Super Quick Introduction to MPI

Makoto Nakajima

Federal Reserve Bank of Philadelphia

April 2009

Outline of Topics

1. Introduction
2. MPI Basics
3. 6 Basic Commands
4. Sample Program: Hello, World!
5. Collective Communication Commands
6. Other Useful Commands

What is a Cluster?

- A cluster is a bunch of computers connected by a network.
- Each computer could have multiple processors, or multiple cores.
- Each core is called a node.
- Clusters become popular as the price of personal computer dropped dramatically, making the price of a cluster cheaper compared with the price of a super-computer (one big super-fast computer).
- One can construct his own cluster, by connecting bunch of personal computers with ethernet cables. If you only use components which are widely available for consumers, it's called the Beowulf cluster.

World's Fastest Computer as of November 2008

- IBM Roadrunner @ Los Alamos
- Uses 129, 600 processors.

World's (Possibly) Slowest Cluster

- My Beowulf cluster
- Uses 2 processors.

Parallel Software

- As the cluster became more and more popular, softwares to utilize the power of clusters became more and more developed.
- Since a cluster consists of a bunch of small computers, in order to use the potential of the cluster, you have to divide a single program into a collection of smaller jobs so that different small jobs can be executed by each node of the cluster simultaneously.
- This is the basic idea of parallel programming.
- MPI is one of the most widely used parallel softwares.

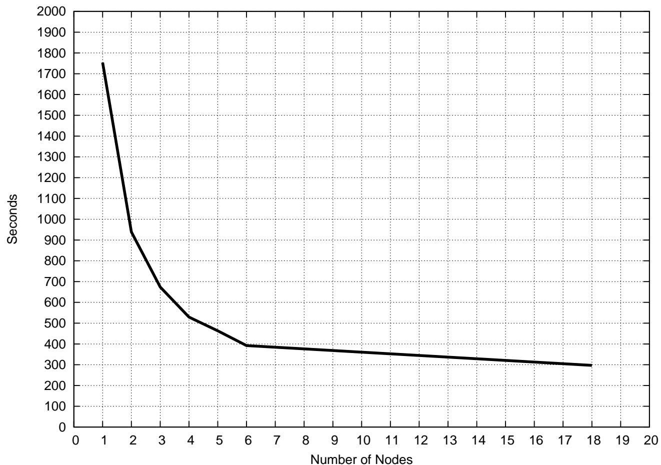
What is MPI?

- Stands for Message Passing Interface.
- Package of procedures that enables nodes of a cluster to communicate (send data each other) easily and efficiently.
- Used as an external library to various computer languages (C, Fortran, R, Python, Java, etc).
- A bit tedious to use. In the program, you have to tell explicitly what tasks are implemented by which nodes.
- Standard parallel software. Installed to almost any cluster.
- Highly portable.
- Highly scalable.

OpenMP versus MPI

- OpenMP can be used by putting special comments (called directives) to the serial code. Can use the same code for parallel and non-parallel environment (directives are ignored as pure comments in non-parallel environment).
- Syntax is pretty simple.
- Recent version of Intel Compilers has a compiler option to automatically use OpenMP without any directives!
- Highly portable.
- Not-so-scalable: Mainly for one machine (up to 8 nodes).
- The gain varies depending on algorithm.

Gain of Using MPI: A Benchmark



Cluster Basics

- Typically a cluster consists of:
  1. Frontend (Master, Mother) node: Cluster's interface to the outside world. You log-in to this node. Better not use this node for computation.
  2. Compute (Slave) nodes: Specialize in computation. When using MPI, you don't deal with them directly.
- Typical workflow to use a cluster from Windows machine:
  1. Write a source code in your Windows machine. Notice you cannot test your program in your Windows machine.
  2. Using (S)FTP software, send the source code to the frontend node.
  3. Using telnet (SSH) client, log-in to the frontend node. You do everything in the frontend.
  4. Compile the source code using a special command (MPI library is automatically linked when compiling).
  5. Run the executable using a special command.

