Message Passing Interface (MPI)

Lecture Overview

- Advantages of Message Passing
- Background on MPI
- “Hello, world” in MPI
- Key Concepts in MPI
- Basic Communications in MPI
- Simple send and receive program

Advantages of Message Passing

- **Universality : MP model fits well on separate processors connected by fast/slow network. Matches the hardware of most of today's parallel supercomputers as well as network of workstations (NOW)**
- **Expressivity : MP has been found to be a useful and complete model in which to express parallel algorithms. It provides the control missing from data parallel or compiler based models**
- **Ease of debugging : Debugging of parallel programs remain a challenging research area. Debugging is easier for MPI paradigm than shared memory paradigm (even if it is hard to believe)**

Advantages of Message Passing

- **Performance:**
  - This is the most compelling reason why MP will remain a permanent part of parallel computing environment
  - As modern CPUs become faster, management of their caches and the memory hierarchy is the key to getting most out of them
  - MP allows a way for the programmer to explicitly associate specific data with processes and allows the compiler and cache management hardware to function fully
  - Memory bound applications can exhibit superlinear speedup when run on multiple PEs compare to single PE of MP machines

Background on MPI

- **MPI - Message Passing Interface**
  - Library standard defined by committee of vendors, implementers, and parallel programmer
  - Used to create parallel SPMD programs based on message passing
- **Available on almost all parallel machines in C and Fortran**
- **About 125 routines including advanced routines**
- **6 basic routines**

MPI Implementations

- **Most parallel machine vendors have optimized versions**
- **Others:**
  - http://www.mpi.nd.edu/MPI/Mpich
  - http://www-unix.mcs.anl.gov/mpi/mpich/
  - indexold.html
  - GLOBUS:
  - http://www.globus.org/mpi/
  - http://exodus.physics.ucla.edu/appleseed/

Key Concepts of MPI

- **Used to create parallel SPMD programs based on message passing**
- **Normally the same program is running on several different nodes**
- **Nodes communicate using message passing**

Include files

- **The MPI include file**
  - C: mpi.h
  - Fortran: mpif.h (a f90 module is a good place for this)
- **Defines many constants used within MPI programs**
- **In C defines the interfaces for the functions**
- **Compilers know where to find the include files**

Communicators

- **Communicators**
  - A parameter for most MPI calls
  - A collection of processors working on some part of a parallel job
  - MPI_COMM_WORLD is defined in the MPI include file as all of the processors in your job
  - Can create subsets of MPI_COMM_WORLD
  - Processors within a communicator are assigned numbers 0 to n-1

Data types

- **Data types**
  - When sending a message, it is given a data type
  - Predefined types correspond to "normal" types
    - MPI_REAL , MPI_FLOAT -Fortran and C real
    - MPI_DOUBLE PRECISION , MPI_DOUBLE - Fortan and C double
    - MPI_INTEGER and MPI_INT - Fortran and C integer
  - Can create user-defined types

Minimal MPI program

- **Every MPI program needs these…**
  - C version

```
#include <mpi.h> /* the mpi include file */
/* Initialize MPI */
ierr=MPI_Init(&argc, &argv);
/* How many total PEs are there */
ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What node am I (what is my rank? */
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
ierr=MPI_Finalize();
```

In C MPI routines are functions and return an error value

Minimal MPI program

- **Every MPI program needs these…**
  - Fortran version

```
 include 'mpif.h'  ! MPI include file
c     Initialize MPI
      call MPI_Init(ierr)
c     Find total number of PEs
      call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
c     Find the rank of this node
      call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
      ...
      call MPI_Finalize(ierr)
```

In Fortran, MPI routines are subroutines, and last parameter is an error value

MPI “Hello, World”

- **A parallel hello world program**
  - Initialize MPI
  - Have each node print out its node number
  - Quit MPI

C/MPI version of “Hello, World”

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
int main(argc, argv)
int argc;
char *argv[ ];
{
        int myid, numprocs;

        MPI_Init (&argc, &argv) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;
        printf(“Hello from %d\n”, myid) ;
        printf(“Numprocs is %d\n”, numprocs) ;
        MPI_finalize():
}
```

Fortran/MPI version of “Hello, World”

