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: