Using the Rmpi package on the MFCF `biglinux` machines (and a list of useful unix shell commands).

Marco Y. S. Shum

University of Waterloo

October 2015

Walkthrough

- Basics of `MPI` technology and `Rmpi` library in `R`.
- Accessing your files on `biglinux`.
- An (embarrassingly) simple example: Monte Carlo integration.
- Ideas for modelling applications.
- Appendix: useful commands on Unix shell.

Basics of MPI technology and Rmpi library

- MPI: Message Passing Interface.

Basics of MPI technology and Rmpi library

- MPI: Message Passing Interface.
- Performing multiple tasks in parallel.

Basics of MPI technology and Rmpi library

- MPI: Message Passing Interface.
- Performing multiple tasks in parallel.
- Computers participating in parallel processing (called slaves) use MPI to communicate with each other. A special slave called master serves as leader. Your source code is usually run on the master node.

Basics of MPI technology and Rmpi library

- MPI: Message Passing Interface.
- Performing multiple tasks in parallel.
- Computers participating in parallel processing (called slaves) use MPI to communicate with each other. A special slave called master serves as leader. Your source code is usually run on the master node.
  - Master will pass instructions to slaves.
  - A piece of data or command passed to all slaves (including Master) is called a broadcast.
  - Tell slaves to run their procedures.
  - Master will gather/reduce the results from working slaves once they finish.

Basics of MPI technology and Rmpi library

- MPI: Message Passing Interface.
- Performing multiple tasks in parallel.
- Computers participating in parallel processing (called slaves) use MPI to communicate with each other. A special slave called master serves as leader. Your source code is usually run on the master node.
  - Master will pass instructions to slaves.
  - A piece of data or command passed to all slaves (including Master) is called a broadcast.
  - Tell slaves to run their procedures.
  - Master will gather/reduce the results from working slaves once they finish.
- Rmpi is a library of commands which simplify the overall message passing mechanism. It also has some parallel versions of vectorisation (such as an MPI version of apply).

Accessing your files on biglinux.

When you first log in, you will likely be in the directory, /u1/my_user_name. Your usual MacProfile directory on the Mac Mini is /u1/my_user_name/MacProfile, so you can also place your files in your MacProfile directory to upload as well.

If you do not have access to your MacMini, to upload your files (e.g. R scripts and data sets), type **at your own terminal, not biglinux:**

```
> scp /path/to/your/file/on/local/machine/file.ext
my_user_name@biglinux.math.uwaterloo.ca:/destination/
```

Similarly, to pull files from the biglinux machine,

```
> scp
  my\_user\_name@biglinux.math.uwaterloo.ca:
     /path/to/your/file/on/biglinux/file.ext
  /destination/on/local/
```

An (embarrassingly) simple example: Monte Carlo integration

Consider the Monte Carlo integration of $I = \int_{x_{\min}}^{x_{\max}} f(u)du$.

$$\widehat{I}_{\mathrm{MC}} = \frac{1}{n}\sum_{i=1}^{n} f(U_i),$$

where $U_i \overset{\text{i.i.d.}}{\sim} \mathrm{Unif}(x_{\min}, x_{\max})$. The algorithm is,

1. Sample $\{u_1, \ldots, u_n\}$ from $\mathrm{Unif}(x_{\min}, x_{\max})$.
2. Compute the mean.

An (embarrassingly) simple example: Monte Carlo integration

Consider the Monte Carlo integration of $I = \int_{x_{\min}}^{x_{\max}} f(u)du$.

$$\widehat{I}_{\mathrm{MC}} = \frac{1}{n}\sum_{i=1}^{n} f(U_i),$$

where $U_i \overset{\text{i.i.d.}}{\sim} \mathrm{Unif}(x_{\min}, x_{\max})$. The algorithm is,

1. Sample $\{u_1, \ldots, u_n\}$ from $\mathrm{Unif}(x_{\min}, x_{\max})$.
2. Compute the mean.

In `R`, this is easily done for, say, $f : [0, 1] \to \mathbb{R} : x \longmapsto e^x$ and `N <- 1E8`,

```
> sum(sapply(runif(N, 0.0, 1.0), function(x)
return(exp(x)); )) / N
```

