Parallel R Practical

M. T. Morgan[*]
Fred Hutchinson Cancer Research Center
Seattle, WA

4 August, 2006

Part 1: getting going

We have two objectives in the first part of this practical: to orient you on using our 'cluster', and to expose you to some of the basic functionality of the Rmpi package.

Connecting to our 'cluster'

We've set up a small 'cluster' of computers to use as a parallel environment. The cluster is very minimalist in some senses (just three nodes), and the linux environment might be unfamiliar to some of you. We'll try to walk through the very basics here.

If you're familiar with `ssh`, then feel free to connect immediately to the ip address available at the front of the room. Otherwise, follow instructions available at the start of the lab. All of us will use the same username `knoppix` and password `bioc`.

Once connected

Congratulations! You are now at a linux command line. You've all logged in as the same user. To avoid stepping on each other's toes too much, please change to your assigned directory immediately. Do this by typing

```
[% user0 %] cd ~/user0
```

where `user0` is the user id assigned to you at the start of the practical. This will be the directory to use for storing any files you might create, and will be the home directory for your R session. An (initially identical) directory exists on each of the other nodes in our cluster.

Here are some linux commands that you might find useful:

[*]mtmorgan@fhcrc.org

ls list contents of directory; `ls` list current directory contents, `ls mydir` list contents of directory `mydir` located in current directory, `ls ~/mydir` list contents of directory `mydir` in the 'home' (`~`) directory.

cd change directory. `cd ..` – change up a level, `cd mydir` to change to directory `mydir` in the current directory; `cd` to change to 'home' directory.

pwd list the path to the current directory.

R launch R, with the 'working directory' set to the current directory.

logout logout of the session.

You are actually in a fairly full-featured linux environment, so if you're familiar with this feel free to launch the editor of your choice, etc. Since we're all sharing the same computer, please don't be too resource greedy, and please do not modify files such as `.bashrc` that will influence other users.

As a simple test that things are going well, try launching the 'command line' version of R. Do this by typing

```
[% user0 %] R
```

at the prompt. You should be greeted by the familiar welcome information screen, and be presented by the > prompt. Try typing some of your favorite R commands, and loading a few packages. Everything should work as expected.

```
> x <- 10:1
> y <- 1:10
> x + y

 [1] 11 11 11 11 11 11 11 11 11 11

> hist(runif(1000))
> sessionInfo()

Version 2.3.1 Patched (2006-07-25 r38701)
x86_64-unknown-linux-gnu

attached base packages:
[1] "methods"   "stats"     "graphics"  "grDevices" "utils"
[6] "datasets"  "base"

> library(Biobase)

Loading required package: tools

> search()

 [1] ".GlobalEnv"        "package:Biobase"   "package:tools"
 [4] "package:methods"   "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "Autoloads"         "package:base"
```

If there are problems, let's try to work them out now before carrying on.

A couple of quick things about R in command-line mode. You might find it useful to use the keys `ctrl-p` and `ctrl-n` to recall commands that you've already entered. Some commands make use of other programs that can be confusing, and in particular the command

```
> f <- function(x) x
> fix(f)
```

will open the `vi` editor to edit the function `f`. Unless you're familiar with `vi`, probably the best thing to do is to exit immediately by typing `:q` followed by the return key. We'll try to stick with simple input and output, to avoid having to make complex editing changes. Another confusing command can result from searching for help

```
> library(help = Rmpi)
> ?mpi.bcast.cmd
> help.search("bcast")
```

Each of these commands opens the `less` 'pager'. You can scroll through the resulting display by typing `f` and `b` to move forward and backward a page at a time, and type `q` to stop displaying the help information.

Launching and exploring Rmpi

The objective of this section is to familiarize you with the basic functionality of the Rmpi package. We won't really do any useful work, but hopefully spur some thoughts about how to use parallel programming.

Start by loading the Rmpi package, starting and stopping a collection of nodes, and executing a few simple commands:

```
> library(Rmpi)
> mpi.spawn.Rslaves()

        1 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 2 is running on: gladstone
slave1 (rank 1, comm 1) of size 2 is running on: gladstone

> mpi.remote.exec(search())

$slave1
 [1] ".GlobalEnv"        "package:Rmpi"      "package:methods"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "Autoloads"
[10] "package:base"

> x <- 1:5
> mpi.bcast.Robj2slave(x)
> rm(x)
> mpi.remote.exec(x^2)
```

```
  out
1   1
2   4
3   9
4  16
5  25
```

```
> mpi.remote.exec(x <- x^2)
```

```
  out
1   1
2   4
3   9
4  16
5  25
```

```
> mpi.remote.exec(x <- x * mpi.comm.rank())
```

```
  out
1   1
2   2
3   3
4   4
5   5
```

```
> mpi.close.Rslaves()
```

```
[1] 1
```

Now let's get a more comprehensive feeling for the functionality of the package. Look at the overview of functions provided by

```
> library(help = Rmpi)
```

Remember that you can scroll forward and backward with the commands `f`, `b`, and that you can stop reading with `q`.

