NPACI Parallel Computing Workshop:

Introduction to Parallel Programming with MPI

Giri Chukkapalli (giri@sdsc.edu)
Amit Majumdar (majumdar@sdsc.edu)
San Diego Supercomputer Center
Introduction

• Message Passing Interface for accessing non-local memory

• Standard defined by committee of vendors, implementors, and parallel programmers

• You will learn how to transfer data on MPP machines using a Single Program Multiple Data (SPMD) model

• NPACI MPI Documentation: http://www.npaci.edu/Resources/Applications/MPI

• MPI home page: http://www.mcs.anl.gov/mpi

• A good companion textbook is "Using MPI" by Gropp, Lusk, and Skjellum.
Advantages of Message-passing Model

• Gives programmer ability to manage the memory hierarchy
• Portability to many architectures
• Easier to create a deterministic program
• Simplifies debugging
The Message Passing Interface

- Late 1980s: vendors had unique libraries
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
- 1992: Work on MPI standard begun
- 1994: Version 1.0 of MPI standard
- 1997: Version 2.0 of MPI standard
- Today: MPI is dominant message passing library standard
Topics

• Overview
• Initialization and termination of MPI
• Communications:
  – Point to point
  – Collective
  – Global operations
  – Synchronous/Asynchronous
• Communicators and groups
• MPI Derived Data Types
• (Example codes)
Overview

- MPI moves data between Processor Element (PE) Memories.
- All PEs are executing the same code (SPMD).
Vocabulary

- Processor Element (PE)
- Rank
- Point-to-Point
- Collective
- Blocking
- Buffer
MPI Conventions

• C: constants defined in mpi.h
  Syntax:  ierr = MPI_Operation (...)
Initialization

• Every MPI program needs these initialization lines.
• Need to know number of PEs (nPEs) and PE number (Rank)
• Initial group of PEs is identified by MPI_Comm_world parameter
• PE ranks range from 0 to nPEs-1
Initialization (cont.)

• C

```c
#include <mpi.h>
#include <stdio.h>

/* Initialize MPI */
MPI_Init(&argc, &argv);
/* How many total PEs are there */
MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What node am I (what is my rank? */
MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
MPI_Finalize();
```
Exercise 1: Hello World

- write a parallel hello world program
  - Initialize MPI
  - have each node print out its node number
Local Variables

```c
int main (int argc, char *argv[]) {
    int i;
    int id; /* Process rank */
    int p;  /* Number of processes */
```

- **Include** `argc` and `argv`: they are needed to initialize MPI
- **One copy of every variable for each process running this program**
Communicators

• Communicator: opaque object that provides message-passing environment for processes

• MPI_COMM_WORLD
  – Default communicator
  – Includes all processes

• Possible to create new communicators
  – Will do this in Chapters 8 and 9
Communicator

MPI_COMM_WORLD

Processes

Ranks

Communicator Name

0 2 1 3 4 5

1 2 3 4
Determine Number of Processes

MPI_Comm_size (MPI_COMM_WORLD, &p);

• First argument is communicator
• Number of processes returned through second argument
Determine Process Rank

MPI_Comm_rank (MPI_COMM_WORLD, &id);

• First argument is communicator
• Process rank (in range 0, 1, ..., p-1) returned through second argument
Replication of Automatic Variables
What about External Variables?

int total;

int main (int argc, char *argv[]) {
    int i;
    int id;
    int p;
    ...

Where is variable total stored?
Shutting Down MPI

MPI_Finalize();

- Call after all other MPI library calls
- Allows system to free up MPI resources
Compiling MPI Programs

mpicc -O -o foo foo.c

• mpicc: script to compile and link C+MPI programs
• Flags: same meaning as C compiler
  – -O  — optimize
  – -o  <file>  — where to put executable
Running MPI Programs

- `mpirun` -np `<p> <exec> <arg1> ...`
  - `--np <p>` — number of processes
  - `<exec>` — executable
  - `<arg1>` ... — command-line arguments
Specifying Host Processors

• File `.mpi-machines` in home directory lists host processors in order of their use