An (embarrassingly) simple example: Monte Carlo integration

Consider the Monte Carlo integration of $I = \int_{x_{\min}}^{x_{\max}} f(u)du$.

$$\widehat{I}_{\mathrm{MC}} = \frac{1}{n}\sum_{i=1}^{n} f(U_i),$$

where $U_i \overset{\text{i.i.d.}}{\sim} \mathrm{Unif}(x_{\min}, x_{\max})$. The algorithm is,

1. Sample $\{u_1, \ldots, u_n\}$ from $\mathrm{Unif}(x_{\min}, x_{\max})$. (Slave work to run parallel!)
2. Compute the mean. (Master gathers from slaves and aggregate).

In `R`, this is easily done for, say, $f : [0, 1] \to \mathbb{R} : x \longmapsto e^x$ and `N <- 1E8`,

```
> sum(sapply(runif(N, 0.0, 1.0), function(x)
return(exp(x)); )) / N
```

So, we can speed it up: the `sapply` can be split to work amongst many slaves.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in R to do so?

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.
- Broadcast `f`, `x_min`, `x_max` and `n` to the slaves. Also pass `n_slaves` to slaves, since they do not know this.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.
- Broadcast `f`, `x_min`, `x_max` and `n` to the slaves. Also pass `n_slaves` to slaves, since they do not know this.
- Broadcast instruction to slaves to tell them to each,
  - Generate `p <- n / n_slaves` samples, say `u <- c(u_1,...,u_p)`.
  - `sapply` the method `f` upon `u`.
  - Compute the sum `s <- sum(sapply(u, f))`.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.
- Broadcast `f`, `x_min`, `x_max` and `n` to the slaves. Also pass `n_slaves` to slaves, since they do not know this.
- Broadcast instruction to slaves to tell them to each,
  - Generate `p <- n / n_slaves` samples, say `u <- c(u_1,...,u_p)`.
  - `sapply` the method `f` upon `u`.
  - Compute the sum `s <- sum(sapply(u, f))`.
- Master will gather/reduce the sums and divide by `n` to obtain estimate.

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves.
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.
- Broadcast `f`, `x_min`, `x_max` and `n` to the slaves. Also pass `n_slaves` to slaves, since they do not know this.
- Broadcast instruction to slaves to tell them to each,
  - Generate `p <- n / n_slaves` samples, say `u <- c(u_1,...,u_p)`.
  - `sapply` the method `f` upon `u`.
  - Compute the sum `s <- sum(sapply(u, f))`.
- Master will gather/reduce the sums and divide by `n` to obtain estimate.
- Clean up. (VERY IMPORTANT!)

An (embarrassingly) simple example: Monte Carlo integration

What are the steps in `R` to do so?

- Start `MPI` and create `n_slaves` slaves. (mpi.spawn.Rslaves)
- Create in Master the function f, doubles $x_{\min}$ and $x_{\max}$, and the integer n, called `f`, `x_min`, `x_max` and `n`, say.
- Broadcast `f`, `x_min`, `x_max` and `n` to the slaves. Also pass `n_slaves` to slaves, since they do not know this. (mpi.bcast.Robj2slave)
- Broadcast instruction to slaves to tell them to each,
  - Generate `p <- n / n_slaves` samples, say `u <- c(u_1,...,u_p)`.
  - `sapply` the method `f` upon `u`.
  - Compute the sum `s <- sum(sapply(u, f))`.

