Introduction to MPI Programming

Winter 2007 Distributed Computing Workshop
“A Consortium to Promote Computational Science and High Performance Computing”

By Yaohang Li, Ph.D.
Department of Computer Science
North Carolina A&T State University
yaohang@ncat.edu

Software Characteristics

• Scientific Computing Application
  – Computation-bounded
    • Require large amount of CPU time
  – Communication-bounded
    • Require large amount of data communication between processes
  – Computation and Communication
    • Require large amount of CPU time and large amount of data communication between processes
The Message-Passing Model

• A process
  – An execution of a program
  – Each process has (traditionally) a program counter and address space.
• Interprocess Communication (IPC)
  – communication among processes, which have separate address spaces.
  – Including
  • Synchronization
  • Movement of data from one process’s address space to another’s.

Introduction to MPI

• MPI - Message Passing Interface
  – Library standard
    • defined by a committee of vendors, implementers, and parallel programmers
    • used for message-passing model in parallel computing
  – History
    • MPI-1: 92-94
    • MPI-2: 95-97
  – 100% portable: one standard, many implementations
    • MPI-CH
    • LAM-MPI
    • intrinsic MPI
  – Available on almost all parallel machines in C and Fortran
  – Over 100 advanced routines but 6 basic
    • MPI
    • Most parallel programs only use the 6 basic routines
Compiling MPI Programs

• Compile an MPI program
  – C
    • mpicc
  – Fortran
    • mpif77
• For large programs, it is ideal to make use of a makefile
• For distributed version, it is ideal to make use of an autoconfig program

Running MPI Programs

• General Case
  – mpirun –np N executable
    • mpirun indicate that you are using the MPI environment
    • -np option specifies the number of processes you like to use
• MPICH
  – mpiexec
• LAM-MPI
  – mpirun c0-7 executable
    • allows to run parallel jobs on specific nodes
Structure of MPI Program

- MPI programs are Single Program Multiple Data
- All MPI programs must include a header file.
  - C: mpi.h
  - Fortran: mpif.h
- All MPI programs must call MPI_INIT as the first MPI call
  - MPI_INIT establishes the MPI environment
  - Only one invocation of MPI_INIT can occur in each program
- All MPI programs must call MPI_FINALIZE as the last call
  - Exits MPI
  - No calls to MPI can be made after MPI_FINALIZE is called

A Minimal MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
A Minimal MPI Program (Fortran)

```fortran
program main
include 'mpi.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Error Handling

- By default, an error causes all processes to abort.
- Error returns
  - In C, MPI functions return error codes or `MPI_SUCCESS`
  - In Fortran, all MPI calls are to subroutines, with a place for the return code in the last argument.
  - In C++, exceptions are thrown (MPI-2)
- The user can cause routines to return (with an error code) instead.
- A user can also write and install custom error handlers.
- Libraries might want to handle errors differently from applications.
Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - `MPI_Comm_size` reports the number of processes.
  - `MPI_Comm_rank` reports the rank, a number between 0 and size-1, identifying the calling process.

Better Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```
Better Hello (Fortran)

```fortran
program main
  include 'mpif.h'
  integer ierr, rank, size

  call MPI_INIT( ierr )
  call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
  call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
  print *, 'I am ', rank, ' of ', size
  call MPI_FINALIZE( ierr )
end
```

Communicators

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

• Data Types
  – When sending a message, it is given a data type
  – Predefined types correspond to “normal” types
    • MPI_REAL, MPI_FLOAT (Fortran Real and C float)
    • MPI_DOUBLE_PRECISION, MPI_DOUBLE (Fortran double precision and C double)
    • MPI_INTEGER and MPI_INT (Fortran and C integer)
  – Can create user-defined type

Point-to-Point Communication

• Basic communication in message passing libraries
  – Data values are transferred from one process to another
    • One process sends the data
    • Another receives the data
**MPI_Send and MPI_Recv**

**MPI_Send**

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

- **Input**
  - buf - initial address of send buffer (choice)
  - count - number of elements in send buffer (nonnegative integer)
  - datatype - datatype of each send buffer element (handle)
  - dest - rank of destination (integer)
  - tag - message tag (integer)
  - comm - communicator (handle)

**MPI_Recv**

```c
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

- **Output**
  - buf - initial address of receive buffer
  - status - status object, provides information about message received; status is a structure of type MPI_Status, the element status.MPI_SOURCE is the source of the message received, and the element status.MPI_TAG is the tag value.

