Practical Introduction to MPI

Makoto Nakajima

Federal Reserve Bank of Philadelphia

November 15, 2011

Outline of Topics

What is a Cluster?

- A cluster is a bunch of computers connected by a network.
- Each computer is called a node or a machine.
- Each node could have multiple processors, and multiple cores.
- 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.
- A typical cluster consists of:
  1. Frontend (Master, Mother) node: Cluster's interface to the outside world. You log-in to this node. Not for computation.
  2. Compute (Slave) nodes: Specialize in computation. When using MPI, you don't deal with them directly.

An Example: World's Fastest Cluster as of November 2011

- K-computer in Japan.
- 705024 cores.

An Example: World's (Possibly) Slowest Cluster

- My Beowulf cluster back in 2007.
- 2 cores.

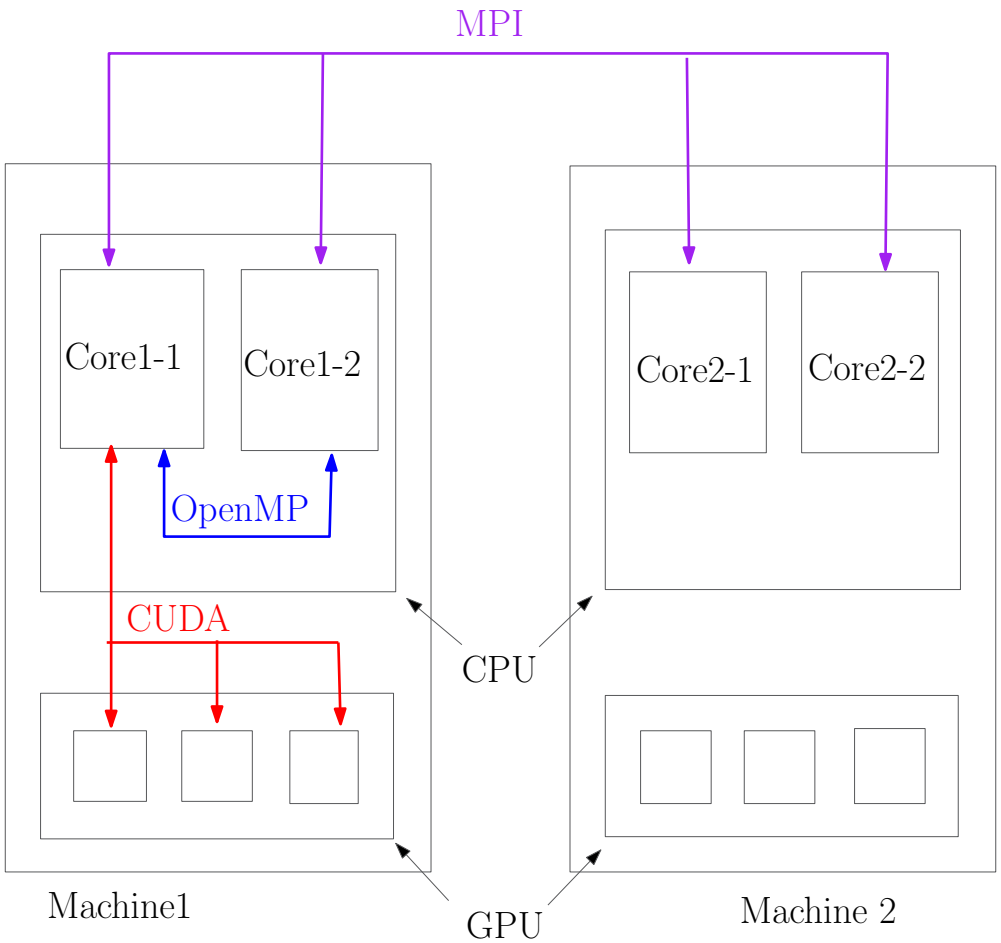
Parallel Software

- 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 core 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 cores 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 processes.
- Standard parallel software. Installed to almost any cluster.
- Highly portable.
- Highly scalable.

MPI / OpenMP / CUDA



- OpenMP is more restricted than MPI but much easier to use.
- Obvious complementarity between MPI and CUDA (if necessary).

MPI Basics

- You only need one code.
- The same code runs in all the processes 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 process does which job. All the processes are assigned an id (an integer which takes value from 0 to (number of processes-1)) when MPI is used. You can assign different jobs to different processes by referring to this id.
- Remember that distributed-memory environment is the default. You have to remember what data each process owns. If necessary, you need to tell the processe to transfer data among them (message-passing).

