred parallel programming environment when it was released for public use, particularly by those who were interested in using a network of computers "and those who programmed on many different platforms, since this paradigm helped them write on program that would run on almost any platform. The public domain implementation works for almost any UNIX platform, and Windows/NT implementations have also been added.

3. Distributed Computing Environment

Distributed Computing Environment or DCE, is a suite of technologies available from The Open Group, a consortium of computer users and vendors interested in advancing open systems technology. DCE enables the development of distributed applications across heterogeneous systems. The three areas of computing in which DCE is most useful are security, internet/intranet computing, and distributed objects. DCE provides six classes of service. It provides a threads service at the lowest level, to allow multiple threads of execution. Above this layer, it provides a remote procedure call (RPC) service which facilitates client-server communication across a network. Sitting on top of the RPC service are time and directory services that synchronize system clocks and provide a single programming model throughout the network, respectively. The next service is a distributed file service, providing access to files across a network including diskless support. Orthogonal to these services is DCE' security service, which, authenticates the identities of users, authorizes access to resources in the network and provides user and server account management. DCE is available from several vendors including Digital, HP, IBM, Silicon Graphs, and Tandem Computers. It is being used extensively in a wide variety of industries including automotive and financial service, telecommunication engineering, government and academia.

3.1 Distributed Java

The popularity of the java language stems largely from its capability and suitability for writing programs that use and interact with resources on the internet in particular, and clusters of heterogeneous computers in general. The basic Java package, the Java Development Kit or JDK, supports many varieties of distributed memory paradigms corresponding to various levels of abstraction. Additionally, several accessory paradigms have been developed for different kinds of distributed computing using Java, although these do not belong to the JDK.

3.2 Sockets

At the lowest level of abstraction, Java provides socket APIs through its set of socket-related classes. The Socket and Server Socket classes provide APIs for stream or TCP sockets, and the Datagram Socket, Datagram Packet and Multicast Socket classes provide APIs for datagram or UDP sockets. Each of these classes has several methods that provide the corresponding APIs.

3.3 Remote Method Invocation

Just like RPCs provide a higher level of abstraction than sockets. Remote Method Invocation or RMI, provides a paradigm for communication between program-level objects residing in different address spaces. RMI allows a Java program to invoke methods of remote Java objects in other Java virtual machines, which could be running on different hosts. A local stub object manages the invocation of remote object methods. RMI employs object serialization to marshal and unmarshal parameters of these calls. Object serialization is a specification by which objects can be encoded into a stream of bytes, and then reconstructed back from the stream. The stream includes sufficient information to restore the fields in the stream to compatible versions of the class. To provide RMI support, Java employs a distributed object model which differs from the base object model in several ways, including: non-remote arguments to and results from an RMI an: passed by copy rather than by reference a remote object is passed by reference and not by copying the actual remote implementation; clients of remote objects interact with remote interfaces, and not with their implementation classes .

3.4 URLS

At a very high level of abstraction, the Java runtime provides classes via which a. program can access resources on another machine in the network. Through the DRL and URL Connection classes, a .Java program an access a resource on the network by specifying its address in a from of a uniform resource locator. A program can also use the URL connection class to connect to a resource on the network. Once the connection is established, actions such as reading from or writing to the connection can be performed.

3.5 Java Space

The Java Space paradigm is an extension of the Linda concept. It creates a shared memory space called a tuple space, which is used as a storage repository for data to and from distinct tasks; the Java Space model provides a medium for RMI-capable applications and hardware to share work and results over a distributed environment. A key attribute of a Java Space is that it can store not only data but serialized objects, which could be combinations of data and methods that can be invoked on any machine supporting the Java runtime. Hence, a Java Space entry can be transferred across machines while retaining its original behavior, achieving distributed object persistence. Analogous to the Linda model, the Java Space paradigm attempts to raise the level of abstraction for the programmer so they can create completely distributed applications without considering details such as hardware and location.

4. Message Passing Interface

The Message Passing Interface or MPI is a standard for message passing that has been developed by a consortium consisting of representatives from research laboratories, universities, and industry. The first version MPI-l was standardized In 1994, and the second version MPI-2 was developed in 1997. MPI is an explicit message passing paradigm where tasks communicate with each other by sending messages.

The two main objectives of MPI are portability and high performance. The MPI environment consists of an MPI library that provides a rich set of functions numbering in the hundreds. MPI defines the concept of communicators which combine message context and task group to provide message security. Intra-communicators allow safe message passing within a group of tasks. MPI provides many different flavors of both blocking and non-blocking point to point communication primitives, and has support for structured buffers and derived data types. It also provides many different types of collective communication routines for communication, between tasks belonging to a group. Other functions include those for application-oriented task topologies, profiling, and environmental query and control functions. MPI-2 also adds dynamic spawning of MPI tasks to this impressive list of functions.

5. JMPI

The MPI-2 specification includes bindings for FORTRAN, C, and C++ languages. However, no binding for Java is planned by the MPI Forum. JMPI is an effort underway at MPI Software Technology Inc. to integrate MPI with Java. JMPI is different from other such efforts in that, where possible; the use of native methods has been avoided for the MPI implementation. Native methods are those that are written in a language other than Java, such as C, C++, or assembly. The use of native methods in Java programs may be necessitated in situations where some platform-dependent feature may be needed, or there may be a need to use existing programs written in another language from a Java application. Minimizing the use of native methods in a Java programs makes the program more portable. JMPI also inlc1udes an optional communication layer that is tightly integrated with the Java Native Interface, which is the native programming interface for Java that is part of the Java Development Kit (JDK). This layer enables vendors to seamlessly implement their own native message passing schemes in, a way that is compatible with the Java programming model. Another characteristic of JMPI is that it only implements MPI functionality deemed essential for commercial customers.

6. JPVM

JPVM is an API written using the Java native methods capability so that Java applications can use the PVM software. JPVM extends the capabilities of PVM to the Java platform, allowing Java applications and existing C, C++, and FORTRAN applications to communicate with each other via the PVM API.

Lab Session 1

OBJECT

Basics of MPI (Message Passing Interface)

THEORY

MPI - Message Passing Interface

The Message Passing Interface or MPI is a standard for message passing that has been developed by a consortium consisting of representatives from research laboratories, universities, and industry. The first version MPI-l was standardized in 1994, and the second version MPI-2 was developed in 1997. MPI is an explicit message passing paradigm where tasks communicate with each other by sending messages.

The two main objectives of MPI are portability and high performance. The MPI environment consists of an MPI library that provides a rich set of functions numbering in the hundreds. MPI defines the concept of communicators which combine message context and task group to provide message security. Intra-communicators allow safe message passing within a group of tasks, and intercommunicates allow safe message passing between two groups of tasks. MPI provides many different flavors of both blocking and non-blocking point to point communication primitives, and has support for structured buffers and derived data types. It also provides many different types of collective communication routines for communication, between tasks belonging to a group. Other functions include those for application-oriented task topologies, profiling, and environmental query and control functions. MPI-2 also adds dynamic spawning of MPI tasks to this impressive list of functions.

Key Points:

- MPI is a library, not a language.
- MPI is a specification , not a particular implementation
- MPI addresses the message passing model.

Implementation of MPI: MPICH

MPICH is one of the complete implementation of the MPI specification, designed to be both portable and efficient. The ``CH" in MPICH stands for ``Chameleon," symbol of adaptability to one's environment and thus of portability. Chameleons are fast, and from the beginning a secondary goal was to give up as little efficiency as possible for the portability.
MPICH is a unified source distribution, supporting most flavors of Unix and recent versions of Windows. In additional, binary distributions are available for Windows platforms.

Structure of MPI Program:

```
#include <mpi.h>
int main(int argc, char ** argv)
    //Serial Code
    {
        MPI_Init(&argc,&argv);
            //Parallel Code
        MPI_Finalize();
    //Serial Code
    }
```

A simple MPI program contains a main program in which parallel code of program is placed between MPI_Init and MPI_Finalize.

- **MPI_Init**

  It is used initializes the parallel code segment. Always use to declare the start of the parallel code segment.

  ```
  int MPI_Init( int* argc ptr /* in/out */ ,char** argv
  ptr[ ] /* in/out */)
  ```

  or simply

  ```
  MPI_Init(&argc,&argv)
  ```

- **MPI Finalize**

  It is used to declare the end of the parallel code segment. It is important to note that it takes no arguments.

  ```
  int MPI Finalize(void)
  ```

  or simply

  ```
  MPI_Finalize()
  ```

Key Points:

- Must include mpi.h by introducing its header #include<mpi.h>. This provides us with the function declarations for all MPI functions.
- A program must have a beginning and an ending. The beginning is in the form of an *MPI_Init()* call, which indicates to the operating system that this is an MPI program and allows the OS to do any necessary initialization. The ending is in the form of an *MPI_Finalize()* call, which indicates to the OS that “clean-up” with respect to MPI can commence.

- If the program is embarrassingly parallel, then the operations done between the MPI initialization and finalization involve no communication.

Predefined Variable Types in MPI

MPI DATA TYPE	***C DATA TYPE***
MPI_CHAR	Signed Char
MPI_SHORT	Singed Short Int
(Cont.)	
MPI_INT	Signed Int
MPI_LONG	Singed Long Int
MPI_UNSIGNED_CHAR	Unsigned Char
MPI_UNSIGNED_SHORT	Unsigned Short Int
MPI_UNSIGNED	Unsigned Int
MPI_UNSIGNED_LONG	Unsigned Long Int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long Double
MPI_BYTE	-------------
MPI_PACKED	-------------

Our First MPI Program:

```
#include <iostream.h>
#include <mpi.h>
int main(int argc, char ** argv)
     {
          MPI_Init(&argc,&argv);
          cout << "Hello World!" << endl;
          MPI_Finalize();
     }
```

On compile and running of the above program, a collection of "Hello World!" messages will be printed to your screen equal to the number of processes on which you ran the program despite there is only one print statement.

Compilation and Execution of a Program:

For Compilation on Linux terminal, ***mpicc - o*** *{object name} {file name with c extension}*
For Execution on Linux terminal, ***mpirun –np {*** *number of process}* ***program name***

Determining the Number of Processors and their IDs

There are two important commands very commonly used in MPI:

- **MPI Comm rank:** It provides you with your process identification or rank (Which is an integer ranging from 0 to P − 1, where P is the number of processes on which are running),

```
int MPI_Comm_rank(MPI Comm comm /* in */,int* result /* out */)
```

or simply

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank)
```

- **MPI Comm size:** It provides you with the total number of processes that have been allocated.