MPI Basics

- You only need one code.
- The same code runs in all the nodes simultaneously.
- It's better to start with a code which perfectly works for a single processor (but writing the code in a way such that it's easy to change to parallel code later).
- In the code, you need to explicitly tell which node does which job. All the nodes are assigned an id (an integer which takes value from 0 to (number of nodes-1)) when MPI is used. You can assign different jobs to different nodes by referring to this id.
- Remember that distributed-memory environment is the default. You have to remember what data each node owns. If necessary, you need to tell the nodes to transfer data among them (message-passing).

MPI Basics: Example 1

Example 1

```
if (your_name==Dirk)
    drink beers
else if (your_name==Makoto)
    clean the bathroom
end if
watch TV
```

- Dirk drinks beers, and watches TV.
- Makoto cleans the bathroom, and watches TV.
- Others just watch TV.
- Notice everybody uses the same code.
- Notice I might start watching TV later than Dirk and others.

MPI Basics: Example 2

Example 2

```
if (your_name==Dirk)
    go to L2
else if (your_name==Makoto)
    wait for Dirk at his office
end if
Drink beers.
```

- Dirk goes to L2, and drinks beers.
- Makoto waits for Dirk forever.

MPI Basics: Example 3

Example 3

```
get (your id)
get (total number of nodes)
set n=id+1
do
    clean n-th floor of McNeil building
    n=n+(total number of nodes)
    if (n>(number of floors in the building)) exit
end do
```

- Suppose the total number of nodes is 3 (id=0,1,2), and there are 5 floors in the building.
- id=0 cleans 1st floor and 4th floor.
- id=1 cleans 2nd floor and 5th floor.
- id=2 cleans 3rd floor.

MPI Basics: Example 4

Example 4

```
get (your id)
get (total number of nodes)
set n=id+1
do
    check if there's anybody on the n-th floor of the building
    n=n+(total number of nodes)
    if (n>(number of floors in the building)) exit
end do
gather all information to id=0
tell whether if there's anybody in the whole building
```

- All the information obtained during the do-loop are gathered to id=0.
- Only id=0 can tell the correct result.

MPI Basics: Example 5

Example 5

```
get (your id)
get (total number of nodes)
set n=id+1
do
    check if there's anybody on the n-th floor of the building
    n=n+(total number of nodes)
    if (n>(number of floors in the building)) exit
end do
gather all information to id=0
id=0 sends the gathered information to all id > 0
tell whether if there's anybody in the whole building.
```

- Not only id=0 but also all the other nodes can tell the correct final result.

MPI Basics: Example 6

Example 6

get (your id)
get (total number of nodes)
compute x=10/(your id)
send x each other *
compute sum of x from all nodes
output the sum

- id=0 crushes when trying to compute 10/0.
- Others keep waiting at *.

Compiling and Linking MPI Code

mpif90 [name of code].f90 -o [name of executable]

- Most likely, you will implement the command using terminal.
- The command does compiling and linking with the MPI library simultaneously.
- Example: `mpif90 foo.f90 -o foo` If you implement this, then you get an executable `foo` in the same directory as the source code `foo.f90`
- Can put compiler options.

- Obviously, this is for Fortran 90.
- For C Language, `mpicc` is used.
- For Fortran 77, `mpif77` is used.

Executing MPI Code

mpirun -np [#1] -machinefile [#2] [name of executable]

- The command executes the already-compiled MPI code.
- #1: Number of nodes used. If you put a number larger than the number of nodes, some nodes are used twice (running two of the same programs separately), which is an inefficient thing to do.
- #2: The option `-machinefile [#2]` is used only when you want to specify the nodes that you want to use. #2 is the name of the file which contains the list of the names of nodes to be used. If omitted, the default list (usually contains all the nodes) is used.
- Example: `mpirun -np 8 ./foo` If you implement this, the first 8 nodes in the default list of nodes run the same executable `foo` in the current directory simultaneously.