  (mpi.remote.exec)
- Master will gather/reduce the sums and divide by `n` to obtain estimate. (mpi.reduce)
- Clean up. (VERY IMPORTANT!) (mpi.finalize)

An (embarrassingly) simple example: Monte Carlo integration

```
library(Rmpi)

# Start MPI.
print("Starting MPI.")
n_slaves <- 20
mpi.spawn.Rslaves(nslaves = n_slaves)


# Function to be integrated.
integrand <- function(x)  return(exp(x));
x_min <- 0.0;
x_max <- 1.0;

# MC parameters.
n <- 1E8
n_per_node <- n / n_slaves  # Master does not do work here.

# Send slaves information.
print("Sending slaves information....")
mpi.bcast.Robj2slave(integrand)
mpi.bcast.Robj2slave(n_per_node)
mpi.bcast.Robj2slave(x_min)
mpi.bcast.Robj2slave(x_max)

# Tell slaves to run Monte Carlo
print("Telling slaves to run simulation.")
mpi.remote.exec(
  slave_sum <- sum(sapply(runif(n_per_node, x_min, x_max), integrand))
)

# Slaves pass back to master.
print("Gathering results using reduce.")
slave_sum <- 0.0
mpi.remote.exec(mpi.reduce(slave_sum, 2, "sum", 0, 1))
mc_estimate <- mpi.reduce(slave_sum, 2, "sum")

# Compute final MC estimate
print("Computing final MC estimate.")
mc_estimate <- mc_estimate / n

mpi.finalize()
```

An (embarrassingly) simple example: Monte Carlo integration

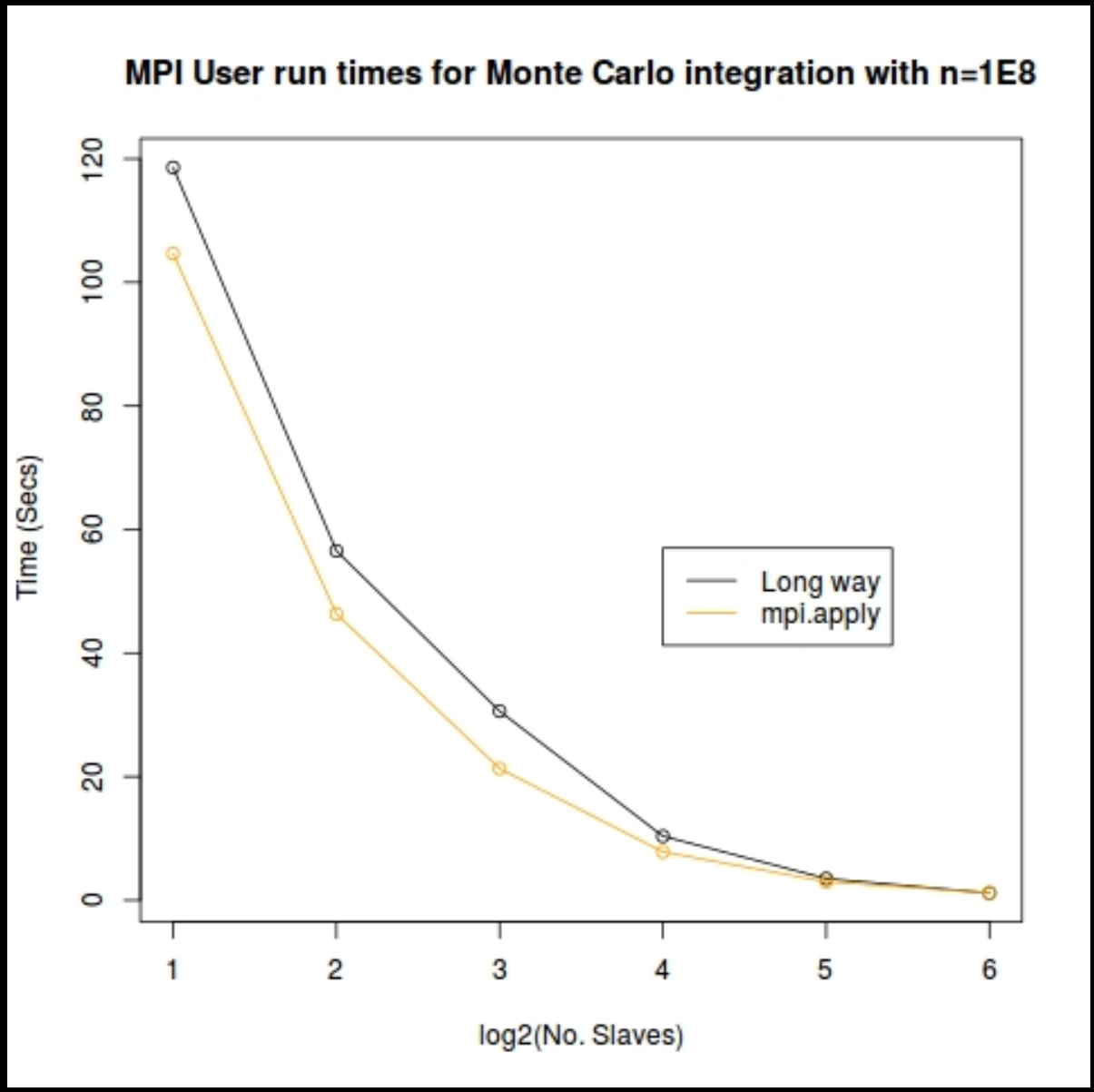
Alternatively, you can use the MPI version of apply (mpi.apply):

```
# The function to be applied
fApply <- function(n_per_node, x_min, x_max)

    # This is what each core does in the apply.
    return(sum(
          sapply(
                runif(n_per_node, x_min, x_max),
                integrand
               )
          ))

# Note that mpi.apply returns a list...
sum(
   unlist(mpi.apply(
         rep(n_per_node, n_slaves),
         fApply,
         x_min=x_min, x_max=x_max))
    ) / n
```