MPI Basics: Example 1

Example 1

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

- Iourii cleans the bathroom, and watches TV.
- Makoto drinks beers, and watches TV.
- Others just watch TV.
- Notice everybody uses the same code.

MPI Basics: Example 2

Example 2

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

- Suppose the total number of processes 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 3

Example 3

```
get (your id)
get (total number of processes)
set n=id+1
do
    check if there's anybody on the n-th floor of the building
    n=n+(total number of processes)
    if (n>(number of floors in the building)) exit
end do
if (id/=0) send what you found to id=0
if (id==0) receive information from other processes
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 4

Example 4

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

- All the processe can tell the correct final result.

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 processes started. If you put a number larger than the number of cores, some cores 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/cores that you want to use. #2 is the name of the file which contains the list of the names of nodes (and number of cores for each node) to be used. If omitted, the default list (usually contains all the nodes and all the cores) is used.
- Example: `mpirun -np 8 ./foo` If you implement this, the first 8 cores in the default list of machines 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 processes (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 processes. MPI_COMM_WORLD is the default communicator, which contains all the processe 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 processes 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 process (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 process. id takes the value from 0 to nproc-1. This is crucial to make each process 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 process `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 process 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 process to another. In this sense, these commands are called one-to-one communication commands.
- In many other occasions, we want to let one process to send data to all the other processes, or gather data from all the processes to one process. 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 processes 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 processes 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 process 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 processes. 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 processes 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 processes 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 processes.
- 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 processes.
- Notice there is no root.

Other Useful Commands [1]

call MPI_ABORT(comm,ierror)

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

call MPI_BARRIER(comm,ierror)

- All the processes included in comm wait until all the processes 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.

Example: Aiyagari (1994)

Let's do some more fun stuff:

1. Show an example of using MPI with Fortran 90.
2. Use a standard heterogeneous agent model of Aiyagari (1994) as an example.