At the Beginning of MPI code...

include 'mpif.h'

- Used to include header containing variables and procedures related to MPI Library. You have to start your program with this.

Now we start 6 fundamental subroutines of MPI. All the subroutines can be used by call, after including `'mpif.h'`.

6 Basic Commands of MPI [1]

MPI_INIT(ierror)

- Used to initialize MPI environment.
- Put it at the beginning of your code after variable declaration without thinking.
- `ierror` is an integer which returns the error code if an error occurs (usually there's no error when implementing this command).
- For C version, there is no `ierror`.

6 Basic Commands of MPI [2]

MPI_FINALIZE(ierror)

- Used to finalize MPI environment.
- Put it at the end of your code without thinking.
- Again, `ierror` is an integer and no `ierror` for C version.

6 Basic Commands of MPI [3]

MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierror)

- Used to obtain the number of nodes (nproc). Obviously, nproc must be declared as an integer.
- Usually this subroutine is called right after MPI_INIT.
- MPI_COMM_WORLD is declared in mpif.h. It is called a communicator. A communicator defines a group of nodes. MPI_COMM_WORLD is the default communicator, which contains all the nodes used. You could define different communicator, but it's an advanced stuff.
- nproc that is returned corresponds to the communicator referred. In the case above, since the default communicator is used, total number of nodes used in the program is returned.
- Again, ierror is an integer and no ierror for C version.

6 Basic Commands of MPI [4]

MPI_COMM_RANK(MPI_COMM_WORLD,id,ierror)

- Used to obtain the unique id of each node (id). Obviously, id must be declared as an integer.
- Usually this subroutine is called right after MPI_INIT.
- Notice that the returned value id is different for each node. id takes the value from 0 to nproc-1. This is crucial to make each node do different jobs.
- MPI_COMM_WORLD is again a communicator.
- Again, ierror is an integer and no ierror for C version.

6 Basic Commands of MPI [5]

MPI_SEND(buf,count,type,dest,tag,comm,ierror)

- Used to send data to the node dest.
- buf indicates the address of the data that are sent. In case sending a scalar, the scalar itself enters as buf. In case sending a 1-dimensional array, buf should be the first element of the array, like x(1).
- count is an integer indicating the length of the data sent.
- type indicates the type of data that are sent. MPI_INTEGER and MPI_DOUBLE_PRECISION are often used. There are many other.
- dest is an integer and indicates id of the destination of the data.
- tag is an integer and is used to refer to the current message passing operation. Can be any integer but should be unique.
- comm is a communicator. We use MPI_COMM_WORLD.
- ierror is an integer and returns the error code if there is one.

6 Basic Commands of MPI [6]

MPI_RECV(buf,count,type,root, tag,comm,STATUS(MPI_STATUS_SIZE),ierror)

- Used to receive data from the node root.
- buf, count, type, tag, comm, ierror are same as for MPI_SEND.
- MPI_RECV is linked with a particular MPI_SEND using tag.
- root is an integer and indicates the source of the data received. id number, which takes the value from 0 to nproc-1, is used.
- STATUS(MPI_STATUS_SIZE) is an integer array which indicates the status of the operation. The variable status must be declared. MPI_STATUS_SIZE is defined in mpif.h.

Remarks on MPI_SEND and MPI_RECV

- Both MPI_SEND and MPI_SEND commands don't end until the data are received by the destination (whether the data are received or not is automatically checked). In this sense, this type of sending and receiving operation is called blocking operation.
- As you can imagine, there is a non-blocking send operation as well. The commands are MPI_ISEND and MPI_IRECV("I" means immediate). It potentially allows the code to implement other operations while the data are sent and received. However, the receiving side is a bit tricky, as the receiving operation ends before all the data are received. Therefore, non-blocking operations are not default.