• Example `.mpi_machines` file contents

  node480.snu.ac.kr
  node481.snu.ac.kr
  node482.snu.ac.kr
  node483.snu.ac.kr
Enabling Remote Logins

- MPI needs to be able to initiate processes on other processors without supplying a password
- Each processor in group must list all other processors in its `.rhosts` file; e.g.,
  
```text
node480.snu.ac.kr eduuserxx
node481.snu.ac.kr eduuserxx
node482.snu.ac.kr eduuserxx
node483.snu.ac.kr eduuserxx
```
Communications

Definitions

• Bytes transferred from one processor to another
• Specify destination/source, data buffer, number of elements, datatype, message ID (called a tag) etc.
• Synchronous send: send call does not return until the message is sent
• Asynchronous send: send call returns immediately, send occurs during other calculation ideally
Communications (cont..)

- Synchronous receive: receive call does not return until the message has been received (may involve a significant wait)
- Asynchronous receive: receive call returns immediately. When received data is needed, call a wait subroutine
- Asynchronous communication used in attempt to overlap communication with computation
Synchronous Send

• Blocking send a message to a processor
  – C

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

• Execution blocked until message in channel
MPI_Send Parameters

- **buffer**: Beginning address of data
- **count**: Length of source array (in elements)
- **datatype**: Type of data, for example: MPI_DOUBLE_PRECISION, MPI_INT, etc
- **destination/source**: Logical processor number of destination/source processor
- **tag**: Message type (arbitrary integer)
- **communicator**: Signifies a set of processors to whom the message is sent
Synchronous Receive

- Blocking receive of a message from another processor
  - C
    
    ```c
    -MPI_Recv(&buffer, COUNT, datatype, source, tag, communicator,&status);
    ```
Send - Receive Example

Pseudo code : Increment Integer & Pass On Value

!Pass Data {0->1, 1->2,...,npes-1 -> 0} add "1" to data

dimension istat(MPI_STATUS_SIZE)
ides = mod(mype+1, npes);  isrc = mod(mype+npes-1,npes);

if(ipe==0) then
  call mpi_Send(1,1,MPI_INTEGER, ides,9,icomm)
  call mpi_Recv(j,1,MPI_INTEGER, isrc,9,icomm, istat)
else
  call mpi_Recv(j,1,MPI_INTEGER, isrc,9,icomm, istat)
i=j+1
  call mpi_Send(i,1,MPI_INTEGER, ides,9,icomm)
endif
MPI types

• Several MPI types are predefined in the include files
• Some C MPI data types (defined in mpi.h):

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</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>
</tbody>
</table>
### MPI types (cont..)

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_UNSIGNED</strong></td>
<td>unsigned int</td>
</tr>
<tr>
<td><strong>MPI_UNSIGNED_LONG</strong></td>
<td>insigned long int</td>
</tr>
<tr>
<td><strong>MPI_FLOAT</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>MPI_DOUBLE</strong></td>
<td>double</td>
</tr>
<tr>
<td><strong>MPI_LONG_DOUBLE</strong></td>
<td>long double</td>
</tr>
<tr>
<td><strong>MPI_BYTE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MPI_PACKED</strong></td>
<td></td>
</tr>
</tbody>
</table>
Wildcards

• Allow you to not necessarily specify a tag or a source

• Example:

  MPI_Recv(buffer, count, MPI_INTEGER, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)

• MPI_ANY_SOURCE and MPI_ANY_TAG are wildcards

• The status structure can be used to clarify wildcard options
The status parameter

- The status parameter returns additional information
  - Parameter of some MPI routines
  - Additional Error status information
  - Additional information with wildcard parameters

- C declaration : a predefined struct

  MPI_Status status;
Accessing status information

status fields

- The tag of a received message
  C : status.MPI_TAG

- The source of a received message
  C : status.MPI_SOURCE

- The error code of the MPI call
  C : status.MPI_ERROR
Broadcast

• One node sends a message (root)

• All others receive the message in the same memory space.

• Execution blocked until All processors arrive to broadcast call.
Broadcast (cont..)