- **Input**
  - count - maximum number of elements in receive buffer (integer)
  - datatype - datatype of each receive buffer element (handle)
  - source - rank of source (integer)
  - tag - message tag (integer)
  - comm - communicator (handle)

---

**A Simple Send and Receive Program**

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

/***************************************************************************/
This is a simple send/receive program in MPI

int main(argc, argv)
    int argc;
    char *argv[];
{
    int myid;
    int tag, source, destination, count;
    int buffer;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    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();
}
```
MPI is Simple

• Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV

• Point-to-point (send/recv) isn’t the only way...

Introduction to Collective Operations in MPI

• Collective operations are called by all processes in a communicator.
• MPI_BCAST distributes data from one process (the root) to all others in a communicator.
• MPI_REDUCE combines data from all processes in communicator and returns it to one process.
• In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency.
MPI_BCAST

- MPI_BCAST( buffer, count, datatype, root, comm )
  - [ INOUT buffer] starting address of buffer (choice)
  - [ IN count] number of entries in buffer (integer)
  - [ IN datatype] data type of buffer (handle)
  - [ IN root] rank of broadcast root (integer)
  - [ IN comm] communicator (handle)

- C prototype
  int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- Fortran prototype
  MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
  <type> BUFFER(*)
  INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

Illustration of MPI_BCAST
MPI_REDUCE

- MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
  - [ IN sendbuf] address of send buffer (choice)
  - [ OUT recvbuf] address of receive buffer (choice, significant only at root)
  - [ IN count] number of elements in send buffer (integer)
  - [ IN datatype] data type of elements of send buffer (handle)
  - [ IN op] reduce operation (handle)
    - Predefined operations including MPI_MAX, MPI_MIN and MPI_SUM
  - [ IN root] rank of root process (integer)
  - [ IN comm] communicator (handle)
  - C Prototype:
    - int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
data type, MPI_Op op, into root, MPI_Comm comm)
  - Fortran Prototype
    - MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
      COMM, IERROR)
      - <type> SENDBUF(*), RECVBUF(*)
      - INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_Scatter

- int MPI_Scatter( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int
  recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
  - Input Parameters
    - sendbuf address of send buffer (choice, significant only at root)
    - sendcount number of elements sent to each process (integer, significant
      only at root)
    - sendtype data type of send buffer elements (significant only at root)
      (handle)
    - recvcount number of elements in receive buffer (integer)
    - recvtype data type of receive buffer elements (handle)
    - root rank of sending process (integer)
    - comm communicator (handle)
  - Output Parameters
    - recvbuf address of receive buffer (choice)
MPI_Gather

- Gathers together values from a group of processes
- 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 (choice)
    - sendcount number of elements in send buffer (integer)
    - sendtype data type of send buffer elements (handle)
    - recvcount number of elements for any single receive (integer, significant only at root)
    - recvtype data type of recv buffer elements (significant only at root) (handle)
    - root rank of receiving process (integer)
    - comm communicator (handle)
  - Output Parameters
    - recvbuf address of receive buffer (choice, significant only at root)

Illustration of MPI_Gather and MPI_Scatter
MPI_ALLGATHER

- Gathers data from all tasks and distribute it to all
- int MPI_Allgather ( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm )
  - Input Parameters
    - sendbuf starting address of send buffer (choice)
    - sendcount number of elements in send buffer (integer)
    - sendtype data type of send buffer elements (handle)