```
int MPI_Comm_size( MPI Comm comm /* in */,int*  size /* out */)
```

or simply

```
MPI_Comm_size(MPI_COMM_WORLD,&mysize)
```

The argument ***comm*** is called the communicator, and it essentially is a designation for a collection of processes which can communicate with each other. MPI has functionality to allow you to specify varies communicators (differing collections of processes); however, generally ***MPI_COMM_WORLD***, which is predefined within MPI and consists of all the processes initiated when a parallel program, is used.

An Example Program:

```
#include <iostream.h>
#include <mpi.h>
int main(int argc, char ** argv)
     {
     int mynode, totalnodes;
          MPI_Init(&argc,&argv);
               MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
               MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
               cout << "Hello world from process " << mynode;
               cout << " of " << totalnodes << endl;
          MPI_Finalize();
     }
```

When run with four processes, the screen output may look like:

Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 2 of 4
Hello world from process 1 of 4

Key Point:

The output to the screen may not be ordered correctly since all processes are trying to write to the screen at the same time, and the operating system has to decide on an ordering. However, the thing to notice is that each process called out with its process identification number and the total number of MPI processes of which it was a part.

Exercises:

1. Re-Code given exercise program on page no 16 using pure C Language syntax.

Code:

____
____
____
____
____
____
____
____
____
____
____
____
____

2. Write a program that prints "*I am even*" for nodes whom rank is divisible by two and print "I am odd" for the other odd ranking nodes

Program:

____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____

Lab Session 2

OBJECT

To learn Communication between MPI processes

THEORY

It is important to observe that when a program running with MPI, all processes use the same compiled binary, and hence all processes are running the exact same code. What in an MPI distinguishes a parallel program running on P processors from the serial version of the code running on P processors? Two things distinguish the parallel program:

- Each process uses its process rank to determine what part of the algorithm instructions are meant for it.
- Processes communicate with each other in order to accomplish the final task.

Even though each process receives an identical copy of the instructions to be executed, this does not imply that all processes will execute the same instructions. Because each process is able to obtain its process rank (using MPI_Comm_rank).It can determine which part of the code it is supposed to run. This is accomplished through the use of IF statements. Code that is meant to be run by one particular process should be enclosed within an IF statement, which verifies the process identification number of the process. If the code is not placed with in IF statements specific to a particular id, then the code will be executed by all processes.

The second point, communicating between processes; MPI communication can be summed up in the concept of sending and receiving messages. Sending and receiving is done with the following two functions: MPI Send and MPI Recv.

- **MPI_Send**

```
int MPI_Send( void* message /* in */, int count /* in */, MPI
Datatype datatype /* in */, int dest /* in */, int tag /* in
*/, MPI Comm comm /* in */ )
```

- **MPI_Recv**

```
int MPI_Recv( void* message /* out */, int count /* in */, MPI
Datatype datatype /* in */, int source /* in */, int tag /* in
*/, MPI Comm comm /* in */, MPI Status* status /* out */)
```

Understanding the Argument Lists

- **message** - starting address of the send/recv buffer.

- **count** - number of elements in the send/recv buffer.
- **datatype** - data type of the elements in the send buffer.
- **source** - process rank to send the data.
- **dest -** process rank to receive the data.
- **tag** - message tag.
- **comm** - communicator.
- **status** - status object.

An Example Program:

The following program demonstrate the use of send/receive function in which sender is initialized as node two (2) where as receiver is assigned as node four (4). The following program requires that it should be accommodated on five (5) nodes otherwise the sender and receiver should be initialized to suitable ranks.

```
#include <iostream.h>
#include <mpi.h>

int main(int argc, char ** argv
{
     int mynode, totalnodes;
     int datasize; // number of data units to be sent/recv
     int sender=2; // process number of the sending process
     int receiver=4; // process number of the receiving process
     int tag; // integer message tag
     MPI_Status   status;   //   variable   to   contain   status
     information

     MPI_Init(&argc,&argv);

     MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
     MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

     // Determine datasize
     double * databuffer = new double[datasize];
     //  Fill  in  sender,  receiver,  tag  on  sender/receiver
     processes,
     // and fill in databuffer on the sender process.

     if(mynode==sender)

     MPI_Send(databuffer,datasize,MPI_DOUBLE,receiver,
     tag,MPI_COMM_WORLD);

     if(mynode==receiver)
        MPI_Recv(databuffer,datasize,MPI_DOUBLE,sender,tag,
     MPI_COMM_WORLD,&status);
     // Send/Recv complete
```

```
      MPI_Finalize();
}
```

Key Points:

- In general, the *message* array for both the sender and receiver should be of the same type and both of same size at least *datasize*.
- In most cases the *sendtype* and *recvtype* are identical.
- The tag can be any integer between 0-32767.
- *MPI Recv* may use for the tag the wildcard *MPI ANY TAG*. This allows an MPI Recv to receive from a send using any tag.
- *MPI Send* cannot use the wildcard *MPI ANY TAG*. A special tag must be specified.
- *MPI Recv* may use for the source the wildcard *MPI ANY SOURCE*. This allows an MPI Recv to receive from a send from any source.
- MPI Send must specify the process rank of the destination. No wildcard exists.

An Example Program: To calculate the sum of given numbers in parallel:

The following program calculates the sum of numbers from 1 to 1000 in a parallel fashion while executing on all the cluster nodes and providing the result at the end on only one node. It should be noted that the print statement for the sum is only executed on the node that is ranked zero (0) otherwise the statement would be printed as much time as the number of nodes in the cluster.

```
#include<iostream.h>
#include<mpi.h>

int main(int argc, char ** argv)
{
      int mynode, totalnodes;
      int sum,startval,endval,accum;
      MPI_Status status;

      MPI_Init(argc,argv);

      MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
      MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
            sum = 0;
            startval = 1000*mynode/totalnodes+1;
            endval = 1000*(mynode+1)/totalnodes;
            for(int i=startval;i<=endval;i=i+1)
            sum = sum + i;
            if(mynode!=0)
            MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
            else
            for(int j=1;j<totalnodes;j=j+1)
            {
```

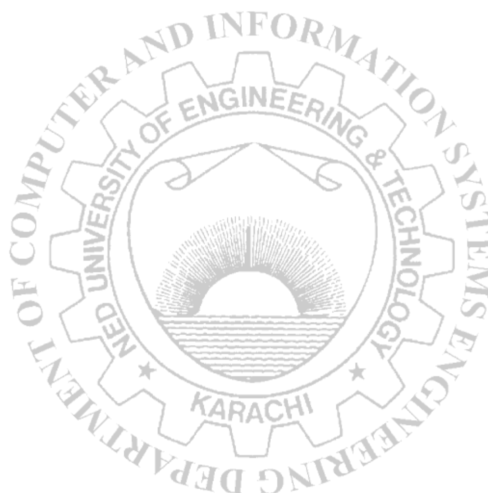
```
            MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD,
            &status);
            sum = sum + accum;
            }
            if(mynode == 0)
            cout << "The sum from 1 to 1000 is: " <<  sum <<
            endl;
            MPI_Finalize();
}
```

Exercise:

1. Code the above example program in C that calculates the sum of numbers in parallel on different numbers of nodes. Also calculate the execution time.
[Note: You have to use time stamp function to also print the time at begging and end of parallel code segment]

Output: (On Single Node)

Execution Time:



Output: (On Two Nodes)

Execution Time:

Speedup:

Output: (On Four Nodes)

Execution Time:

Speedup:

Output: (On Sixteen Nodes)

Execution Time:

Speedup:

2. Suppose you are in a scenario where you have to transmit an array buffer from all other nodes to one node by using send/ receive functions that are used for intra-process synchronous communication. The figure below demonstrates the required functionality of the program.

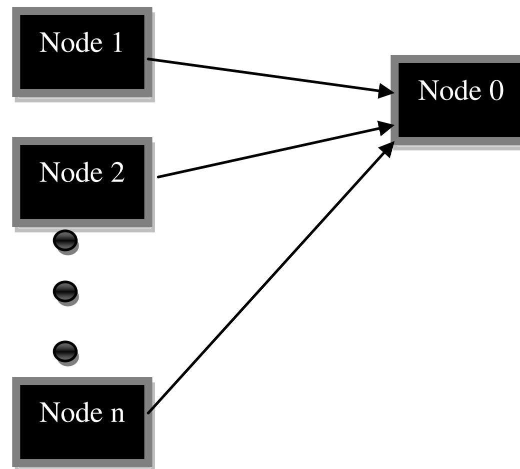


Figure 2.1

Lab Session 3

OBJECT

To get familiarized with advance communication between MPI processes

THEORY

This Lab session will focus on more information about sending and receiving in MPI like sending of arrays and simultaneous send and receive

Key Points

- Whenever you send and receive data, MPI assumes that you have provided non overlapping positions in memory. As discussed in the previous lab session, *MPI_COMM_WORLD* is referred to as a **communicator**. In general, a communicator is a collection of processes that can send messages to each other. *MPI_COMM_WORLD* is pre-defined in all implementations of MPI, and it consists of all MPI processes running after the initial execution of the program.

- In the send/receive, we are required to use a *tag*. The tag variable is used to distinguish upon receipt between two messages sent by the same process.

- The order of sending does not necessarily guarantee the order of receiving. Tags are used to distinguish between messages. MPI allows the tag *MPI_ANY_TAG* which can be used by *MPI_Recv* to accept any valid tag from a sender but you *cannot* use *MPI_ANY_TAG* in the *MPI_Send* command.

- Similar to the *MPI_ ANY_ TAG* wildcard for tags, there is also an *MPI_ANY_SOURCE* wildcard that can also be used by *MPI_Recv*. By using it in an *MPI_Recv*, a process is ready to receive from any sending process. Again, you *cannot* use *MPI_ ANY_ SOURCE* in the *MPI_ Send* command. There is no wildcard for sender destinations.

- When you pass an array to *MPI_ Send/MPI_Recv*, it need not have exactly the number of items to be sent – *it must have greater than or equal to the number of items to be sent*. Suppose, for example, that you had an array of 100 items, but you only wanted to send the first ten items, you can do so by passing the array to *MPI_Send* and only stating that ten items are to be sent.

An Example Program:

In the following MPI code, array on each process is created, initialize it on process 0. Once the array has been initialized on process 0, then it is sent out to each process.

```
#include<iostream.h>
#include<mpi.h>
```

```
int main(int argc, char * argv[])
{
      int i;
      int nitems = 10;
      int mynode, totalnodes;
      MPI_Status status;
      double * array;