3. Start with the serial (non-parallel) version of the code, and show how the code is parallelized.
4. Show how much the code runs faster if parallelized.

The Model

- Standard incomplete market economy with mass of atomless infinitely-lived agents.
- Individual earnings shock follows a Markov chain.
- An agent receives stochastic earnings each period, and chooses how much to save and consume.
- No labor-leisure choice.
- Can save only in the form of physical capital. Ad-hoc borrowing constraint (set at zero). No state-contingent security allowed to be traded.
- Representative firm has an access to CRS technology.
- Focus on the steady state where interest rate and wage are constant over time.

Solution Method: Discretization

- Asset space is discretized into $na(= 2000)$ grid points.
- Productivity shock can take one of the 7 values ($ne = 7$)
- Each agent is characterized by (e, a), where:
  - $e \in \{1, 2, 3, 4, 5, 6, 7\}$
  - $a \in \{1, 2, 3, ..., 2000\}$
- Therefore, total number of individual states is ni=na*ne=14000.
- Restrict the choice to be on the asset grids. Therefore, the optimization problem for an agent of each type is just choosing the grid point $a' \in \{1, 2, 3, ..., 2000\}$ associated with the highest value.

Solution Method: Main Loop

1. Beginning of Loop 0 Guess $\frac{K}{Y}$. This gives the guess for r and w (remember CRS production technology).
2. Loop 1 Given the prices, using the value function iteration, find the optimal value function $V(e, a)$ and associated optimal decision rule $a' = g_a(e, a)$.
3. Loop 2 Using $g_a(e, a)$ and the Markov transition matrix $p(e, e')$, compute the ergodic distribution of agent types.
4. Using the ergodic distribution, compute the aggregate labor supply and the aggregate capital stock.
5. Compute $\frac{K}{Y}$ associated with the computed aggregate capital stock and aggregate labor supply.
6. End of Loop 0 Compare the guess of $\frac{K}{Y}$ and newly obtained $\frac{K}{Y}$. If not close enough, update the guess of $\frac{K}{Y}$ and go back to the top.

Solution Method: Inside Loop 1

1. Beginning of Loop 1 Take r and w as given. Guess the value function $V_0(e, a)$.
2. Given the prices, and $V_0(e, a)$, update the value function using the Bellman operator. Notice that the optimization problem for each type (e, a) is just choosing the optimal grid point a' out of 2000 grid points.
3. Call the updated value function $V_1(e, a)$.
4. End of Loop 1 Compare $V_0(e, a)$ and $V_1(e, a)$. If not close enough, set $V_0 = V_1$ and go back to the top.

Solution Method: Inside Loop 2

1. Beginning of Loop 2 Take the optimal decision rule $g_a(e, a)$ as given. Guess the type distribution of agents $d_0(e, a)$.

2. Using $g_a(e, a)$ and the Markov transition matrix $p(e, e')$, update the distribution and denote the new distribution as $d_1(e, a)$.

3. End of Loop 2 Compare $d_0(e, a)$ and $d_1(e, a)$. If not close enough, set $d_0 = d_1$ and go back to the top.

Parallelization: Basic Idea

- Suppose we use $nproc = 10$ processes: $id = 0, 1, ..., 9$
- Remember there are $ni = na * ne = 14000$ types.
- We assign $1400 (= 14000/10)$ states for each process.
- Each process updates value only for 1400 types.
- Each process updates distribution only for 1400 types.

Parallelization: Preparation

- Function *iafun*: converts i to a