- Automatically acts as a synchronizing point.

```c
MPI_Bcast(&buffer,COUNT,datatype,
          root, communicator);
```
Collective Communications

• MPI_Gather

• A global exchange of data
  – Each processor sends different data to the root processor and it collects it into an array

• C:
  int MPI_Gather (&sendbuf, sendcount, datatype,&recvbuf, recvcount,datatype, root, communicator)
Gather

MPI_Gather(sdata, iscount, istype, rdata, ircount, irtype, iroot, icomm)
Scatter

• A global exchange of data
  – Root processor breaks up an array and sends parts to each processor

• C:
  int MPI_Scatter(&sendbuf, sendcount, datatype, &recvbuf, recvcount, datatype, root, communicator)
MPI_Gather/Scatter

PE0

PE1

PE2

PE3

Scatter

Gather
Parallel Reductions

Definitions

- Used to combine partial results from all processors
- Called a parallel prefix or parallel reduction
- Final result can be available on a particular processor or on all processors
- Processors synchronize
MPI Reduction Subroutines

MPI routine is MPI_Allreduce

- C

```c
int MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, operation, communicator)
```

- Also available: MPI_Reduce().
  - Like MPI_Bcast, a root is specified.
  - Results are only sent back to the root node.
Types of Global Operations

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_Op_create can be used to bind a user-defined global operation to an op handle.
Example of Computing a Global Sum

- Each processor has variables `sum_partial` and `sum_global`
- Value of `sum_global` updated on all processors

```c
double sum_partial, sum_global;
sum_partial = ...;
ierr = MPI_Allreduce(&sum_partial, &sum_global, 1, MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD);
```
Parallel Programming Example: Dot Product

Algorithm:

- Initialize mpi, get mynode and numnodes
- Map vectors to nodes
- Initialize local vectors
- Compute local sums
- Compute global sum
Initialize mpi, get mynode and numnodes

MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &iam);
MPI_Comm_size(MPI_COMM_WORLD, &PEs);
Map vectors to nodes

if (mod(n, PEs) != 0) {
    write(*,*) 'Number of PEs must evenly divide n= ', n
}
ishare = n / PEs
i1 = iam * ishare + 1
i2 = (iam+1) * ishare
Initialize local vectors

```plaintext
for(i = i1, i<=i2, i++) {
    ilocal = i-i1+1
    x(ilocal) = i
    y(ilocal) = 1.0
}
```
Compute local sums

\[
\text{sum} = 0.0;
\]

\[
\text{for}(i = i1, i <= i2, i++) \{
\]
\[
\text{ilocal} = i - i1 + 1;
\]
\[
\text{sum} = \text{sum} + 
\]
\[
x(\text{ilocal}) \times y(\text{ilocal});
\]
\[
\}
\]
Compute global sum

MPI_Allreduce(sum, sum_global, 1,
MPI_REAL, MPI_SUM, MPI_COMM_WORLD)

if (iam==0)
    printf("The dot product is" %f,
sum_global);
C Example code

(/usr/local/examples/dot.c)

/*
This program is provided by the San Diego Supercomputer Center to the attendees of its parallel workshops. You are welcome to copy this program and share it with others as long as you keep this header paragraph intact.
Author: Ken Steube, San Diego Supercomputer Center

Compute a dot product in parallel.
Given vectors X and Y of length N, compute
X(1)*Y(1) + X(2)*Y(2) + ... + X(N)*Y(N)

Each PE gets an equal part of X and Y, and it computes its share of the sum. The partial sums are combined to form a global sum. The PEs do not allocate enough space to store the global vector. Each PE can work with at most PEmax elements of each vector.
*/
C Example code (cont.)

```c
#include <stdio.h>
#include <mpi.h>
/* Size of PE's share of the vector */
#define PEmax 1024
int main(argc, argv)
    int argc;
    char **argv;
{
    int n, il, i2, ishare, i, ilocal, iam, PEs, ierr;
    double x[PEmax], y[PEmax], sum, sum_global;
    /* Size of global vector */
    n = 1024;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &PEs);
```
C Example code (cont.)