      MPI_Init(&argc,&argv);
           MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
           MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
           array = new double[nitems];
           if(mynode == 0)
           {
           for(i=0;i<nitems;i++)
           array[i] = (double) i;
           }
           if(mynode==0)
           for(i=1;i<totalnodes;i++)
           MPI_Send(array,nitems,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
           else
           MPI_Recv(array,nitems,MPI_DOUBLE,0,1,MPI_COMM_WORLD,
           &status);
           for(i=0;i<nitems;i++)
               {
                cout << "Processor " << mynode;
                cout << ": array[" << i << "] = " << array[i] <<
               endl;
               }
           MPI_Finalize();
}
```

Key Points:

- *An array is* created, on each process, using dynamic memory allocation.
- On process 0 only (i.e., mynode == 0), an array is initialized to contain the ascending index values.
- On process 0, program proceeds with (totalnodes-1) calls to *MPI Send*.
- On all other processes other than 0, MPI_*Recv is* called *to* receive the sent message.
- On each individual process, the results are printed of the sending/receiving pair.

Simultaneous Send and Receive, MPI_Sendrecv:

The subroutine MPI_Sendrecv exchanges messages with another process. A send-receive operation is useful for avoiding some kinds of unsafe interaction patterns and for implementing remote procedure calls. A message sent by a send-receive operation can be

received by MPI_Recv and a send-receive operation can receive a message sent by an MPI_Send.

MPI_Sendrecv(&data_to_send, send_count, send_type, destination_ID, send_tag, &received_data, receive_count, receive_type, sender_ID, receive_tag, comm, &status)

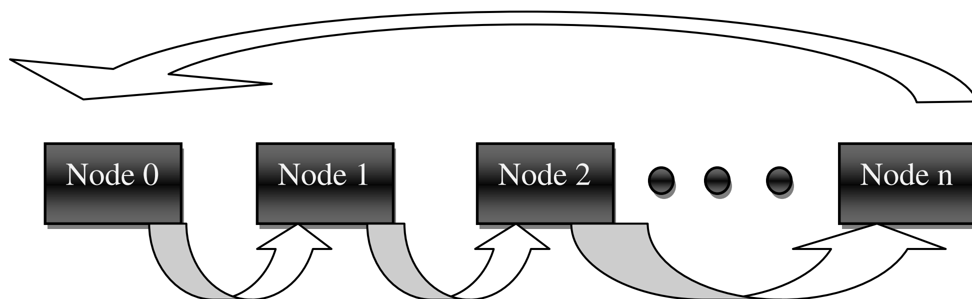
Understanding the Argument Lists

- **data_to_send**: variable of a C type that corresponds to the MPI send_type supplied below
- **send_count:** number of data elements to send (int)
- **send_type:** datatype of elements to send (one of the MPI datatype handles)
- **destination_ID**: process ID of the destination (int)
- **send_tag**: send tag (int)
- **received_data**: variable of a C type that corresponds to the MPI receive_type supplied below
- **receive_count**: number of data elements to receive (int)
- **receive_type:** datatype of elements to receive (one of the MPI datatype handles)
- **sender_ID**: process ID of the sender (int)
- **receive_tag**: receive tag (int)
- **comm**: communicator (handle)
- **status**: status object (MPI_Status)

It should be noted in above stated arguments that they contain all the arguments that were declared in send and receive functions separately in the previous lab session

Exercise:

1. **Write a program in which every node receives from its left node and sends message to its right node simultaneously as depicted in the following figure**



Program:

____
____
____
____
____
____
____
____

2. Write a program to calculate prefix sum S_n of 'n' numbers on 'n' processes

- **Each node has two variables 'a' and 'b'.**
- **Initially a=node id**
- **Each node sends 'a' to the other node.**
- **'b' is a variable that receives a sent from another node.**

3. Write a parallel program that calculates the sum of array and execute it for different numbers of nodes in the cluster. Also calculate their respective execution time.

Program Code:

Output: (On Single Node)

Execution Time:

Output: (On Two Nodes)

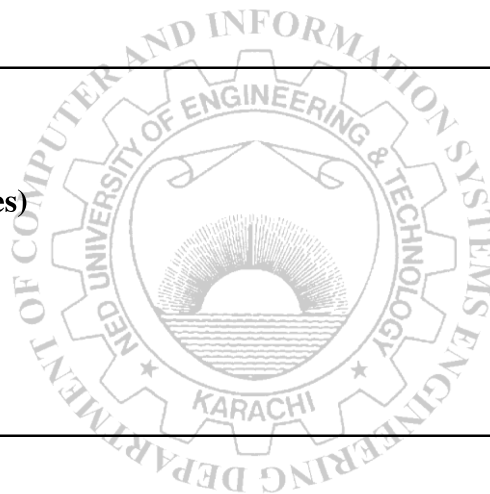
Execution Time:

Speedup:

Output: (On Four Nodes)

Execution Time:

Speedup:



Output: (On Sixteen Nodes)

Execution Time:

Speedup:

Lab Session 4

OBJECT

Study of MPI collective operations using 'Synchronization'

THEORY

Collective operations

MPI_Send and MPI_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operations are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission. MPI supports three classes of collective operations:

- Synchronization,
- Data Movement, and
- Collective Computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

Synchronization

The MPI_Barrier function can be used to synchronize a group of processes. To synchronize a group of processes, each one must call MPI_Barrier when it has reached a point where it can go no further until it knows that all its partners have reached the same point. Once a process has called MPI_Barrier, it will be blocked until all processes in the group have also called MPI_Barrier.

- **MPI Barrier**

```
int MPI_Barrier( MPI Comm comm /* in */  )
```

Understanding the Argument Lists

- *comm* - communicator

Example of Usage

```
int mynode, totalnodes;

MPI_Init(&argc,&argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

MPI_Barrier(MPI_COMM_WORLD);
// At this stage, all processes are synchronized
```

Key Point

- This command is a useful tool to help insure synchronization between processes. For example, you may want all processes to wait until one particular process has read in data from disk. Each process would call *MPI_Barrier* in the place in the program where the synchronization is required.

Exercise:

1. Write a parallel program, after discussing your instructor which uses MPI_Barrier

Program:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 5

OBJECT

Study of MPI collective operations using 'Data Movement'

THEORY

Collective operations

MPI_Send and MPI_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operations are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission. MPI supports three classes of collective operations:

- Synchronization,
- Data Movement, and
- Collective Computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

Collective data movement

There are several routines for performing collective data distribution tasks:

- **MPI_Bcast**, The subroutine MPI_Bcast sends a message from one process to all processes in a communicator.

- **MPI_Gather, MPI_Gatherv**, Gather data from participating processes into a single structure

- **MPI_Scatter, MPI_Scatterv,** Break a structure into portions and distribute those portions to other processes

- **MPI_Allgather, MPI_Allgatherv**, Gather data from different processes into a single structure that is then sent to all participants (Gather-to-all)

- **MPI_Alltoall, MPI_Alltoallv**, Gather data and then scatter it to all participants (All-to-all scatter/gather)

  The routines with "V" suffixes move variable-sized blocks of data.

MPI_Bcast

- The subroutine MPI_Bcast sends a message from one process to all processes in a communicator.

- In a program all processes must execute a call to MPI_BCAST. There is no separate MPI call to receive a broadcast.

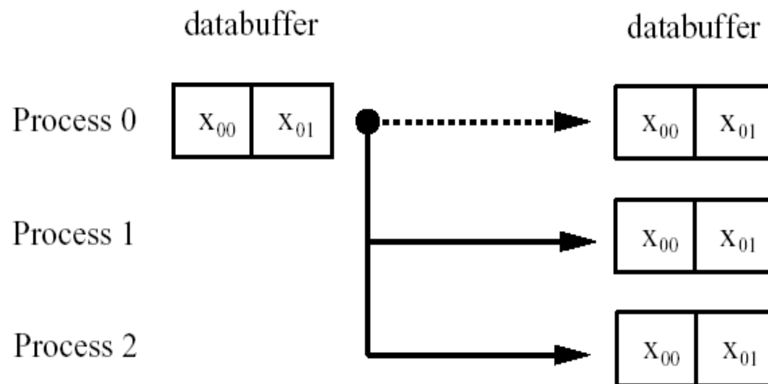


Figure 5.1 *MPI Bcast* schematic demonstrating a broadcast of two data objects from process zero to all other processes.

```
int MPI Bcast( void* buffer /* in/out */, int count /* in */,
MPI Datatype datatype /* in */,
          int root /* in */, MPI Comm comm /* in */)
```

Understanding the Argument List

- **buffer** - starting address of the send buffer.
- **count** - number of elements in the send buffer.
- **datatype** - data type of the elements in the send buffer.
- **root** - rank of the process broadcasting its data.
- **comm** - communicator.

MPI_Bcast broadcasts a message from the process with rank "root" to all other processes of the group.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units to be broadcast
int root; // process which is broadcasting its data

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
```

```
// Determine datasize and root
double * databuffer = new double[datasize];
// Fill in databuffer array with data to be broadcast

MPI_Bcast(databuffer,datasize,MPI_DOUBLE,root,MPI_COMM_WORLD);

// At this point, every process has received into the
// databuffer array the data from process root
```

Key Point

- Each process will make an identical call of the *MPI Bcast* function. On the broadcasting (root) process, the *buffer* array contains the data to be broadcast. At the conclusion of the call, all processes have obtained a copy of the contents of the *buffer* array from process root.

MPI_Scatter:

MPI_Scatter is one of the most frequently used functions of MPI Programming. Break a structure into portions and distribute those portions to other processes. Suppose you are going to distribute an array elements equally to all other nodes in the cluster by decomposing the main array into its sub segments which are then distributed to the nodes for parallel computation of array segments on different cluster nodes.

```
int MPI_Scatter
      (
     void *send_data,
     int send_count,
     MPI_Datatype send_type,
     void  *receive_data,
     int receive_count,
     MPI_Datatype receive_type,
     int sending_process_ID,
     MPI_Comm comm.
     )
```

MPI_Gather

- Gather data from participating processes into a single structure
- Synopsis:

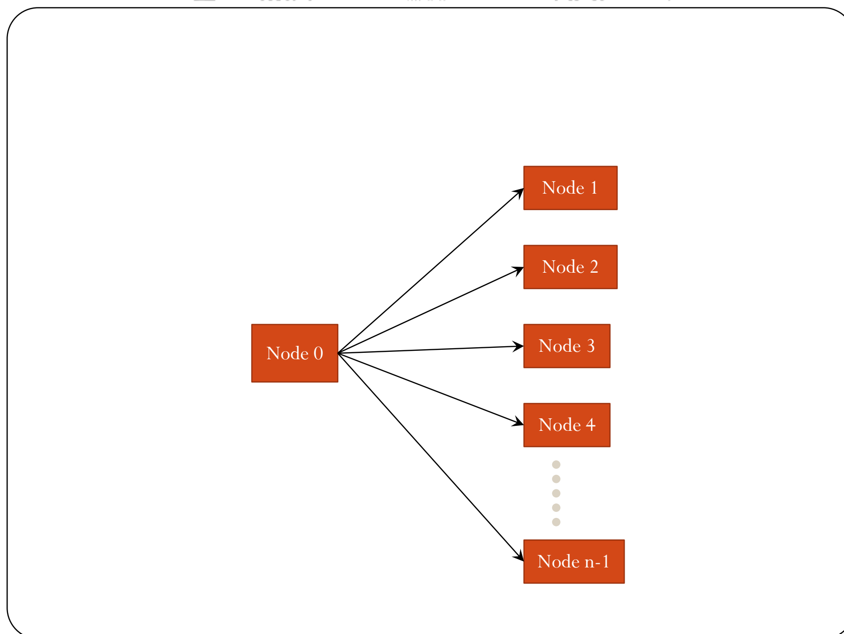
  #include "mpi.h"
  ```
  int MPI_Gather
       (
       void *sendbuf,
       int sendcnt,
       MPI_Datatype sendtype,
       void *recvbuf,
       int recvcount,
  ```

```
MPI_Datatype recvtype,
int root,
MPI_Comm comm
)
```

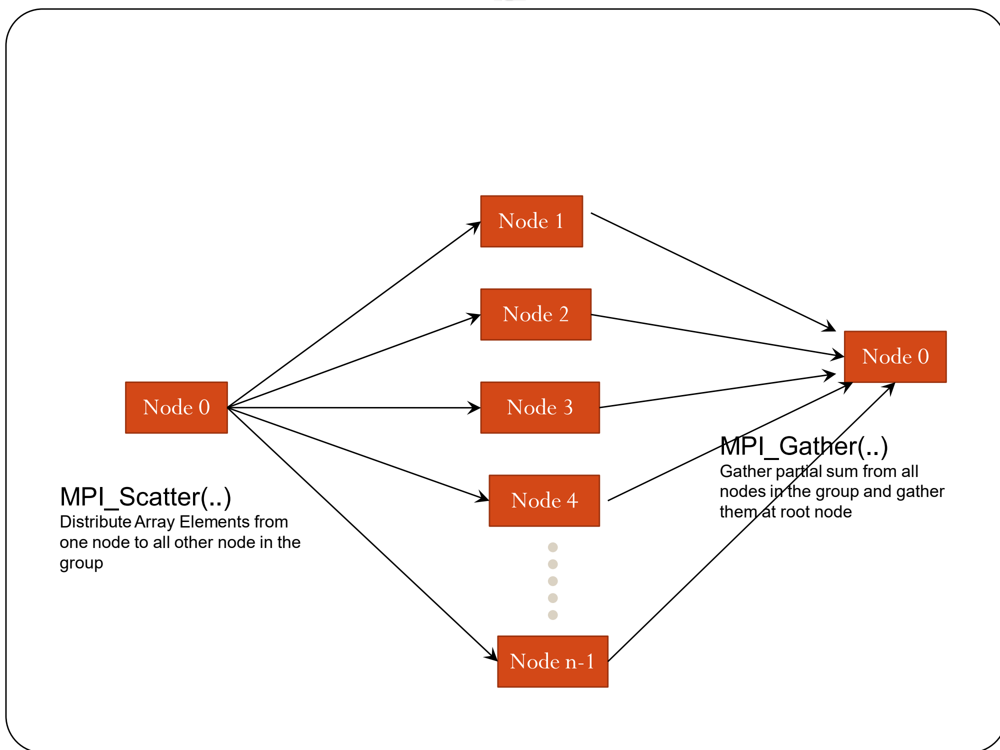
- Input Parameters:
    - **sendbuf:** starting address of send buffer
    - **sendcount:** number of elements in send buffer
    - **sendtype:** data type of send buffer elements
    - **recvcount:** number of elements for any single receive (significant only at root)
    - **recvtype:** data type of recv buffer elements (significant only at root)
    - **root:** rank of receiving process
    - **comm:** communicator
- Output Parameter:
    - recvbuf: address of receive buffer (significant only at root)

