Parallel Programming
Basic MPI

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu
Talk Overview

• Background on MPI
• Documentation
• Hello world in MPI
• Basic communications
• Simple send and receive program
Examples at

http://geco.mines.edu/workshop

mkdir class2
cd class2
wget http://geco.mines.edu/workshop/class2/examples/examples.tgz
tar -xzf examples.tgz
Background on MPI

- MPI - Message Passing Interface
- Library standard defined by a committee of vendors, implementers, & parallel programmers
- Used to create parallel programs based on message passing
- Portable: one standard, many implementations
- Available on almost all parallel machines in C and Fortran
- Over 100 advanced routines but 6 basic
Documentation

- MPI home page (contains the library standard): www.mcs.anl.gov/mp

- Books
  - "MPI: The Complete Reference" by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press (also in Postscript and html)
  - "Using MPI" by Gropp, Lusk and Skjellum, MIT Press

- Tutorials
  - many online, just do a search
MPI Implementations

- Most parallel supercomputer vendors provide optimized implementations
  - LAM
    - www.lam-mpi.org (deprecated)
  - OpenMPI
    - www.open-mpi.org (default on Ra)
MPI Implementations

• MPICH:
  • http://www-unix.mcs.anl.gov/mpi/mpich1/download.html

• MVAPICH & MVAPICH2
  • Infiniband optimized version of MPICH
  • http://mvapich.cse.ohio-state.edu/index.shtml
Key Concepts of MPI

• Used to create parallel programs based on message passing
• Normally the same program is running on several different processors
• Processors communicate using message passing
• Typical methodology:

```
start job on n processors
do i=1 to j
    each processor does some calculation
    pass messages between processor
end do
end job
```
Messages

• Simplest message: an array of data of one type.
• Predefined types correspond to commonly used types in a given language
  • MPI_REAL (Fortran), MPI_FLOAT (C)
  • MPI_DOUBLE_PRECISION (Fortran), MPI_DOUBLE (C)
  • MPI_INTEGER (Fortran), MPI_INT (C)
• User can define more complex types and send packages.
Communicators

- Communicator
  - A collection of processors working on some part of a parallel job
  - Used as a parameter for most MPI calls
  - MPI_COMM_WORLD includes all of the processors in your job
  - Processors within a communicator are assigned numbers (ranks) 0 to n-1
  - Can create subsets of MPI_COMM_WORLD
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
Minimal MPI program

- Every MPI program needs these...

- C version

```c
/* the mpi include file */
#include <mpi.h>
int nPEs,ierr,iam;
/* Initialize MPI */
ierr=MPI_Init(&argc, &argv);
/* How many processors (nPEs) are there? */
ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What processor 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

```fortran
! MPI include file
include 'mpif.h'
integer nPEs, ierr, iam
! Initialize MPI
call MPI_Init(ierr)
! How many processors (nPEs) are there?
call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
! What processor am I (what is my rank)?
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
Exercise 1: Hello World

- Write a parallel “hello world” program
- Initialize MPI
- Have each processor print out “Hello, World” and its processor number (rank)
- Quit MPI
Compiling

- Most everywhere including Ra
  - mpif77 mpif90
  - mpicc mpiCC
- On IBM AIX
  - mpxlif, mpxlif90,
  - mpcc, mpCC
- Most MPI compilers are actually just scripts that call underlying Fortran or C compilers
Running

• Most often you will use a batch system
• Write a batch script file.
• Use the command `mpiexec`
  • You must tell the system how many copies to run
• On some systems you must tell where to run the program
#!/bin/bash

#PBS -q dque
#PBS -N a_long_job_name
#PBS -l nodes=2:ppn=2
#PBS -l walltime=00:5:00
#PBS -o e3d.out
#PBS -e e3d.err
#PBS -A USE300
##PBS -k eo
#PBS -V

cd /gpfs/projects/tkaiser/mpi_tests

cp $PBS_NODEFILE nodes

mpiexec -np 4 example.exe
A More Complex PBS run script

#!/bin/bash

#PBS -q dque
#PBS -N a_long_job_name
#PBS -l nodes=2:ppn=2
#PBS -l walltime=00:5:00
#PBS -o e3d.out
#PBS -e e3d.err
#PBS -A USE300
##PBS -k eo
#PBS -V

cd /gpfs/projects/tkaiser/mpi_tests

cp $PBS_NODEFILE nodes

setenv EXAM `ls *exe`

foreach EXE ($EXAM)
  echo time01 `date`
  echo running $EXE
  setenv OUT `echo $EXE | sed -e "s/exe/out/"
  mpiexec -np 4 ./$EXE > $OUT
  echo time02 `date`
end
Basic Communication

