

# Practical Introduction to MPI for Economists

Makoto Nakajima<sup>1</sup>

<sup>1</sup>University of Illinois, Urbana-Champaign

April 2006

# Outline of Topics

- 1 Introduction
- 2 Environment
- 3 Motivating Examples
- 4 MPI Basics
- 5 6 Basic Commands
- 6 Collective Communication Commands
- 7 Other Useful Commands

# Motivation

- Models with a large state space or a large number of agents require a large amount of time to solve.
- In addition, if a model is calibrated or structurally estimated, the model has to be run many times with different parameter values, to find a set of parameters which enables the model to replicate key properties of the data.
- How to make a code to run faster?

# Things You Can Do to Solve a Problem Faster

- Use a faster language: Fortran or C. But once you write an efficient code in Fortran or C, there's no more room for improvement.
- Use a faster processor. There is a limitation for the speed of a single processor.
- Connect multiple processors = Cluster.

# Remarks

- Previously, many vendors tried to make super fast computers by connecting multiple processors and configuring them to work as a single processor (vector supercomputer). But it is really expensive, not scalable, and not easy to maintain.
- Instead, currently, a major way to make a super fast computer is to connect multiple processors and use them in parallel.
- Theoretically, it is possible to construct a super fast cluster by connecting a lot of cheap PCs. (A cluster which is made only using widely commercially available components is called a Beowulf cluster.)
- A multiple core machine can be seen itself a cluster, if you treat multiple cores as different nodes. Multiple core processors have become a feasible and convenient option to construct a cluster as well. But there's a limitation in the number of cores in a processor.

# Shared Memory or Distributed Memory

Consider a cluster with two processors (nodes). There are two possible specifications.

## 2 Specifications of a Cluster of 2 Nodes

**Shred Memory** The two nodes share one memory space. A variable in two nodes always has one value. An example is a dual core processor.

**Distributed Memory** The two nodes independently have their own memory space. A variable in two nodes can have different values. An example is two computers connected by ethernet.

If there are more than two nodes, a **hybrid** is possible. An example is multiple PCs equipped with multiple core processors, connected by ethernet.

# Parallel Softwares

- There are three major software environments which are used for parallel programming; MPI (Message Passing Interface), OpenMP, and PVM (Parallel Virtual Machine).
- All are used as an additional library to a computing language.
- Usually parallel library is used with Fortran or C.
- Can be used with other languages (Java, MATLAB, R, etc.) as well but less popular.

# MPI

## MPI

- Mainly designed for **distributed-memory** environment.
- MPI Library adds commands which enable different nodes to send and receive data each other.
- Have to write exactly what each nodes do. Requires relatively **large amount of programming**.
- **Scalability**: Code doesn't depend on the number of nodes. The speed of the code usually increases as the number of nodes increases.
- **Portability**: Very well standardized. A MPI code can be used for virtually any cluster with MPI installed.

# PVM

## PVM

- A close alternative to MPI. Indeed, older than MPI, but (seems to me) less popular than MPI right now.
- Initially developed by Oak Ridge National Laboratory.

# OpenMP

## OpenMP

- Mainly used for **shared-memory** environment.
- But ways to use it in distributed-memory or hybrid environment are developed.
- OpenMP Library adds commands which tells the processors which part of the code to be parallelized.
- Requires **small modifications** from the code for a single processor.
- **Scalability**: Code doesn't depend on the number of nodes. The speed of the code doesn't always increase as the number of nodes increases.
- **Portability**: Very well standardized. An OpenMP code can be used for virtually any cluster with OpenMP installed.

# Two Major Libraries of MPI

## MPICH

- Developed by Argonne National Laboratory.
- Free.
- Can be used with Windows as well.
- Most popular (it seems to me).

## LAM

- Developed by Laboratory for Scientific Computing, University of Notre Dame.
- Free.
- Unix only.

# Things to Consider

- The gain of using parallel code is larger if you parallelize larger part of a code.
- The gain of using parallel code is larger if you transmit less data across nodes.
- The gain of using parallel code is larger if you can avoid less idle time for all the nodes.