EXERCISE:

1. Write a program that broadcasts a number from one process to all others by using MPI_Bcast.



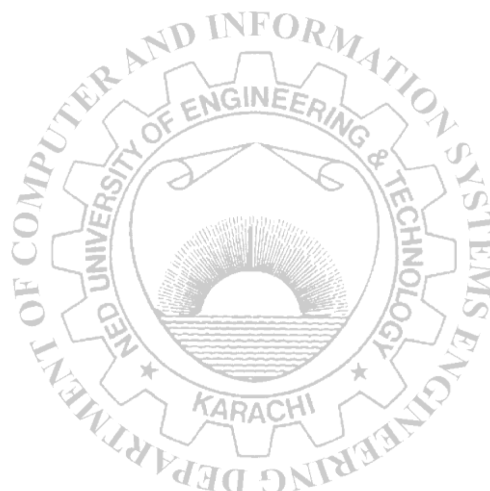
2. Break up a long vector into sub-vectors of equal length. Distribute sub-vectors to processes. Let the processes to compute the partial sums. Collect the partial sums from the processes and add them at root node using collective computation operations



3. Write a parallel program that calculates the value of PI using integral method.

Algorithm: The algorithm suggested here is chosen for its simplicity. The method evaluates the integral of 4/(1+x*x) between -1/2 and 1/2. The method is simple: the integral is approximated by a sum of n intervals; the approximation to the integral in each interval is (1/n)*4/(1+x*x). The master process (rank 0) asks the user for the number of intervals; the master should then broadcast this number to all of the other processes. Each process then adds up every n'th interval (x = -1/2+rank/n, -1/2+rank/n+size/n,). Finally, the sums computed by each process are added together using a reduction.

Program:


DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 6

OBJECT

Study of MPI collective operations using 'Collective Computation'

THEORY

Collective operations

MPI_Send and MPI_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operations are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission. MPI supports three classes of collective operations:

- Synchronization,
- Data Movement, and
- Collective Computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

Collective Computation Routines

Collective computation is similar to collective data movement with the additional feature that data may be modified as it is moved. The following routines can be used for collective computation.

- **MPI_Reduce:** Perform a reduction operation.
- **MPI_Allreduce:** Perform a reduction leaving the result in all participating processes
- **MPI_Reduce_scatter:** Perform a reduction and then scatter the result
- **MPI_Scan:** Perform a reduction leaving partial results (computed up to the point of a process's involvement in the reduction tree traversal) in each participating process. (parallel prefix)

Collective computation built-in operations

Many of the MPI collective computation routines take both built-in and user-defined combination functions. The built-in functions are:

Table 6.1 Collective Computation Operations

Operation handle	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical Exclusive OR
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise Exclusive OR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

MPI_Reduce:

MPI_Reduce apply some operation to some operand in every participating process. For example, add an integer residing in every process together and put the result in a process specified in the MPI_Reduce argument list. The subroutine MPI_Reduce combines data from all processes in a communicator using one of several reduction operations to produce a single result that appears in a specified target process.

When processes are ready to share information with other processes as part of a data reduction, all of the participating processes execute a call to MPI_Reduce, which uses local data to calculate each process's portion of the reduction operation and communicates the local result to other processes as necessary. Only the target_process_ID receives the final result.

```
int MPI_Reduce(
          void* operand /* in */,
          void* result /* out */,
          int count /* in */,
          MPI Datatype datatype /* in */,
          MPI Op operator /* in */,
          int root /* in */,
          MPI Comm comm /* in */
          )
```

Understanding the Argument List

- **operand** - starting address of the send bu.er.
- **result** - starting address of the receive bu.er.

- **count** - number of elements in the send bu.er.
- **datatype** - data type of the elements in the send/receive bu.er.
- **operator** - reduction operation to be executed.
- **root** - rank of the root process obtaining the result.
- **comm** - communicator.

Example of Usage

The given code receives data on only the root node (rank=0) and passes null in the receive data argument of all other nodes

```
int mynode, totalnodes;
int datasize; // number of data units over which
// reduction should occur
int root; // process to which reduction will occur

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

// Determine datasize and root
double * senddata = new double[datasize];
double * recvdata = NULL;

if(mynode == root)
recvdata = new double[datasize];
// Fill in senddata on all processes

MPI_Reduce(senddata,recvdata,datasize,MPI_DOUBLE,MPI_SUM,
root,MPI_COMM_WORLD);

// At this stage, the process root contains the result of
the reduction (in this   case MPI_SUM) in the recvdata
array
```

Key Points

- The recvdata array only needs to be allocated on the process of rank root (since root is the only processor receiving data). All other processes may pass NULL in the place of the recvdata argument.
- Both the senddata array and the recvdata array must be of the same data type. Both arrays should contain at least datasize elements.

MPI_Allreduce*:* Perform a reduction leaving the result in all participating processes

```
int MPI Allreduce(
          void* operand /* in */,
          void* result /* out */,
          int count /* in */,
```

```
MPI Datatype datatype /* in */,
MPI Op operator /* in */,
MPI Comm comm /* in */
)
```

Understanding the Argument List

- **operand** - starting address of the send buffer.
- **result** - starting address of the receive buffer.
- **count** - number of elements in the send/receive buffer.
- **datatype** - data type of the elements in the send/receive buffer.
- **operator** - reduction operation to be executed.
- **comm** - communicator.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units over which
// reduction should occur

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

// Determine datasize and root
double * senddata = new double[datasize];
double * recvdata = new double[datasize];
// Fill in senddata on all processes

MPI_Allreduce(senddata,recvdata,datasize,MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
// At this stage, all processes contains the result  of
the reduction (in this case MPI_SUM) in the recvdata array
```

Remarks

- In this case, the *recvdata* array needs to be allocated on all processes since all processes will be receiving the result of the reduction.

- Both the *senddata* array and the *recvdata* array must be of the same data type. Both arrays should contain at least *datasize* elements.

MPI_Scan:

- MPI_Scan: Computes the scan (partial reductions) of data on a collection of processes
- Synopsis:

  #include "mpi.h"

int MPI_Scan (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Input Parameters
  - sendbuf: starting address of send buffer
  - count: number of elements in input buffer
  - datatype: data type of elements of input buffer
  - op: operation
  - comm: communicator

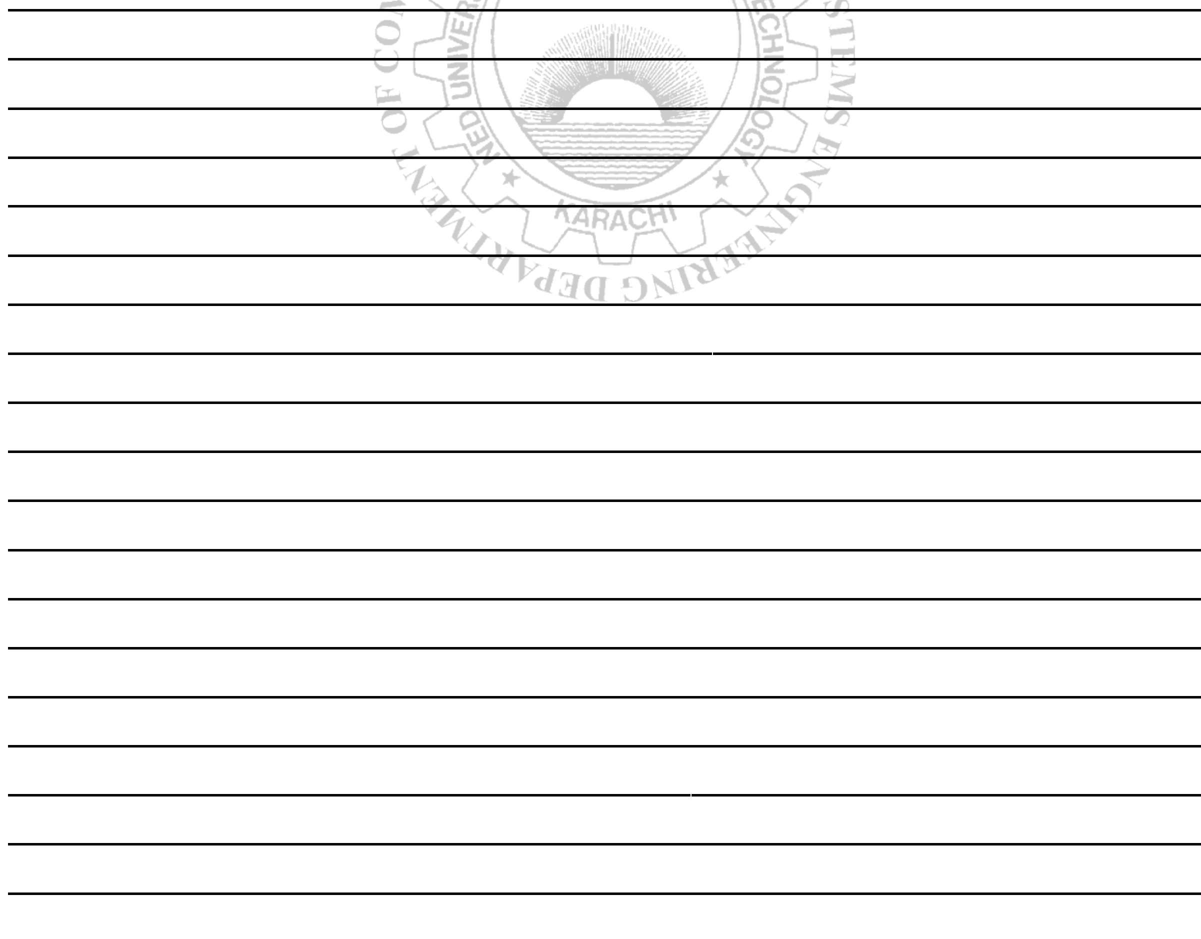
- Output Parameter:

  recvbuf: starting address of receive buffer

EXERCISE:

1. Break up a long vector into sub-vectors of equal length. Distribute sub-vectors to processes. Let the processes to compute the partial sums. Collect the partial sums from the processes and add them at root node using collective computation operations

Program:



2. Write a program to calculate prefix sum of ‘n’ numbers on ‘n’ processes. Use MPI_Scan to address this problem.

The above problem can be best stated with the help of following figure.

Node 0 Node 1 Node 2 Node n

S_0 S_1 S_2 S_n

Program:

3. Write and explain argument lists of the following and say how they are different from the two functions you have seen:
- **MPI_Allreduce**
- **MPI_Reduce_scatter**

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 7

OBJECT

To understand MPI Non-Blocking operation

THEORY

MPI: Non-Blocking Communications

Non-blocking point-to-point operation allows overlapping of communication and computation to use the common parallelism in modern computer systems more efficiently. This enables the user to use the CPU even during ongoing message transmissions at the network level.

As *MPI_Send and MPI_Recv* or *MPI Sendrecv fuctions* require some level of synchronization for associating the corresponding sends and receives on the appropriate processes. *MPI_Send* and *MPI_Recv* are *blocking communications*, which means that they will not return until it is safe to modify or use the contents of the send/recv buffer respectively.

MPI also provides *non-blocking versions* of these functions called *MPI_Isend* and *MPI_Irecv*, where the "I" stands for immediate. These functions allow a process to post that it wants to send to or receive from a process, and then later is allowed to call a function (*MPI Wait*) to complete the sending/receiving. These functions are useful in that they allow the programmer to appropriately stagger computation and communication to minimize the total waiting time due to communication.

The understand the basic idea behind MPI_Isend and MPI_Irecv, Suppose process 0 needs to send information to process 1, but due to the particular algorithms that these two processes are running, the programmer knows that there will be a mismatch in the synchronization of these processes. Process 0 initiates an MPI_Isend to process 1 (posting that it wants to send a message), and then continues to accomplish things which do not require the contents of the buffer to be sent. At the point in the algorithm where process 0 can no longer continue without being guaranteed that the contents of the sending buffer can be modified, process 0 calls MPI Wait to wait until the transaction is completed. On process 1, a similar situation occurs, with process 1 posting via MPI_Irecv that it is willing to accept a message. When process 1 can no longer continue without having the contents of the receive buffer, it too calls MPI_Wait to wait until the transaction is complete. At the conclusion of the MPI_Wait, the sender may modify the send buffer without compromising the send, and the receiver may use the data contained within the receive buffer.

Why Non Blocking Communication?

The communication can consume a huge part of the running time of a parallel application. The communication time in those applications can be addressed as overhead because it does not progress the solution of the problem in most cases (with exception of reduce operations). Using overlapping techniques enables the user to move communication and the necessary synchronization in the background and use parts of the original communication time to perform useful computation.

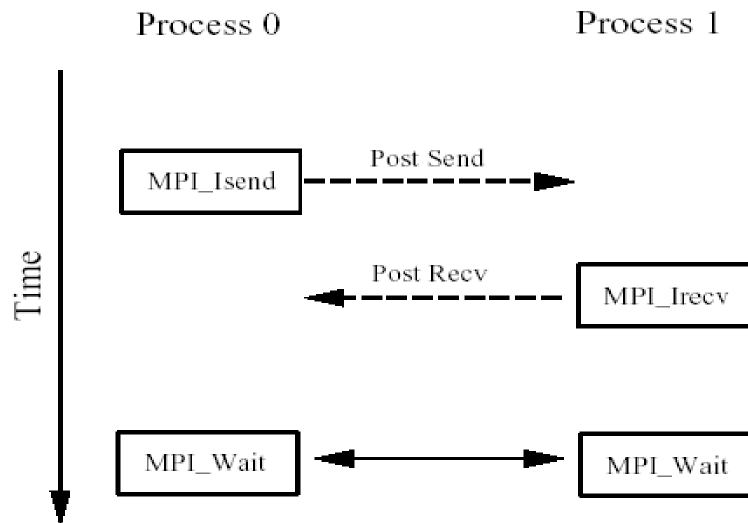


Figure7.1 MPI Isend/MPI Irecv schematic demonstrating the communication between two processes.

Function Call Syntax