/* Figure this PE's share */
if (n % PEs != 0) {
    printf("Number of PEs must evenly divide n=%i\n", n);
    exit(1);
}

ishare = n / PEs;
i1 = iam * ishare + 1;
i2 = (iam+1) * ishare;

/* Make sure each PE has allocated enough memory */
if (ishare > PEmax) {
    if (iam == 0) fprintf(stderr, "Increase PEmax to %i\n", ishare);
    exit(1);
}

C Example code (cont.)

/* Define the inputs */
for (i=i1; i<=i2; i++) {
    ilocal=i-i1+1;
    x[ilocal]=(double) i;
    y[ilocal]=(double) 1.0;
}
/* Compute the local sums */
sum = 0.0;
for (i=i1; i<=i2; i++) {
    ilocal=i-i1+1;
    sum = sum + x[ilocal] * y[ilocal];
}
/* Form the global sum of each PE's contribution */
ierr = MPI_Allreduce(&sum, &sum_global, 1, MPI_DOUBLE,
                    MPI_SUM, MPI_COMM_WORLD);
if (iam == 0) {
    printf("The dot product is %.1lf\n", sum_global);
    printf("The answer should be %i\n", n*(n+1)/2);
}

MPI_Finalize();
}
Asynchronous Communication

MPI_Isend

- Non Blocking send
  - C

  ```c
  int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  ```

- Parameters

  [ IN buf] initial address of send buffer (choice)
  [ IN count] number of elements in send buffer (integer)
  [ IN datatype] datatype of each send buffer element (handle)
  [ IN dest] rank of destination (integer)
  [ IN tag] message tag (integer)
  [ IN comm] communicator (handle)

  [ OUT request] communication request (handle)
Asynchronous Communication (cont.)

Receive

- Non Blocking receive

  - C

```c
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- Parameters

  [ OUT buf] initial address of receive buffer (choice)
  [ IN count] number of elements in receive buffer (integer)
  [ IN datatype] datatype of each receive buffer element (handle)
  [ IN source] rank of source (integer)
  [ IN tag] message tag (integer)
  [ IN comm] communicator (handle)
  [ OUT request] communication request (handle)
Asynchronous Communication (cont.)

MPI_Wait

- Used to complete a nonblocking communication
- The completion of a send operation indicates that the sender is now free to update the locations in the send buffer
- The completion of a receive operation indicates that the receive buffer contains the received message

```c
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- **Parameters**
  - [INOUT request] request (handle)
  - [OUT status] status object (Status)
Asynchronous Communication (cont.)

MPI_Test
• Similar to MPI_Wait, but does not block.
• Value of flags signifies whether a message has been delivered

- C

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

- Parameters

- [ INOUT request] communication request (handle)
- [ OUT flag] true if operation completed (logical)
- [ OUT status] status object (Status)
Asynchronous Communication (cont.)

- Non-blocking send example that acts like MPI_Wait

```fortran
    call MPI_Isend (buffer, count, datatype, dest, tag, com, request, ierr)
    (do computation that doesn’t modify buffer)
10   call MPI_Test (request,flag, status, ierr)
    if (.not. flag) go to 10
```
Example
Inner loop of block $A \times B = C$.

$a$, $b$, and $c$ are sub-blocks of $A$, $B$, and $C$.

$a$ has two planes.

Each PE multiplies its $b$ into every PE’s $a$ and accumulates in $c$.

$a$ is copied to “next” PE during a matrix multiply while alternate plane receives $a$. 
ides = mod(mype+1, npe) !next PE
isrc = mod(mype-1+npe,npe)

do i = 0, npes-1  !v switch planes
  m0=mod(i,2); m1=mod(i+1,2)
  call mpi_irecv(a(1,1,m0), NN,ityp, ides, i,icom, ireqr, ir)
  call mpi_isend(a(1,1,m1), NN,ityp, isrc, i,icom, ireqs, ir)
  call sgemm('n','n', N,N,N, one, a(1,1,m1),N, b, N,&
          one, c, N) !MxM
  call mpi_wait(ireqr, istat, ir) !wait for completion
  call mpi_wait(ireqs, istat, ir)
endo
Asynch. Comm. (Barriers)

- **MPI_BARRIER** blocks the caller until all members in the communicator have called it.
- **Used as a synchronization tool.**
  - C
    ```c
    int MPI_BARRIER(MPI_Comm comm )
    ```

- **Parameters**
  - [ IN comm] communicator (handle)
Asynch. Comm. (MPI_Probe)

- MPI_Probe allows incoming messages to be checked for, without actually receiving them.
- The user can then decide how to receive the data.
- Useful when different action needs to be taken depending on the "who, what, and how much" information of the message.

- C