An (embarrassingly) simple example: Monte Carlo integration



Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- `for` loop $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- ~~`for loop`~~ $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- ~~`for loop`~~ $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Not so embarrassingly parallel: when objects are not homogeneous amongst slaves, or requires synchronous updating

Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- ~~`for loop`~~ $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Not so embarrassingly parallel: when objects are not homogeneous amongst slaves, or requires synchronous updating

- Parallel MCMC

Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- ~~`for loop`~~ $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Not so embarrassingly parallel: when objects are not homogeneous amongst slaves, or requires synchronous updating

- Parallel MCMC
- Parallel belief propagation

Ideas for modelling applications

Embarrassingly parallel:

- Expectation calculations using i.i.d. samples
- Bootstrap/Bagging
- Independent instances of algorithms with independent starting points (optimisation. MCMC)
- ~~`for loop`~~ $\longrightarrow$ `*apply`, `map`, `reduce` $\longrightarrow$ parallelise!

Not so embarrassingly parallel: when objects are not homogeneous amongst slaves, or requires synchronous updating

- Parallel MCMC
- Parallel belief propagation
- Swarm optimisation

Appendix: useful commands on Unix shells.

- `man command`: shows the manual (called "manpage") of the `command`.
- `ls directory`: lists files and subdirectories of `directory`.
- `echo $PWD`: shows which directory you are in currently.
- `mv path/to/file path/to/destination/`: moves (cut and paste, in Windows terms) `file` at `path/to/` to `path/to/destination/`.
- `cp path/to/file path/to/destination/`: copy and paste, instead of cut.
- `find directory -name "*search_string*"`: finds the files containing `search_string` in `directory`.
- `rm -r /path/to/file_or_folder`: removes either file or folder and its content.
- `mkdir path/to/folder`: creates `folder` in `path/to/`.
- `cat path/to/file`: prints `file` in terminal.
- `nano path/to/file`: opens `file` in the text editor `nano`.
- `ps auwx -fu -|grep user_name`: lists the processes running for `user_name` the currently logged on user (usually you...)
- `kill pid`: aborts and cleans up the process with process id `pid`.
- `<Ctrl-c>`: aborts and cleans up running process, and returns to prompt (frees up terminal for you to type again).
- `exit`: logs out of `ssh` (if jobs are still running, use `ps` to identify them, and `kill` to kill them first).

Thank you!