- Data values are transferred from one processor to another
  - One processor 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 receive, and another call is made to determine if finished
Synchronous Send

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

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

- C
  - `MPI_Recv(&buffer,count, datatype, source, tag, communicator, &status);`

- Fortran
  - `Call MPI_RECV(buffer, count, datatype, source,tag,communicator, status, ierr)`

- Call blocks the program until message is in buffer
- Status - contains information about incoming message

- C
  - `MPI_Status status;`

- Fortran
  - `Integer status(MPI_STATUS_SIZE)`
Call MPI_Recv(buffer, count, datatype, source, tag, communicator, status, ierr)

- **Buffer**: The data array to be received
- **Count**: Maximum length of data array (in elements, 1 for scalars)
- **Datatype**: Type of data, for example: MPI_DOUBLE_PRECISION, MPI_INT, etc
- **Source**: Source processor number (within given communicator)
- **Tag**: Message type (arbitrary integer)
- **Communicator**: Your set of processors
- **Status**: Information about message
- **Ierr**: Error return (Fortran only)
Exercise 2 : Basic Send and Receive

- Write 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

c_ex01.c
f_ex01.f
program send_receive
include "mpif.h"
integer myid,ierr,numprocs,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
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
Send and Receive Program C

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[])
{
    int myid, numprocs, tag,source,destination,count, buffer;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;  source=0;  destination=1;  count=1;
    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        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();
}
```
Summary

• 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)
  – MPI_FINALIZE(ierr)
Lots more MPI calls

- In our full week workshop we cover about 30-40 of the calls
- We will talk in detail about more
  - Data Types
  - Sending arrays
  - MPI_Bcast
  - MPI_Reduce
<table>
<thead>
<tr>
<th>Call</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Gather</td>
<td>Gathers values from a group of processes.</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>All processes send data to all processes</td>
</tr>
<tr>
<td>MPI_Alltoallv</td>
<td>All processes send different amount of data to, and receive different</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>Blocks until all processes have reached this routine.</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>Broadcasts a message from the process with rank root to all other</td>
</tr>
<tr>
<td>MPI_Comm_create</td>
<td>Creates a new communicator.</td>
</tr>
<tr>
<td>MPI_Comm_dup</td>
<td>Duplicates an existing communicator with all its cached information.</td>
</tr>
<tr>
<td>MPI_Comm_split</td>
<td>Creates new communicators based on colors and keys.</td>
</tr>
<tr>
<td>MPI_File_close</td>
<td>Closes a file (collective).</td>
</tr>
<tr>
<td>MPI_File_open</td>
<td>Opens a file (collective).</td>
</tr>
<tr>
<td>MPI_File_seek</td>
<td>Updates individual file pointers (noncollective).</td>
</tr>
<tr>
<td>MPI_File_write_at</td>
<td>Writes a file at an explicitly specified offset (blocking, noncollective).</td>
</tr>
<tr>
<td>MPI_File_write_at_all</td>
<td>Writes a file at explicitly specified offsets (blocking, collective).</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>Gathers varying amounts of data from all processes to the root process</td>
</tr>
<tr>
<td>MPI_Get_processor_name</td>
<td>Gets the name of the processor.</td>
</tr>
<tr>
<td>MPI_Group_incl</td>
<td>Produces a group by reordering an existing group and taking only listed</td>
</tr>
<tr>
<td>MPI_Iprobe</td>
<td>Nonblocking test for a message.</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>Starts a standard-mode, nonblocking receive.</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>Starts a standard-mode, nonblocking send.</td>
</tr>
<tr>
<td>MPI_Probe</td>
<td>Blocking test for a message.</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>Reduces values on all processes within a group.</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>Sends data from one task to all tasks in a group.</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>Scatters a buffer in parts to all tasks in a group.</td>