  ```c
  int MPI_Probe(int source, int tag, MPI_Comm comm,
                MPI_Status *status)
  ```

- Parameters

  - [ IN source] source rank, or MPI_ANY_SOURCE (integer)
  - [ IN tag] tag value, or MPI_ANY_TAG (integer)
  - [ IN comm] communicator (handle)
  - [ OUT status] status object (Status)
Asynch. Comm. (MPI_Get_count)

• Returns the number of entries received. (Again, we count entries, each of type datatype, not bytes.)

• The datatype argument should match the argument provided by the receive call that set the status variable.

− C

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                   int *count)
```

• Parameters

[ IN status] return status of receive operation (Status)
[ IN datatype] datatype of each receive buffer entry (handle)
[ OUT count] number of received entries (integer)
Communicators

MPI_Comm_create

- `MPI_Comm_create` creates a new communicator `newcomm` with group members defined by a group data structure.

```c
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

- **Parameters**
  - [ IN comm] communicator (handle)
  - [ IN group] Group, which is a subset of the group of comm (handle)
  - [ OUT newcomm] new communicator (handle)

- **So how do you define a group?**
Communicators (cont.)

MPI_Comm_group

- MPI_Comm_group returns in group a handle to the group of comm.
  
  - C

  int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

• Parameters

  [ IN comm] communicator (handle)

  [ OUT group] group corresponding to comm (handle)
Communicators (cont.)

MPI_Comm_incl

- MPI provides several functions to manipulate existing groups.
- The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks rank[0], ..., rank[n-1];

```c
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

- Parameters
  - [ IN group] group (handle)
  - [ IN n] number of elements in array ranks (and size of newgroup) (integer)
  - [ IN ranks] ranks of processes in group to appear in newgroup (array of integers)
  - [ OUT newgroup] new group derived from above, in the order defined by ranks (handle)
Communicators (cont.)

MPI_Comm_excl

- The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0], ... , ranks[n-1]

- Parameters

  int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

- Parameters

  [ IN group] group (handle)
  [ IN n] number of elements in array ranks (integer)
  [ IN ranks] array of integer ranks in group not to appear in newgroup
  [ OUT newgroup] new group derived from above, preserving the order defined by group (handle)
MPI Derived Data Types

• Can create derived data types

b(1) = 3; b(2) = 10; b(3) = 2
d(1) = 0; d(2) = 24; d(3) = 64
t(1) = MPI_DOUBLE_PRECISION;
t(2) = MPI_INTEGER; t(3) = MPI_LOGICAL

call MPI_TYPE_STRUC(3,b,d,t,
MPI_CHARLES,mpi_err)

call MPI_TYPE_COMMIT(MPI_CHARLES,
mpi_err)
MPI TIME

MPI_Wtime

- MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.
  - C
    
    ```
    double MPI_Wtime(void)
    ```

- Parameters
  None.
MPI Time (cont.)

• Example Usage

```c
{ double starttime, endtime;
  starttime = MPI_Wtime();
  .... stuff to be timed ....
  endtime = MPI_Wtime();
  printf("That took %f seconds\n", endtime-starttime);
}
```
SUMMARY

• A message passing standard
• Available on most Unix machines
• Most Programs can be written using only few calls
• Large number of calls for "special" functions and "parallel library" developers
Exercise 4

• Again, let's practice on the pre-existing integration code

• Practice non-blocking communication:
  – Replace MPI_Sends with MPI_ISends
  – Use MPI_Wait and MPI_Irecv to receive messages
  – Question: Is this a good scenario to use non-blocking calls?

• Use MPI_Probe and MPI_Get_count to print out size, tag and source of each message when it is received
Exercise 5

• Use MPI_Wtime to calculate the total execution time