```
program  hello
include ‘mpif.h’
integer myid, ierr, numprocs

call MPI_INIT( ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs,ierr)

write (*,*) “Hello from “, myid
write (*,*) “Numprocs is”, numprocs
call MPI_FINALIZE(ierr)
stop
end
```

Basic Communications in MPI

- **Data values are transferred from one processor to another**
  - One process sends the data
  - Another receives the data
- **Synchronous**
  - Call does not return until the message is sent or received
- **Asynchronous**
  - Call indicates a start of send or received, and another call is made to determine if finished

Synchronous Send

- **MPI_Send: Sends data to another processor**
- **Use MPI_Receive to "get" the data**
- **C**
  - MPI_Send(&buffer,count,datatype, destination,tag,communicator);
- **Fortran**
  - Call MPI_Send(buffer, count, datatype,destination, tag, communicator, ierr)
- **Call blocks until message on the way**

MPI_Send

Call MPI_Send(buffer, count, datatype, destination, tag, communicator, ierr)

- **Buffer: The data**
- **Count : Length of source array (in elements, 1 for scalars)**
- **Datatype : Type of data, for example : MPI_DOUBLE_PRECISION, MPI_INT, etc**
- **Destination : Processor number of destination processor in communicator**
- **Tag : Message type (arbitrary integer)**
- **Communicator : Your set of processors**
- **Ierr : Error return (Fortran only)**

Synchronous Receive

- **Call blocks until message is in buffer**
- **C**
  - MPI_Recv(&buffer,count, datatype, source, tag, communicator, &status);
- **Fortran**
  - Call MPI_ RECV(buffer, count, datatype, source,tag,communicator, status, ierr)
- **Status - contains information about incoming message**
  - C
    - MPI_Status status;
  - Fortran
    - Integer status(MPI_STATUS_SIZE)

Status

- **In C**
  - status is a structure of type MPI_Status which contains three fields MPI_SOURCE, MPI_TAG, and MPI_ERROR
  - status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message
- **In Fortran**
  - status is an array of INTEGERS of length MPI_STATUS_SIZE, and the 3 constants MPI_SOURCE, MPI_TAG, MPI_ERROR are the indices of the entries that store the source, tag, & error
  - status(MPI_SOURCE), status(MPI_TAG), status(MPI_ERROR) contain respectively the source, the tag, and the error code of the received message.

MPI_Recv

Call MPI_Recv(buffer, count, datatype, source, tag, communicator, status, ierr)

- **Buffer: The data**
- **Count : Length of source array (in elements, 1 for scalars)**
- **Datatype : Type of data, for example : MPI_DOUBLE_PRECISION, MPI_INT, etc**
- **Source : Processor number of source processor in communicator**
- **Tag : Message type (arbitrary integer)**
- **Communicator : Your set of processors**
- **Status: Information about message**
- **Ierr : Error return (Fortran only)**

Basic MPI Send and Receive

- **A parallel program to send & receive data**
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive an integer from processor 0
  - Both processors print the data
  - Quit MPI

Simple Send & Receive Program

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>
 /*************************************************************
This is a simple send/receive program in MPI
*************************************************************/
int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;
     MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Simple Send & Receive Program (cont.)

```
tag=1234;
   source=0;
   destination=1;
   count=1;
   if(myid == source){
     buffer=5678;

    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WO
    RLD);
     printf("processor %d  sent %d\n",myid,buffer);
   }
   if(myid == destination){

    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,
    &status);
       printf("processor %d  got %d\n",myid,buffer);
   }
   MPI_Finalize();
}
```

Simple Send & Receive Program

```
      program send_recv
      include "mpif.h"
! This is MPI send - recv program
      integer myid, ierr,numprocs
      integer tag,source,destination,count
      integer buffer
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      tag=1234
      source=0
      destination=1
      count=1
```

Simple Send & Receive Program (cont.)

```
if(myid .eq. source)then
        buffer=5678
        Call MPI_Send(buffer, count, MPI_INTEGER,destination,&
         tag, MPI_COMM_WORLD, ierr)
        write(*,*)"processor ",myid," sent ",buffer
      endif
      if(myid .eq. destination)then
        Call MPI_Recv(buffer, count, MPI_INTEGER,source,&
         tag, MPI_COMM_WORLD, status,ierr)
        write(*,*)"processor ",myid," got ",buffer
      endif
      call MPI_FINALIZE(ierr)
      stop
      end
```

The 6 Basic MPI Calls

- **MPI is used to create parallel programs based on message passing**
- **Usually the same program is run on multiple processors**
- **The 6 basic calls in MPI are:**
  - MPI_INIT(ierr)
  - MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  - MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  - MPI_Send(buffer, count,MPI_INTEGER,destination, tag, MPI_COMM_WORLD, ierr)
  - MPI_Recv(buffer, count, MPI_INTEGER,source,tag, MPI_COMM_WORLD, status,ierr)
  - call MPI_FINALIZE(ierr)