```
int MPI Isend(
          void* message /* in */,
          int count /* in */,
          MPI Datatype datatype /* in */,
          int dest /* in */,
          int tag /* in */,
          MPI Comm comm /* in */,
          MPI Request* request /* out */
          )


int MPI Irecv(
          void* message /* out */,
          int count /* in */,
          MPI Datatype datatype /* in */,
          int source /* in */,
          int tag /* in */,
          MPI Comm comm /* in */,
          MPI Request* request /* out */
          )


int MPI Wait(
          MPI Request* request /* in/out */
          MPI Status* status /* out */
          )
```

Understanding the Argument Lists

- **message** - starting address of the send/recv bu.er.
- **count** - number of elements in the send/recv bu.er.
- **datatype** - data type of the elements in the send bu.er.
- **source** - process rank to send the data.
- **dest** - process rank to receive the data.
- **tag** - message tag.
- **comm** - communicator.
- **request** - communication request.
- **status** - status object.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units to be sent/recv
int sender; // process number of the sending process
int receiver; // process number of the receiving process
int tag; // integer message tag
MPI_Status status; // variable to contain status
information
MPI_Request request; // variable to maintain
// isend/irecv information

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

// Determine datasize
double * databuffer = new double[datasize];
// Fill in sender, receiver, tag on sender/receiver
processes,
// and fill in databuffer on the sender process.

if(mynode==sender)
MPI_Isend(databuffer,datasize,MPI_DOUBLE,receiver,tag,
MPI_COMM_WORLD,&request);

if(mynode==receiver)
MPI_Irecv(databuffer,datasize,MPI_DOUBLE,sender,tag,
MPI_COMM_WORLD,&request);
// The sender/receiver can be accomplishing various things
// which do not involve the databuffer array

MPI_Wait(&request,&status); //synchronize to verify
// that data is sent
// Send/Recv complete
```

Key Points

- In general, the *message* array for both the sender and receiver should be of the same type and both of size at least *datasize*.
- In most cases the *sendtype* and *recvtype* are identical.
- After the *MPI_Isend* call and before the *MPI_Wait* call, the contents of *message* should not be changed.
- After the *MPI_Irecv* call and before the *MPI_Wait* call, the contents of *message* should not be used.
- An *MPI_Send* can be received by an *MPI_Irecv/MPI_Wait*.
- An *MPI_Recv* can obtain information from an *MPI_Isend/MPI_Wait*.
- The tag can be any integer between 0-32767.
- *MPI_Irecv* may use for the tag the wildcard *MPI_ANY_TAG*. This allows an *MPI_Irecv* to receive from a send using any tag.
- *MPI_Isend* cannot use the wildcard *MPI_ANY_TAG*. A specific tag must be specified.
- *MPI_Irecv* may use for the source the wildcard *MPI_ANY_SOURCE*. This allows an *MPI_Irecv* to receive from a send from any source.
- *MPI_Isend* must specify the process rank of the destination. No wildcard exists.

EXERCISE:

1. Write a parallel program having the non blocking processes communications which calculates the sum of numbers in parallel on different numbers of nodes. Also calculate the execution time.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Output: (On Single Node)

Execution Time:

Output: (On Two Nodes)

Execution Time:

Speedup:

Output: (On Four Nodes)

Execution Time:

Speedup:

Output: (On Sixteen Nodes)

Execution Time:

Speedup:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

2. Write two programs that utilizes the functions MPI_Waitall and MPI_Waitany respectively.

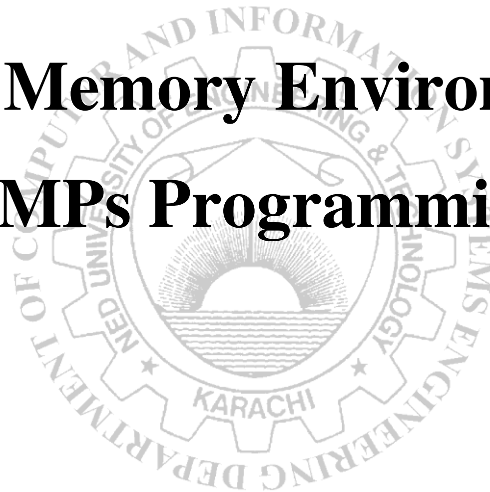
Program 1:

Program 2:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Part Two

Shared Memory Environments / SMPs Programming



Introduction

As Parallel commuters matured, demand for higher abstraction for programming such machines grew. Most of the early parallel computers consisted of tens of CPUs connected together and to a globally addressable shared memory, via a bus. That is a task running on any CPU could access any memory location in the computer with equal speed. Such systems are referred to as Uniform Memory Access or UMA architectures. For these machines their respective vendors started providing shares memory language. Such languages typically consist of a means for spawning multiple tasks for a problem, Synchronization constructs for task to exchange data via shares memory and machine to allow synchronization which each other via barriers and related functions. Such environments are characterized as pure shared memory environments.

As the number of CPUs started increasing on parallel computers, bus based architectures exhibited limited performance improvement, since the bus bandwidth requirement reached its saturation point. Hence larger parallel computers adopted switch interconnects for connecting CPUs to memory modules. For such systems the distance of a memory module from a CPU is not constant, resulting in a non uniform memory access speed by a task. Such systems are referred to as Non Uniform Memory Access or NUMA architectures.

Typically, on NUMA architectures each CPU has local memory that is only addressable by task on that CPU, and the collection of all these local memory modules form the entire memory of the system. Programming to this model involves explicitly passing information or message from a task on one CPU to another, since no shared memory exists that can be utilized for this purpose. This message passing programming model is viewed by many as difficult, when compared to the shared memory model. To overcome this difficulty in using distributed memory machines vendors of such systems started providing simulated globally addressable shared memory environments to the user, taking care of the translation to the physically distributed local memories in the operating systems. Such systems are referred to as Distributed Shared Memory or DSM. Since the memory is not really shared in hardware, but is available as such as to the programmer, such a shared memory paradigm will be characterized as a virtual shared memory paradigm.

1. Pure Shared Memory Environment

These languages are designed for systems where the entire memory in the system is uniformly globally addressable and those environments do not make any provision for, nor do they have any tuning hooks for, application to take care of non uniform memory accesses. While these languages will continue to work on virtual shared memory systems, they have no inherent features to support any NUMA characteristics that virtual shared memory machines exhibit. Most of the early parallel machine had their own shared memory parallel environments, and all those had a similar flavor. From these languages evolved the concept of threads. Threads are lightweight entities, which are similar to processes except that they require minimal resource to run. A process may consist of several threads, each of which represents a separate execution

context; hence, a separate program counter, stacks, and registers. All threads of a process share the remaining resources with the other threads in the process. Two types of thread environment are gaining widespread popularity across various platforms: Pthreads and Java threads.

1.1 Pthreads

Pthreads is an abbreviation for POSIX threads and refers to a standard specification for threads developed by the POSIX committee. The Pthreads environment provides two types of synchronization primitives Mutex and Condition Variable. Mutexes are simple lock primitives that can be used to control access to a shared resource and the operations supported on a mutex to achieve this are lock and unlock primitives. Only one thread may own a mutex at a time and is thus guaranteed exclusive access to the associated resource.

Synchronization using mutexes may not be sufficient for many programs since they have limited functionality. Condition variables supplement the functionality of mutexes by allowing threads to block and wait for an event and be woken up when the event occurs. Pthreads are limited to use within a single address space, and can not be spread across distinct address spaces.

1.2 Java Threads

The Java language also provides a threads programming model via its Thread and Thread Group classes. The Thread class implements a generic thread that, by default, does nothing. Users specify the body of the thread by providing a run method for their Thread objects. The Thread Group class provides a mechanism for manipulating a collection of threads at the same time, such as starting or stopping a set of threads via a single invocation of a method. Synchronization between various threads in a program is provided via two constructs synchronized blocks and walt-notify constructs. In the Java language a block or a method of the program that is identified with the synchronized keyword represents a critical section in the program. The Java platform associates a lock with every object that has synchronized code. The wait construct allows a thread to relinquish its lock and wait for notification of a given event, and the notify construct allows thread to signal the occurrence of an event to another thread that is waiting for this event.