- Function *iefun*: converts i to e
- Variable $itop = ni/nproc * id + 1$
- Variable $iend = ni/nproc * (id + 1)$
- Each process is in charge of $i = itop, itop + 1, ..., iend$
- Confirm that the entire type space is covered.
- If ni is not a multiple of $nproc$, a bit tricky but the idea is the same.

Parallelization: Pseudo Code [1]

Initialization Block

1. (Start of the code)
2. include 'mpif.h' ($\rightarrow$ link mpi library)
3. (Variable declaration)
4. call MPI_INIT ($\rightarrow$ initialization)
5. call MPI_COMM_RANK ($\rightarrow$ id assigned)
6. call MPI_COMM_SIZE ($\rightarrow$ nproc set)
7. if (id==0) open 'output file'
8. (set iafun, iefun, itop, iend)

- Be careful not to allow multiple processes to access to the same file simultaneously.

Parallelization: Pseudo Code [2]

Loop 0 (main loop for prices)

1. All processes compute k_0 (initial guess for $\frac{K}{Y}$)
2. Beginning of loop 0
3. (Loop 1: obtain optimal value function)
4. (Loop 2: obtain ergodic distribution)
5. All processes compute k_1 (updated $\frac{K}{Y}$)
6. If k_0 and k_1 are close, get out of the loop
7. Update k_0
8. End of loop 0
9. call MPI_FINALIZE

- Make sure that all processes have the same prices at any point of time!

Parallelization: Pseudo Code [3]

Loop 1 (value function iteration)

1. All processes set the initial guess V_0
2. Beginning of loop 1
3. Each process computes updated value V_1 for $i = itop, ..., iend$
4. call MPI_ALLGATHER ($\rightarrow$ sharing updated value V_1)
5. If V_0 and V_1 are close, get out of the loop
6. Update $V_0 = V_1$
7. End of loop 1

- Make sure that V_0 is shared among all processes all the time!
- Notice that each process *id* has optimal decision rule only for $i = itop, ..., iend$.

Parallelization: Pseudo Code [4]

Loop 2 (Ergodic Distribution)

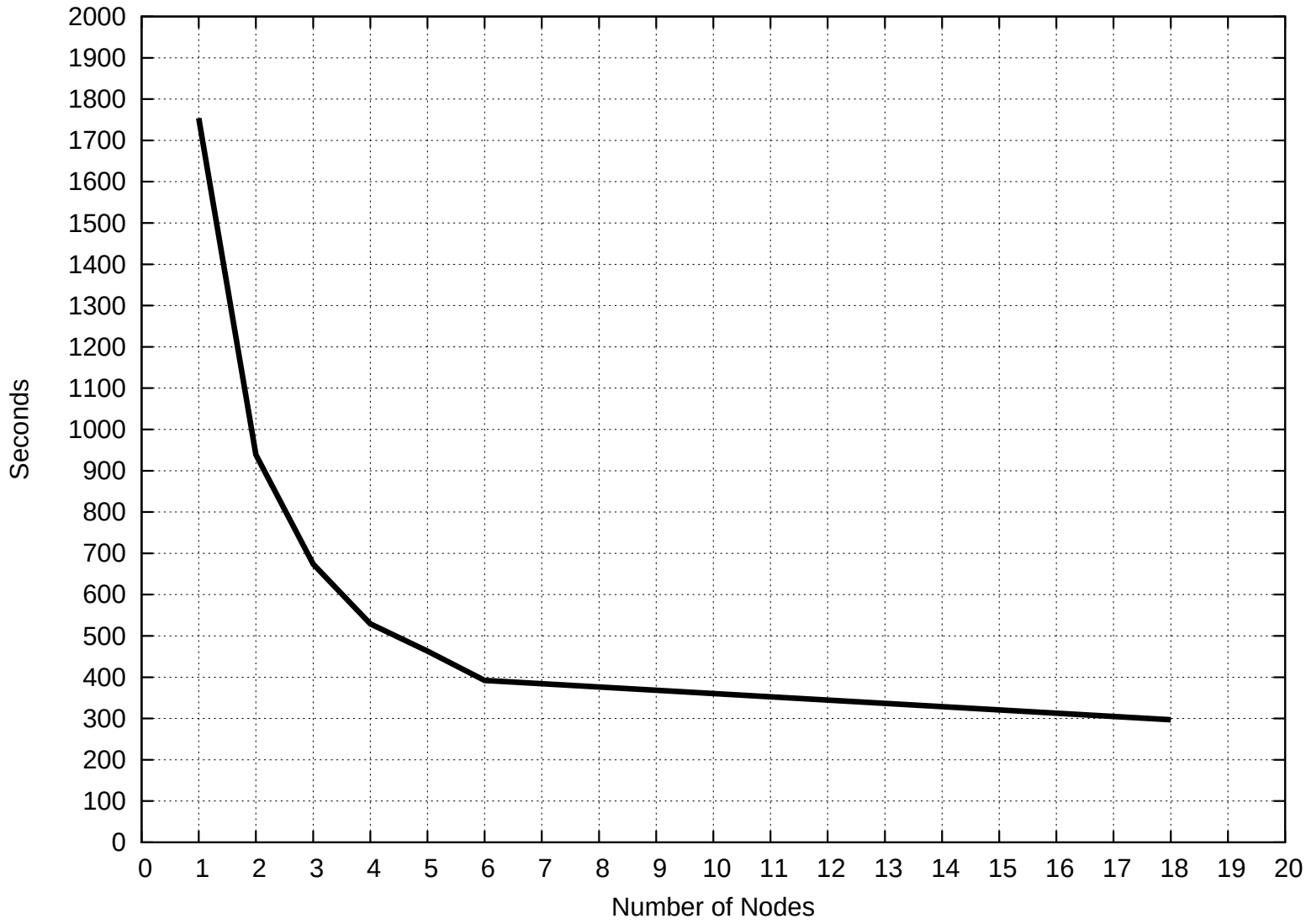
1. All processes set the initial guess d_0
2. Beginning of loop 2
3. Each process updates the distribution and obtain d_1 only for agents of type $i = itop, ..., iend$
4. call MPI_ALLREDUCE ($\rightarrow$ sum up d_1 across all processes)
5. If d_0 and d_1 are close, get out of the loop
6. Update $d_0 = d_1$
7. End of loop 2

- Make sure that V_0 is shared among all processes all the time!
- Notice that each process *id* has optimal decision rule only for $i = itop, ..., iend$.

Benchmark: Overview

- Use my `susquehanna` cluster for assessing performance of the parallel code.
- `susquehanna` consists of 7 nodes (1 frontend + 6 compute nodes).
- Each node is equipped with Athlon2600 (1.8Ghz)... Machine from the 20th century.
- Gigabit ethernet network.
- Compiled with Intel Compiler For Linux Version 7. only $-O3$ option is used.
- First of all, I ran the sequential code using (of course) one node. Then ran the parallel code with 2-6 nodes.
- I also ran the parallel code using In-Koo Cho's 18 nodes cluster (9 dual-Opteron) at UIUC.

Benchmark: Result: Figure



Benchmark: Result: Table

Number of Nodes	Time (Minutes:Seconds)	(Seconds)	Normalized
1 (Sequential code)	29:14	1754	1.00
2 (Susquehanna)	15:39	939	0.54
3 (Susquehanna)	11:14	674	0.38
4 (Susquehanna)	8:49	529	0.30
5 (Susquehanna)	7:43	463	0.26
6 (Susquehanna)	6:32	392	0.22
18 (Cho)	4:57	297	0.17

- Pretty large gain from parallelization.
- The marginal gain is diminishing, as the time spent for message passing gets larger and larger.