    - recvcount number of elements received from any process (integer)
    - recvtype data type of receive buffer elements (handle)
    - comm communicator (handle)
  - Output Parameters
    - recvbuf address of receive buffer (choice)

Illustration of MPI_ALLGATHER
MPI_ALLTOALL

- Sends data from all to all processes
- int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvct, MPI_Datatype recvtype, MPI_Comm comm )
  - Input Parameters
    - sendbuf starting address of send buffer (choice)
    - sendcount number of elements to send to each process (integer)
    - sendtype data type of send buffer elements (handle)
    - recvct number of elements received from any process (integer)
    - recvtype data type of receive buffer elements (handle)
    - comm communicator (handle)
  - Output Parameters
    - recvbuf address of receive buffer (choice)

Illustration of MPI_ALLTOALL

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₀</td>
<td>B₁</td>
<td>B₂</td>
<td>B₃</td>
<td>B₄</td>
<td>B₅</td>
</tr>
<tr>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
<td>C₃</td>
<td>C₄</td>
<td>C₅</td>
</tr>
<tr>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>D₄</td>
<td>D₅</td>
</tr>
<tr>
<td>E₀</td>
<td>E₁</td>
<td>E₂</td>
<td>E₃</td>
<td>E₄</td>
<td>E₅</td>
</tr>
<tr>
<td>F₀</td>
<td>F₁</td>
<td>F₂</td>
<td>F₃</td>
<td>F₄</td>
<td>F₅</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A₀</th>
<th>B₀</th>
<th>C₀</th>
<th>D₀</th>
<th>E₀</th>
<th>F₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>B₁</td>
<td>C₁</td>
<td>D₁</td>
<td>E₁</td>
<td>F₁</td>
</tr>
<tr>
<td>A₂</td>
<td>B₂</td>
<td>C₂</td>
<td>D₂</td>
<td>E₂</td>
<td>F₂</td>
</tr>
<tr>
<td>A₃</td>
<td>B₃</td>
<td>C₃</td>
<td>D₃</td>
<td>E₃</td>
<td>F₃</td>
</tr>
<tr>
<td>A₄</td>
<td>B₄</td>
<td>C₄</td>
<td>D₄</td>
<td>E₄</td>
<td>F₄</td>
</tr>
<tr>
<td>A₅</td>
<td>B₅</td>
<td>C₅</td>
<td>D₅</td>
<td>E₅</td>
<td>F₅</td>
</tr>
</tbody>
</table>

alltoall
Example: PI in C -1

/* Example program to calculate the value of pi by
integrating f(x) = 4 / (1 + x^2). */

#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if (myid == 0) {
    printf("Enter the number of intervals: ");
    scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n == 0) break;
  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (myid == 0)
    printf("pi is approximately %.16f, Error is
%.16f\n", pi, fabs(pi - PI25DT));
  MPI_Finalize();
  return 0;
}
Timing

• MPI Timing
  – Performance Evaluation
  – Debugging
• MPI Prototype
  – MPI_WTIME()
    • returns a floating-point number of seconds
    • wall-clock time
  – C: double MPI_Wtime(void)
  – Fortran: DOUBLE PRECISION MPI_WTIME(

PI Program with Timing (I)

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

int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    double mytime;
    char str[100];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {
        printf("Enter the number of intervals: ");
        scanf("%d",&n);
    }
```
MPI Program with Timing (II)

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
mytime=MPI_Wtime();
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
x = h * ((double)i - 0.5);
sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
mytime=MPI_Wtime()-mytime;
fprintf(stderr,"Computation time at process %d: %f\n",myid,mytime);
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0)
printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
MPI_Finalize();
return 0;

Advantages of MPI Programming

• 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

• Ease of Debugging
  – Debugging of parallel programs remains a challenging research area
  – Debugging is easier in MPI paradigm than shared memory paradigm
When not to use MPI?

• Regular computation matches High Performance Fortran
• Solution (e.g., library) already exists
• Require Fault Tolerance
  – Sockets
• Require Security
• Distributed Computing
  – CORBA, DCOM, etc.

Summary

• MPI Programming
  – Programming Structure
  – Environment
  – Point-to-Point Communication
  – Collective Operations
  – Timing
• Running a MPI Program
Online resources

- [http://www.erc.msstate.edu/mpi](http://www.erc.msstate.edu/mpi)
- [http://www.epm.orl.gov/~walker/mpi](http://www.epm.orl.gov/~walker/mpi)
- [http://www.epcc.ed.ac.uk/mpi](http://www.epcc.ed.ac.uk/mpi)