# Motivating Example [1]: Monte Carlo Simulation

- Remember the standard Aiyagari's model.
- Suppose we obtained optimal decision rules and want to run Monte Carlo simulation with 1,000,000 agents for 1,000 periods.
- If you use, let's say 10 nodes, you can distribute 100,000 agents to each node, and run Monte Carlo simulation simultaneously and independently.
- If you want to compute aggregate capital and labor, you only need to aggregate capital and labor for each node, and sum up the numbers obtained from each node.
- Notice that you don't need to transmit all the individual data.
- If you need to apply the same operation with different data independently, this is an ideal chance to benefit from parallelization. This situation is called **data parallelism**.

## Motivating Example [2]: Computing Lots of Different Statistics

- Suppose you want to compute 100 statistics using the same data on the type distribution.
- Suppose the statistics are independent, in the sense that computation of each statistic doesn't require other statistics.
- In this case, if you have 10 nodes, you can assign 10 statistics to each node, and let each node to compute assigned 10 statistics independently.
- If the statistics are not use for future computation but you just want to print them out, then you only need to collect all the statistics to one node, and let the node to print out the gathered statistics.
- If you need to apply different operations with the same data independently, this is a also chance to benefit from parallelization. This situation is called **functional parallelism**.

## Motivating Example [3]: Value Function Iteration

- Suppose you are implementing value function iteration in the standard Aiyagari's model.
- Suppose you have 10 different realizations of shocks and 1000 grid points in the asset space.
- In order to update a value function, you basically need to solve 10,000 optimization problems.
- If you use 10 nodes, you can assign 1,000 individual states to each node and solve the optimization problem simultaneously.
- But be careful. Every time you update the value function, value function stored in each node should be updated. Therefore, in each iteration, the whole value function has to be shared by all the nodes. So the data transmission cost is potentially large.
- In general, be careful of the **data dependency**.

# MPI Basics

- You only need **one code**.
- Each processor is called a **node**. A node can be a core in a processor.
- The **same code** runs in all the nodes simultaneously.
- It's better to start with a code which perfectly works for a single processor.
- 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.

# Compiling and Linking MPI Code

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

- 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`
- Obviously, this is for Fortran 90.
- For C Language, `mpicc` is used.
- For Fortran 77, `mpif77` is used (but who dare?).

# 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` simultaneously.

# Remarks

- Usually, your home directory (the directory where you store and execute your code) is shared by all the nodes. Therefore, you only need to compile, link and execute once. If you execute, all the nodes use the same executable.
- If you read or write a file, be careful not to allow different nodes to operate the same command simultaneously. All the nodes share the same directory, and you don't want different nodes to open the same file simultaneously.
- Possibly you want to assign one of the nodes to do all the writing and reading files. Or you want to control the timing so that each node read or write the same file sequentially.

# At the Beginning of MPI code...

```
include 'mpi.h'
```

- Used to include 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(ierr)`

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

## 6 Basic Commands of MPI [2]

### `MPI_FINALIZE(ierr)`

- 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` is an integer.
- Usually this subroutine is called right after `MPI_INIT`.
- `MPI_COMM_WORLD` is defined 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 id of the node (`id`). Obviously, `id` is 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`.
- `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.
- `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 data transfer 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`.
- `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. `MPI_STATUS_SIZE` is defined in `mpif.h`.

# Remarks on MPI\_SEND and MPI\_RECV

- Both MPI\_SEND and MPI\_RECV 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 programmer 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 Codes using Basic Commands

- Sample code 1: "Hello, World!"
- Sample code 2: "Hello, World!" only from one node.
- Sample code 3: Sum from 1 to N.

# 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 communication**.
- 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. `MPI_PROD` multiplies all the data. `MPI_MAX` returns the maximum value among the data from all nodes. `MPI_MIN` returns the minimum.

# 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.

```
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.

# Sample Codes using Collective Communication Commands

- Sample code 4: Sum from 1 to N.
- Sample code 5: Computation of  $\pi$  using Monte Carlo method.
- Sample code 6: Monte Carlo simulation of the standard Aiyagari's model.

# 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(ierror)
```

- All the nodes wait until all the nodes call this subroutine
- Therefore, used to synchronize the timing.
- `ierror` is same as before.

## Other Useful Commands [2]

### MPI\_WTIME()

- 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.
- There's no arguments.

### MPI\_WTICK()

- Returns the number of seconds which is equivalent to one unit in MPI\_WTIME
- There's no arguments.