2. Virtual Shared Memory Environment

The parallel environments that fall under this category provide a shared memory interface to the programmer. That is, to the programmer the entire memory in the system is globally addressable. The underlying hardware may consist of physically separate memory modules. In some cases, the operating system may take care of the distributed nature of the system memory providing a shared memory view to these environments. In other cases, the memory is still presented to those environments as logically distinct modules, and these languages need to hide that detail from the programmer. These languages do this by providing a shared memory environment, to the programmer and by performing the translation to the actual distributed memory at their layer. In either case, memory access becomes non uniform, and virtual shared memory environments typically provide hooks to the programmer to manipulate the placement of data, placement of tasks on processors, movement of data relative to the tasks, and/or migration of tasks from one

CPU to another. These hooks can be used by a parallel program in an attempt to maximize the performance of the system controlling the placement of data, both statically and dynamically, so it is near the tasks that access that data. The biggest advantage of such an environment is that it still preserves a shared memory programming interface which is considered by many to be a simpler programming model than a message programming interface. Popular examples .of such environments are described below.

2.1 Linda

Linda allows tasks to communicate with each other by inserting and retrieving data items called Tuples, into a debuted shared memory called Tuple Space. A tuple consists of a string which is the tap1e's identifier, and zero or more data items. A tuple Space is a segment of memory in one or mare computers whose purpose is to serve as a temporary storage area for data being transferred between tasks. It is an associative memory abstraction where tasks communicate by inserting and removing tuples from this tuple space. When a task is ready to send information to another task, it places the corresponding tuple in the tuple space. When the receiver task is ready to receive this information, it retrieves this tuple from the tuple space. This decouples the send and receives parts of the communication so that the sender task does not have to block until the receiver is ready to receive the data being communicated.

Linda works very well smaller parallel programs with a few component tasks. However as the number of tasks increases in a program controlling access to a single tuple space and managing the tasks becomes difficult. This is mainly because there is no scoping in the tuple space, making the probability of access conflicts higher, which places a responsibility on programmers to be extra careful when the number of tasks becomes large.

2.2 SHMEM

The SHMEM environment provides the view of a logically shared, distributed memory access to the programmer and is available on massively parallel distributed memory machines as well on distributed shared memory machines. It enables tasks to communication among themselves via low-latency, high-bandwidth primitives. In addition to being used on shared memory architecture, SHMEM can be used by tasks in distinct address spaces to explicitly pass data among each other. It provides an efficient alternative to using message passing for inter-task communication. SHMEM supports remote data transfer throughput operations, which can transfer data to another task, and through get operations, which can retrieve data from another task. This one-sided style of communication offered by SHMEM makes programming simpler since a matching receive request need not match every send request. SHMEM also supports broadcast and reduction operations, barrier synchronization and atomic memory operations.

2.3 High-Performance FORTRAN

High Performance Fortran or HPF is a standard defined for FORTRAN parallel programs with the goals of achieving program portability across a number of parallel machines, and achieving high performance on parallel computers with no uniform memory access costs. HPF supports the data parallel programming model, where data are divided across machines, and the same program

is executed on different machines on different subsets of the overall data. The HPF environment provides software tools such as HPF compilers that produce programs for parallel computers with no uniform access cost. There have been two versions of this standard to date. The first version, HPF1, defined in 1993, was an extension to FORTRAN 90. The second version, HPF-2, defined in 1997, is an extension to the current FORTRAN standard (FORTRAN 95).

An HPF programmer expresses parallelism explicitly in his program, and the data distribution is tuned to control the load balance and to minimize inter-task communication. On the other hand, given a data distribution, an HPF compiler may be able to identify operations that can be executed concurrently, and thus generate even more efficient code. HPF's constructs allow programmers to indicate potential parallelism at a relatively high level, without entering into the low-level details of message-passing and synchronization. When an HPF program is compiled, the compiler assumes responsibility for scheduling the parallel operations on the physical machines, thereby reducing the time and effort required for parallel program development.

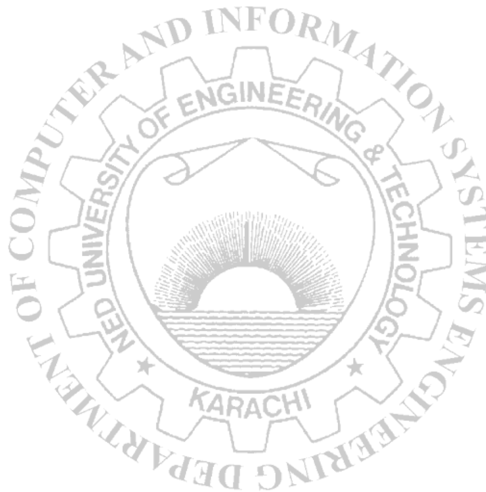
3.4 Remote Threads

Pthreads can not be extended across distinct address space boundaries such as a cluster of workstations. Remote threads or Rthreads extend Pthreads-like constructs between address spaces. They provide software distributed shared memory system that supports sharing of global variables on clusters of computers with physically distributed memory. Rthreads use explicit function calls to access distributed shared data. Its synchronization primitives are syntactically and semantically similar to those of Pthreads. The Rthreads environment consists of a pre-compiler that automatically transforms Pthreads programs into Rthreads programs. The programs can still change the Rthreads code for further optimization in the transformed program. Also, Pthreads and Rthreads can be mixed within a single program. Heterogeneous clusters are supported in Rthreads by implementing it on top of portable message passing environments such as PVM, MPI, and DCE.

3.5 OpenMP

OpenMP is a new standard that has been defined by several vendors as the standard Application Programming Interface or API for the shared memory multiprocessing model. It attempts to standardize existing practices from several different vendor-specific shared memory environments. OpenMP provides a portable shared memory API across different platforms including DEC, HP, IBM, Intel, Silicon Graphics/Cray, and Sun. The languages supported by OpenMP are FORTRAN, C and C++. Its main emphasis is on performance and scalability. OpenMP consists of a collection of directives, library routines, and environment variables used to specify shared memory parallelism in a program's source code. It standardizes fine grained (loop level) parallelism and also supports coarse-grain parallelism. Fine grain parallelism is achieved via the fork/join model. A typical OpenMP program starts executing as a single task, and on encountering a parallel construct, a group of tasks is spawned to execute the parallel region, each with its own data environment. The complier is responsible for assigning the appropriate iteration to the tasks in the group. The parallel region ends with the end do construct, which represents an implied barrier. At this point, the results of the parallel region are used to update the data environment of the original task, which then resumes execution. This sequence of fork/join

actions is repeated for every parallel construct in the program, enabling loop-level parallelism in a program. For enabling coarse-grain parallelism effectively, OpenMP introduces the concept of orphan directives, which are directives encountered outside the lexical extent of the parallel region. This allows a parallel program to specify control from anywhere inside the parallel region, as opposed to only from the lexically contained portion, which is often necessary in coarse-grained parallel programs.



Lab Session 8

OBJECT

Basics of OpenMP API (Open Multi-Processor API)

THEORY

OpenMP

OpenMP is a portable and standard Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

OpenMP attempts to standardize existing practices from several different vendor-specific shared memory environments. OpenMP provides a portable shared memory API across different platforms including DEC, HP, IBM, Intel, Silicon Graphics/Cray, and Sun. The languages supported by OpenMP are FORTRAN, C and C++. Its main emphasis is on performance and scalability.

Goals of OpenMP

- **Standardization:** Provide a standard among a variety of shared memory architectures/platforms
- **Lean and Mean:** Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:** Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach and also provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:** Supports Fortran (77, 90, and 95), C, and C++

OpenMP Programming Model

Shared Memory, Thread Based Parallelism:

- OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Fork - Join Model:

- OpenMP uses the fork-join model of parallel execution:

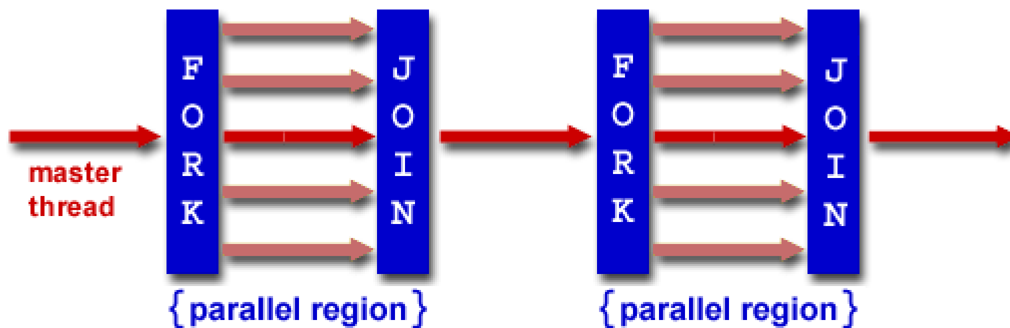


Figure 8.1 Fork and Join Model

- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a ***team*** of parallel threads
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Compiler Directive Based:

- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

Nested Parallelism Support:

- The API provides for the placement of parallel constructs inside of other parallel constructs.
- Implementations may or may not support this feature.

Dynamic Threads:

- The API provides for dynamically altering the number of threads which may used to execute different parallel regions.
- Implementations may or may not support this feature.

I/O:

- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.

Components of OpenMP API

- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

C / C++ - General Code Structure

```
#include <omp.h>
main ()
{

     int var1, var2, var3;

     Serial code


          #pragma omp parallel private(var1, var2)
          shared(var3)

          {
                 Parallel section executed by all threads
                       .
                       .
                       .
                 All threads join master thread and disband
            }

      Resume serial code
  }
```

Important terms for an OpenMP environment

Construct: A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct.

directive: A C or C++ **#pragma** followed by the **omp** identifier, other text, and a new line. The directive specifies program behavior.

Region: A dynamic extent.

dynamic extent: All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

lexical extent: Statements lexically contained within a *structured block*.

structured block: A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement. A compound statement is a structured block if its execution always begins at the opening **{** and always ends at the closing **}**. An expression statement, selection statement, iteration statement is a structured block if the corresponding compound statement obtained by enclosing it in **{** and **}**would be a structured block. A jump statement, labeled statement, or declaration statement is not a structured block.

Thread: An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

master thread: The thread that creates a team when a *parallel region* is entered.

serial region: Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

parallel region: Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.

Variable: An identifier, optionally qualified by namespace names, that names an object.

Private: A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a threadprivate directive, a private, firstprivate, lastprivate, or reduction clause, or use of the variable as a forloop control variable in a for loop immediately following a for or parallel for directive.

Shared: A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.

Team: One or more threads cooperating in the execution of a construct.

Serialize: To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **omp_in_parallel()** (apart from the effects of any nested parallel constructs).

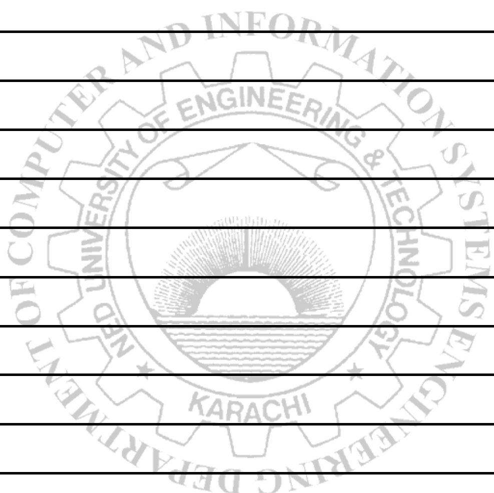
Barrier: A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

EXERCISE:

1. Discuss the different types of Parallel Programming Models.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

2. List down the possible characteristics of an APIs (Application programming interface).



Lab Session 9

OBJECT

To get familiarized with OpenMP Directives

THEORY

OpenMP Directives Format

OpenMP directives for C/C++ are specified with the pragma preprocessing directive.