</tr>
<tr>
<td>MPI_Type_commit</td>
<td>Commits a data type.</td>
</tr>
<tr>
<td>MPI_Type_contiguous</td>
<td>Creates a contiguous data type.</td>
</tr>
<tr>
<td>MPI_Type_size</td>
<td>Returns the number of bytes occupied by entries in a data type.</td>
</tr>
<tr>
<td>MPI_Type_struct</td>
<td>Creates a struct data type -- use of this routine is deprecated.</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>Waits for an MPI send or receive to complete.</td>
</tr>
<tr>
<td>MPI_Wtick</td>
<td>Returns the resolution of MPI_Wtime.</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td>Returns an elapsed time on the calling processor.</td>
</tr>
</tbody>
</table>

Some MPI calls we talk about in the weeklong workshop.
MPI Types

- MPI has many different predefined data types
- Can be used in any communication operation
### Predefined types in C

<table>
<thead>
<tr>
<th>C MPI Types</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPIDOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>-</td>
</tr>
</tbody>
</table>
Predefined types in Fortran

<table>
<thead>
<tr>
<th>Fortran MPI Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>-</td>
</tr>
</tbody>
</table>
Sending/Receiving Arrays

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

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

• Buffer becomes an array
• Count is the length of the array in elements of datatype
Sending/Receiving Arrays

MPI_Send(buffer, count, datatype, destination, tag, communicator);

MPI_Recv(buffer, count, datatype, source, tag, communicator, &status);

• Buffer becomes an array (pointer so you don’t need the “&”)
• Count is the length of the array in elements of datatype
Exercise 3: Basic Send and Receive

• Write a parallel program to send & receive data
• Initialize MPI
• Have processor 0 send 3 reals to processor 1
• Have processor 1 receive 3 reals from processor 0
• Both processors print the data
• Quit MPI
MPI Broadcast call: MPI_Bcast

• All nodes call MPI_Bcast
• One node (root) sends a message all others receive the message

• C
  – MPI_Bcast(&buffer, count, datatype, root, communicator);

• Fortran
  – call MPI_Bcast(buffer, count, datatype, root, communicator, ierr)

• Root is node that sends the message
Exercise 4: Broadcast

• Write a parallel program to broadcast data using MPI_Bcast
  – Initialize MPI
  – Have processor 0 broadcast an integer
  – Have all processors print the data
  – Quit MPI
Scatter Operation using MPI_Scatter

- Similar to Broadcast but sends a section of an array to each processors

Data in an array on root node:

\[ A(0) \quad A(1) \quad A(2) \quad \ldots \quad A(N-1) \]

Goes to processors:

\[ P_0 \quad P_1 \quad P_2 \quad \ldots \quad P_{n-1} \]
MPI_Scatter

• C
  ```c
  int MPI_Scatter(&sendbuf, sendcnts, sendtype,
                   &recvbuf, recvcnts, recvtype, root, comm );
  ```

• Fortran
  ```fortran
  MPI_Scatter(sendbuf,sendcnts,sendtype,
              recvbuf,recvcnts,recvtype,root,comm,ierror)
  ```

• Parameters
  – `Sendbuf` is an array of size (number processors*sendcnts)
  – `Sendcnts` number of elements sent to each processor
  – `Recvcnts` number of elements obtained from the root processor
  – `Recvbuf` elements obtained from the root processor, may be an array
Reduction Operations

- Used to combine partial results from all processors
- Result returned to root processor
- Several types of operations available
- Works on single elements and arrays
MPI routine is MPI_Reduce

- **C**
  - `int MPI_Reduce(&sendbuf, &recvbuf, count, datatype, operation, root, communicator)`

- **Fortran**
  - `call MPI_Reduce(sendbuf, recvbuf, count, datatype, operation, root, communicator, ierr)`

- **Parameters**
  - Like MPI_Bcast, a root is specified.
  - Operation is a type of mathematical operation
Operations for MPI_Reduce

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
Global Sum with MPI_Reduce

C

double sum_partial, sum_global;
sum_partial = ...;
ierr = MPI_Reduce(&sum_partial, &sum_global,
1, MPI_DOUBLE_PRECISION,
MPI_SUM, root,
MPI_COMM_WORLD);

Fortran

double precision sum_partial, sum_global
sum_partial = ...
call MPI_Reduce(sum_partial, sum_global,
1, MPI_DOUBLE_PRECISION,
MPI_SUM, root,
MPI_COMM_WORLD, ierr)
Exercise 5: Global Sum with MPI_Reduce

- Write a program
- Uses MPI_Scatter to send N values to all nodes
- Does a local sum
- Calls MPI_Reduce to do a global sum

c_ex06.c
f_ex06.f