Sample Program: "Hello, World!"

hello_world.f90 Page 1

```
 1 program hello_world
 2
 3   implicit none
 4
 5   include 'mpif.h'
 6
 7   integer:: ierror, id, nproc
 8
 9   call mpi_init(ierror)
10
11   call mpi_comm_rank(mpi_comm_world, id, ierror)
12
13   call mpi_comm_size(mpi_comm_world, nproc, ierror)
14
15   print *, 'hello, world! i am node ',id
16
17   if (id==0) then
18     print *,'and I am the master!'
19   end if
20
21   call mpi_finalize(ierror)
22
23 end program hello_world
```

Introduction to Collective Communication

- MPI_SEND and MPI_RECV only support a message passing from one node to another. In this sense, these commands are called one-to-one communication commands.
- In many other occasions, we want to let one node to send data to all the other nodes, or gather data from all the nodes to one node. These operations are called `collective communications`.
- In theory, collective communication can be achieved by a combination of one-to-one communications, but using collective communications make the code simpler and maybe faster.
- MPI has a variety of collective communication commands. We will see the most useful ones below.

Collective Communication Commands [1]

MPI_BCAST(buf,count,type,root,comm,ierror)

- Broadcast data defined by [buf,count,type] from root to all the nodes in comm
- comm, ierror are same as before.

Collective Communication Commands [2]

MPI_REDUCE(sendbuf,recvbuf,count,type, op,root,comm,ierror)

- Summarize data [sendbuf,count,type] of all the nodes in comm, create [recvbuf,count,type], and store it at root.
- comm, ierror are same as before.
- sendbuf refers to the address of the data stored in each node and which are summarized.
- recvbuf refers to the address of the summarized data stored in root.
- There are various options for op. Examples are: MPI_SUM sums up the data across all the nodes. MPI_PROD multiplies all the data. MPI_MAX returns the maximum. MPI_MIM returns the minimum.
- When [sendbuf,count,type] is an array, the operation op is applied to each element of array.

Collective Communication Commands [3]

MPI_GATHER(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,root,comm,ierror)

- Combine data [sendbuf,sendcount,sendtype] of all the nodes in comm, create [recvbuf,nproc*recvcount,recvtype], and store it at root.
- comm, ierror are same as before.
- Typically sendcount=recvcount, sendtype=recvtype, and the length of the array recvbuf is nproc*recvcount.

Collective Communication Commands [4]

MPI_SCATTER(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,root,comm,ierror)

- Scatter data [sendbuf,nproc*sendcount,sendtype] held originally by root to all the nodes in comm, as [recvbuf,nproc*recvcount,recvtype].
- In a sense, the opposite of MPI_GATHER.
- comm, ierror are same as before.
- Typically sendcount=recvcount, sendtype=recvtype, and the length of the array sendbuf is nproc*sendcount.

Collective Communication Commands [5]

```
MPI_ALLREDUCE(sendbuf,recvbuf,count,type,
op,comm,ierror)
```

- MPI_REDUCE plus MPI_BCAST.
- The result of MPI_REDUCE operation is shared by all the nodes.
- Notice there is no root.

```
MPI_ALLGATHER(sendbuf,sendcount,sendtype,
recvbuf,recvcount,recvtype,comm,ierror)
```

- MPI_GATHER plus MPI_BCAST.
- The result of MPI_GATHER operation is shared by all the nodes.
- Notice there is no root.

Other Useful Commands [1]

call MPI_ABORT(comm,ierror)

- Used to kill the code running on all the nodes included in the communicator `comm`.
- The default communicator is `MPI_COMM_WORLD`.
- You only need one node to call this subroutine to abort the entire program.
- `ierror` is same as before.

call MPI_BARRIER(comm,ierror)

- All the nodes included in `comm` wait until all the nodes call this subroutine
- Therefore, used to synchronize the timing.
- Useful for debugging.
- `ierror` is same as before.

Other Useful Commands [2]

MPI_WTIME()

- This is a function.
- Returns current time measured by the time passed since some arbitrary point of time in the past.
- Only the difference between two points of time matter, because the starting point is arbitrary.
- No argument necessary.

MPI_WTICK()

- This is a function.
- Returns the number of seconds which is equivalent to one unit in MPI_WTIME
- No argument necessary.