#pragma omp directive-name [clause[[,] clause]. . .] new-line

Where:

- **#pragma omp:** Required for all OpenMP C/C++ directives.
- **directive-name:** A valid OpenMP directive. Must appear after the pragma and before any clauses.
- **[clause, ...]:** Optional, Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- **Newline:** Required, Precedes the structured block which is enclosed by this directive.

General Rules:

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

OpenMP Directives or Constructs

- Parallel Construct
- Work-Sharing Constructs
  - Loop Construct
  - Sections Construct

- Single Construct

• Data-Sharing, No Wait, and Schedule Clauses

• Barrier Construct

• Critical Construct

• Atomic Construct

• Locks

• Master Construct

Directive Scoping

Static (Lexical) Extent:

- The code textually enclosed between the beginning and the end of a structured block following a directive.
- The static extent of a directives does not span multiple routines or code files

Orphaned Directive:

- An OpenMP directive that appears independently from another enclosing directive is said to be an orphaned directive. It exists outside of another directive's static (lexical) extent.
- Will span routines and possibly code files

Dynamic Extent:

- The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Parallel Construct

This construct is used to specify the computations that should be executed in parallel. Parts of the program that are not enclosed by a parallel construct will be executed serially. When a thread encounters this construct, a team of threads is created to execute the associated parallel region, which is the code dynamically contained within the parallel construct. But although this construct ensures that computations are performed in parallel, it does not distribute the work of the region among the threads in a team. In fact, if the programmer does not use the appropriate syntax to specify this action, the work will be replicated. At the end of a parallel region, there is an implied *barrier* that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel region.

The thread that encounters the parallel construct becomes the *master* of the new team. Each thread in the team is assigned a unique thread number (also referred to as the "thread id") to identify it. They range from zero (for the master thread) up to one less than the number of threads within the team, and they can be accessed by the programmer. Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution.

Format

```
#pragma omp parallel [clause ...] ……. newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)
{
 structured_block
            }
```

Example of a parallel region

```
#include <omp.h>
main()
        {
                #pragma omp parallel
           {
               printf("The parallel region is executed by
               thread %d\n",
               omp_get_thread_num());

          } /*-- End of parallel region --*/
                 }/*-- End of Main Program --*/
```

Here, the OpenMP library function **omp_get_thread_num()** is used to obtain the number of each thread executing the parallel region. Each thread will execute all code in the parallel region, so that we should expect each to perform the print statement.. Note that one cannot make any assumptions about the order in which the threads will execute the printf statement. When the code is run again, the order of execution could be different.

Possible output of the code with four threads.

The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
The parallel region is executed by thread 1

Clauses supported by the parallel construct

- **if***(scalar-expression)*
- **num threads***(integer-expression)*
- **private***(list)*
- **firstprivate***(list)*
- **shared***(list)*
- **default(none|shared)**
- **copyin***(list)*
- **reduction***(operator:list)*

Details and usage of clauses are disused in lab session B.4

Key Points:

- A program that branches into or out of a parallel region is nonconforming. In other words, if a program does so, then it is *illegal*, and the behavior is undefined.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive or on any side effects of the evaluations of the clauses.
- At most one if clause can appear on the directive.
- At most one num_threads clause can appear on the directive. The expression for the clause must evaluate to a positive integer value.

Determining the Number of Threads for a parallel Region

When execution encounters a parallel directive, the value of the if clause or num_threads clause (if any) on the directive, the current parallel context, and the values of the nthreads-var, dyn-var, thread-limit-var, max-active-level-var, and nest-var ICVs are used to determine the number of threads to use in the region.

Note that using a variable in an if or num_threads clause expression of a parallel construct causes an implicit reference to the variable in all enclosing constructs. The if clause expression and the num_threads clause expression are evaluated in the context outside of the parallel construct, and no ordering of those evaluations is specified. It is also unspecified whether, in what order, or how many times any side-effects of the evaluation of the num_threads or if clause expressions occur.

Example: use of num_threads Clause

The following example demonstrates the num_threads clause. The parallel region is executed with a maximum of 10 threads.

```
#include <omp.h>
main()
      {
                ...
```

```
                #pragma omp parallel num_threads(10)
                {
                    ... parallel region ...
                }
        }
```

Specifying a Fixed Number of Threads

Some programs rely on a fixed, pre-specified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation-defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using omp_set_dynamic and omp_set_num_threads

Example:

```
#include <omp.h>
main()
     {
        omp_set_dynamic(0);
          omp_set_num_threads(16);

              #pragma omp parallel num_threads(10)
               {
                    ... parallel region ...
               }
        }
```

EXERCISE:

1. Code the above example programs and write down their outputs.

Out put of Program 01:

____

____

____

____

____

____

____

____

____

____

____

Output of Program 02:

Output of Program 03:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 10

OBJECT

Sharing of work among threads using Loop Construct in OpenMP

THEORY

Introduction

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed below.

- **#pragma omp for:** Distribute iterations over the threads
- **#pragma omp sections:** Distribute independent work units
- **#pragma omp single:** Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the nowait clause.

The Loop Construct

The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel. At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.

Format:

```
#pragma omp for [clause ...]  newline

                         schedule (type [,chunk])
                         ordered
                         private (list)
                         firstprivate (list)
                         lastprivate (list)
                         shared (list)
                         reduction (operator: list)
                         nowait

   for_loop
```

Example of a work-sharing loop
Each thread executes a subset of the total iteration space $i = 0, \ldots, n - 1$

```
#include <omp.h>
main()

{
     #pragma omp parallel shared(n) private(i)
     {
          #pragma omp for
          for (i=0; i<n; i++)
          printf("Thread %d executes loop iteration %d\n",
          omp_get_thread_num(),i);
     }
  }
```

Here we use a parallel directive to define a parallel region and then share its work among threads via the for work-sharing directive: the **#pragma omp for** directive states that iterations of the loop following it will be distributed. Within the loop, we use the OpenMP function omp_get_thread_num(), this time to obtain and print the number of the executing thread in each iteration. Parallel construct that state which data in the region is shared and which is private. Although not strictly needed since this is enforced by the compiler, loop variable i is explicitly declared to be a private variable, which means that each thread will have its own copy of i. its value is also undefined after the loop has finished. Variable n is made shared.
Output from the example which is executed for $n = 9$ and uses four threads.

Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6

Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4

Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the parallel construct immediately followed by the work-sharing construct.

<table>
<tr><th>Full version Combined construct</th><th>Combined construct</th></tr>
<tr><td><pre>#pragma omp parallel
  {
     #pragma omp for
       for-loop
   }</pre></td><td><pre>#pragma omp parallel for
     {
       for-loop
     }</pre></td></tr>
</table>

EXERCISE:

1. Code the above example programs and write down their outputs.

Output of Program:

____
____
____
____
____
____
____
____
____
____
____
____
____
____
____

2. write a parallel program, after discussing your instructor, which uses Loop Construct.

Program:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 11

OBJECT

Clauses in Loop Construct

THEORY

Introduction

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped.

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.

These constructs provide the ability to control the data environment during execution of parallel constructs.

- They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
- They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.

List of Clauses

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION
- COPYIN

PRIVATE Clause

The PRIVATE clause declares variables in its list to be private to each thread.

Format: PRIVATE (list)

Notes:

- PRIVATE variables behave as follows:

- A new object of the same type is declared once for each thread in the team
- All references to the original object are replaced with references to the new object
- Variables declared PRIVATE are uninitialized for each thread

Example of the private clause – Each thread has a local copy of variables i and a.

```
#pragma omp parallel for private(i,a)

      for (i=0; i<n; i++)
      {
            a = i+1;
            printf("Thread %d has a value of a = %d for i =
            %d\n",
            omp_get_thread_num(),a,i);
      } /*-- End of parallel for --*/
```

SHARED Clause

The SHARED clause declares variables in its list to be shared among all threads in the team.

Format: SHARED *(list)*

Notes:

- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

Example of the shared clause – All threads can read from and write to vector a.

```
#pragma omp parallel for shared(a)

      for (i=0; i<n; i++)
      {
            a[i] += i;
      } /*-- End of parallel for --*/
```

DEFAULT Clause

The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope for all variables in the lexical extent of any parallel region.
The default clause is used to give variables a default data-sharing attribute. Its usage is straightforward. For example, default (shared) assigns the shared attribute to all variables

referenced in the construct. This clause is most often used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed.
If default(none) is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct. Although variables with a predetermined data-sharing attribute need not be listed in one of the clauses, it is strongly recommend that the attribute be explicitly specified for *all* variables in the construct.

Format: DEFAULT (SHARED | NONE)

Notes:

- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
- The C/C++ OpenMP specification does not include "private" as a possible default. However, actual implementations may provide this option.
- Only one DEFAULT clause can be specified on a PARALLEL directive

Example of the Deafulat clause: all variables to be shared, with the exception of a, b, and c.

```
#pragma omp for default(shared) private(a,b,c),
```

FIRSTPRIVATE Clause

The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

Format: FIRSTPRIVATE *(LIST)*

Notes:

- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct

Example using the firstprivate clause – Each thread has a pre-initialized copy of variable indx. This variable is still private, so threads can update it individually.

```
for(i=0; i<vlen; i++) a[i] = -i-1;
indx = 4;
{
#pragma omp parallel default(none) firstprivate(indx)
private(i,TID) shared(n,a)

    {
          TID = omp_get_thread_num();
          indx += n*TID;
```

```
            for(i=indx; i<indx+n; i++)
            a[i] = TID + 1;
        }
    }
    printf("After the parallel region:\n");
    for (i=0; i<vlen; i++)
    printf("a[%d] = %d\n",i,a[i]);
```

LASTPRIVATE Clause

The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

Format: LASTPRIVATE *(LIST)*

Notes:

- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution

Example of the lastprivate clause – This clause makes the sequentially last value of variable a accessible outside the parallel loop.

```
    #pragma omp parallel for private(i) lastprivate(a)
        for (i=0; i<n; i++)
        {
            a = i+1;
            printf("Thread %d has a value of a = %d for i =
            %d\n",
            omp_get_thread_num(),a,i);
        } /*-- End of parallel for --*/

            printf("Value of a after parallel for: a =
%d\n",a);
```

COPYIN Clause

The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.

Format: COPYIN *(LIST)*

Notes:

- List contains the names of variables to copy. The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

REDUCTION Clause

The REDUCTION clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Format: REDUCTION *(OPERATOR: LIST)*

Notes:

- Variables in the list must be named scalar variables. They can not be array or structure type variables. They must also be declared SHARED in the enclosing context.
- Reduction operations may not be associative for real numbers.
- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:

C / C++

`x = x op expr`
`x = expr op x (except subtraction)`
`x binop = expr`
`x++`
`++x`
`x--`
`--x`

x is a scalar variable in the list ***expr*** is a scalar expression that does not reference ***x*** ***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, || ***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

Example of REDUCTION - Vector Dot Product. Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (SCHEDULE STATIC). At the end of the parallel loop construct, all threads will add their values of "result" to update the master thread's global copy.

```
#include <omp.h>
main ()
{
     int   i, n, chunk;
     float a[100], b[100], result;
     n = 100;
     chunk = 10;
     result = 0.0;
```

```
      for (i=0; i < n; i++)
        {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
        }
      #pragma omp parallel for  default(shared) private(i)
      schedule(static,chunk) reduction(+:result)
        {
           for (i=0; i < n; i++)
           result = result + (a[i] * b[i]);

      printf("Final result= %f\n",result);
      }
}
```

SCHEDULE:

Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

Nowait Clause

The nowait clause allows the programmer to fine-tune a program's performance. When we introduced the work-sharing constructs, we mentioned that there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP; in other words, if it is added to a construct, the barrier at the end of the associated construct will be suppressed. When threads reach the end of the construct, they will immediately proceed to perform other work. Note, however, that the barrier at the end of a parallel region cannot be suppressed.

Example of the nowait clause in C/C++ – The clause ensures that there is no barrier at the end of the loop.

```
#pragma omp for nowait
    for (i=0; i<n; i++)
    {
    ............
    }
```

Clauses / Directives Summary

Table 11.1 Comparative Analysis for a set of instruction

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

- The following OpenMP directives do not accept clauses:
  - MASTER
  - CRITICAL
  - BARRIER
  - ATOMIC
  - FLUSH
  - ORDERED
  - THREADPRIVATE

- Implementations may (and do) differ from the standard in which clauses are supported by each directive.

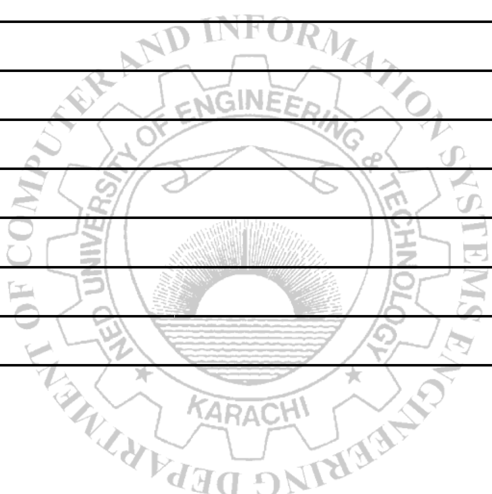
EXERCISE:

1. Code the above example programs and write down their outputs.

Output of example programs

2. Write a parallel program that sums a given arrays using reduction clause.

Program:



Lab Session 12

OBJECT

Sharing of work among threads in an OpenMP program using 'Sections Construct'

THEORY

Introduction

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed below.

- **#pragma omp for !$omp do** Distribute iterations over the threads
- **#pragma omp sections !$omp sections** Distribute independent work units
- **#pragma omp single !$omp single** Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the nowait clause .

The Sections Construct

The *sections construct* is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads. It consists of two directives: first, **#pragma omp sections:** to indicate the start of the construct and second, **the #pragma omp section:** to mark each distinct section. Each section must be a structured block of code that is independent of the other sections.

At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks. If there are fewer code blocks than threads, the remaining threads will be idle. Note that the assignment of code blocks to threads is implementation-dependent.

Format:

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
        {

          #pragma omp section   newline
              structured_block
          #pragma omp section   newline
             structured_block
          }
```

Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the parallel construct immediately followed by the work-sharing construct.

<table>
<tr><th>Full version Combined construct</th><th>Combined construct</th></tr>
<tr><td><pre>
#pragma omp parallel
{
     #pragma omp sections
    {

        [#pragma omp section]
        structured block

        [#pragma omp section
       structured block
    }
}
</pre></td><td><pre>
#pragma omp parallel sections

{

    [#pragma omp section]
    structured block

    [#pragma omp section]
    structured block
. . .
}
</pre></td></tr>
</table>

Example of parallel sections

If two or more threads are available, one thread invokes funcA() and another thread calls funcB(). Any other threads are idle.

```
#include <omp.h>

main()

{
     #pragma omp parallel
                    {
          #pragma omp sections
          {
               #pragma omp section
               (void) funcA();

               #pragma omp section
               (void) funcB();

           } /*-- End of sections block --*/

        } /*-- End of parallel region --*/

        } /*-- End of Main Program --*/

     void funcA()
     {
     printf("In funcA: this section is executed by thread
     %d\n",
     omp_get_thread_num());
     }

      void funcB()
     {
     printf("In funcB: this section is executed by thread
     %d\n",
     omp_get_thread_num());
     }
```

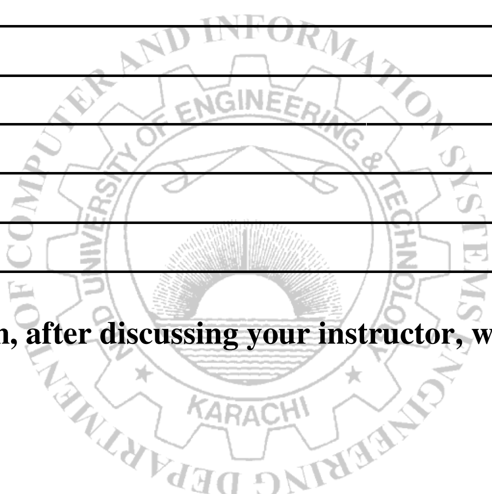
Output from the example; the code is executed by using two threads.

In funcA: this section is executed by thread 0
In funcB: this section is executed by thread 1

EXERCISE:

1. Code the above example programs and write down their outputs.

Output of Program:



2. write a parallel program, after discussing your instructor, which uses Sections Construct.

Program:

Lab Session 13

OBJECT

Sharing of work among threads in an OpenMP program using 'Single Construct'

THEORY

Introduction

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed below.

- **#pragma omp for !$omp do** Distribute iterations over the threads
- **#pragma omp sections !$omp sections** Distribute independent work units
- **#pragma omp single !$omp single** Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the nowait clause .

The Single Construct

The *single construct* is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only. It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another. It can also differ for different single constructs within one application. This construct should really be used when we do not care which thread executes this part of the application,

as long as the work gets done by exactly one thread. The other threads wait at a barrier until the thread executing the single code block has completed.

Format:

```
#pragma omp single [clause ...]  newline

                              private (list)
                          firstprivate (list)
                          nowait

     structured_block
```

Example of the single construct

Only one thread initializes the shared variable a.

```
#include <omp.h>
main()
{
     #pragma omp parallel shared(a,b) private(i)
     {
          #pragma omp single
          {
               a = 10;
               printf("Single construct executed by thread
               %d\n", omp_get_thread_num());
          }
          #pragma omp for
          for (i=0; i<n; i++)
          b[i] = a;
                  } /*-- End of parallel region --*/
          printf("After the parallel region:\n");
          for (i=0; i<n; i++)
          printf("b[%d] = %d\n",i,b[i]);
        }
```

Output from the example, the value of variable *n* is set to 9, and four threads are used.

```
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
```

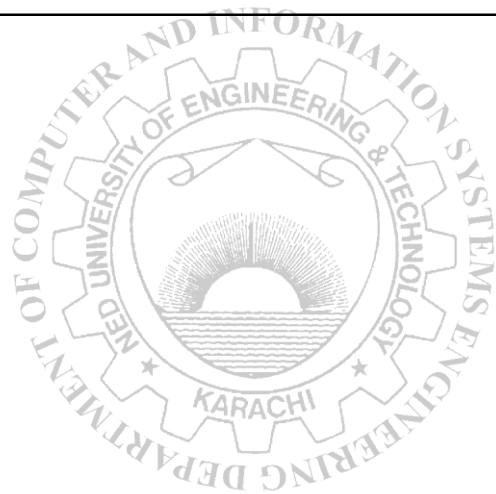
EXERCISE:

1. Code the above example programs and write down their outputs.

Output of Program:

2. Write a parallel program, after discussing your instructor, which uses Single Construct.

Program:



Lab Session 14

OBJECT

Use of Environment Variables in OpenMP API

THEORY

Environment Variables

This Lab session describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs. The names of the environment variables must be upper case. The values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The environment variables are as follows:

- **OMP_SCHEDULE**
- **OMP_NUM_THREADS**
- **OMP_DYNAMIC**
- **OMP_NESTED**
- **OMP_STACKSIZE**
- **OMP_WAIT_POLICY**
- **OMP_MAX_ACTIVE_LEVELS**
- **OMP_THREAD_LIMIT**

OMP_SCHEDULE

The OMP_SCHEDULE environment variable controls the schedule type and chunk size of all loop directives that have the schedule type runtime, by setting the value of the *run-sched-var* ICV.

The value of this environment variable takes the form: *type*[,*chunk*] where

- *type* is one of **static**, **dynamic**, **guided**, or **auto**
- *chunk* is an optional positive integer that specifies the chunk size

Example

Set OMP_SCHEDULE "guided,4"

Set OMP_SCHEDULE "dynamic"

OMP_NUM_THREADS

The OMP_NUM_THREADS environment variable sets the number of threads to use for parallel regions by setting the initial value of the *nthreads-var* ICV. The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of OMP_NUM_THREADS is greater than the number of threads an implementation can support, or if the value is not a positive integer.

Example:

Set OMP_NUM_THREADS 16

OMP_DYNAMIC

The OMP_DYNAMIC environment variable controls dynamic adjustment of the number of threads to use for executing parallel regions by setting the initial value of the *dyn-var* ICV. The value of this environment variable must be true or false. If the environment variable is set to true, the OpenMP implementation may adjust the number of threads to use for executing parallel regions in order to optimize the use of system resources. If the environment variable is set to false, the dynamic adjustment of the number of threads is disabled. The behavior of the program is implementation defined if the value of OMP_DYNAMIC is neither true nor false.

Example:

Set OMP_DYNAMIC true

OMP_NESTED

The OMP_NESTED environment variable controls nested parallelism by setting the initial value of the *nest-var* ICV. The value of this environment variable must be true or false. If the environment variable is set to true, nested parallelism is enabled; if set to false, nested parallelism is disabled. The behavior of the program is implementation defined if the value of OMP_NESTED is neither true nor false.

Example:

Set OMP_DYNAMIC true
Set OMP_NESTED false

OMP_STACKSIZE

The OMP_STACKSIZE environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for the initial thread. The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes, Megabytes, or Gigabytes, respectively. If one of these letters is present, there may be white space between *size* and the letter.
- If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.
- The behavior of the program is implementation defined if OMP_STACKSIZE does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
set     OMP_STACKSIZE 2000500B
set     OMP_STACKSIZE "3000 k "
set     OMP_STACKSIZE 10M
set     OMP_STACKSIZE " 10 M "
set     OMP_STACKSIZE "20 m "
set     OMP_STACKSIZE " 1G"
set     OMP_STACKSIZE 20000
```

OMP_WAIT_POLICY

The OMP_WAIT_POLICY environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the *wait-policyvar* ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. The value of this environment variable takes the form: **ACTIVE | PASSIVE**

The ACTIVE value specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin. The PASSIVE value specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. An OpenMP implementation, may for example, make waiting threads yield the processor to other threads or go to sleep.

Examples:

```
Set     OMP_WAIT_POLICY ACTIVE
Set     OMP_WAIT_POLICY PASSIVE
```

OMP_MAX_ACTIVE_LEVELS

The OMP_MAX_ACTIVE_LEVELS environment variable controls the maximum number of nested active parallel regions by setting the initial value of the *max-active-levels-var* ICV. The value of this environment variable must be a non-negative integer. The behavior of the

program is implementation defined if the requested value of OMP_MAX_ACTIVE_LEVELS is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer.

OMP_THREAD_LIMIT

The OMP_THREAD_LIMIT environment variable sets the number of OpenMP threads to use for the whole OpenMP program by setting the *thread-limit-var* ICV. The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of OMP_THREAD_LIMIT is greater than the number of threads an implementation can support, or if the value is not a positive integer.