There are three groups of functions. We'll find those toward the end, in the section 'MPI Extensions specifically to slavedaemon.R' most useful (slavedaemon.R refers to a script in the Rmpi package used to launch nodes when we use the command `mpi.spawn.Rslaves`). Explore the help pages for specific functions that look particularly relevant, e.g.,

```
> ?mpi.remote.exec
> ?mpi.parLapply
> ?mpi.setup.rngstream
> ?mpi.close.Rslaves
```

Think about how these functions might be helpful in parallel analysis.

The functions in the section 'MPI Extensions in R Environment' are useful in conjunction with the `mpi.spawn.Rslaves` function, as well as in more general parallel programming contexts. Scan the help pages for a few of these

```
> ?mpi.spawn.Rslaves
> ?mpi.bcast.Robj
```

How would you launch a cluster with, say, 6 slaves? Any ideas about when would you want a cluster with more or fewer slaves than there are compute nodes in the cluster?

The functions in the section 'MPI APIs' are the core of Rmpi. If you have used MPI in other programming languages, then you'll recognize that Rmpi has an interface to some but not all of the functionality of MPI. Rmpi provides the common point-to-point (e.g., `send` and `recv` data between two nodes), collective (e.g., `bcast`, `scatter` and `gather` data efficiently to collections of nodes), and non-blocking operations. We will not emphasize these functions, though they are actually used in many of the functions we have already executed.

A first example: mad about Golub

Many Bioconductor packages summarize gene expression data in *ExpressionSet* objects. Type the following

```
> library(golubEsets)
> data(golubMerge)
> golubMerge

Expression Set (exprSet) with
        7129 genes
        72 samples
                 phenoData object with 11 variables and 72 cases
         varLabels
                Samples: Sample index
                ALL.AML: Factor, indicating ALL or AML
                BM.PB: Factor, sample from marrow or peripheral blood
                T.B.cell: Factor, T cell or B cell leuk.
                FAB: Factor, FAB classification
                Date: Date sample obtained
                Gender: Factor, gender of patient
                pctBlasts: pct of cells that are blasts
                Treatment: response to treatment
                PS: Prediction strength
                Source: Source of sample
```

and view the help page (`?golubMerge`) about this data set to get a feel for what it represents.

The R function `mad` calculates the *median average deviation* of a sample. The `mad` is a non-parametric statistic like the standard deviation, describing variability in the sample. As an exploratory step, we might be interested in identifying genes with large variability, thinking perhaps that these are going to be most useful in distinguishing between samples (this is naive!). Here is how we might extract the matrix of expression values from `golubMerge`, calculate the `mad` for each gene, and view the top 10 most variable genes in `golubMerge`:

```
> exprSet <- exprs(golubMerge)
> res <- apply(exprSet, 1, mad)
> sort(res, decreasing = TRUE)[1:10]

       M25079_s_at HG1428-HT1428_s_at          D86974_at 
         14824.517           9648.020           9335.191 
       D49824_s_at         hum_alu_at     Z84721_cds2_at 
          8434.511           8326.282           8065.344 
       X57351_s_at        Z19554_s_at          L06499_at 
          7864.452           7701.366           7590.912 
  HG2887-HT3031_at 
          7032.713 
```

See the help pages (`?exprs`, `?apply`, etc.) for information on R functions you do not recognize, or to learn more about arguments provided to functions.

We'll now walk through a couple of different ways in which we could use a parallel environment to perform these calculations. The first method is to replace `apply` with `mpi.parApply`. To do this, evaluate commands like the following:

```
> library(Rmpi)
> mpi.spawn.Rslaves(nslaves = 3)

        3 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 4 is running on: gladstone 
slave1 (rank 1, comm 1) of size 4 is running on: gladstone 
slave2 (rank 2, comm 1) of size 4 is running on: gladstone 
slave3 (rank 3, comm 1) of size 4 is running on: gladstone 

> res <- mpi.parApply(exprSet, 1, mad)
> sort(res, decreasing = TRUE)[1:10]

       M25079_s_at HG1428-HT1428_s_at          D86974_at 
         14824.517           9648.020           9335.191 
       D49824_s_at         hum_alu_at     Z84721_cds2_at 
          8434.511           8326.282           8065.344 
       X57351_s_at        Z19554_s_at          L06499_at 
          7864.452           7701.366           7590.912 
  HG2887-HT3031_at 
          7032.713 

> mpi.close.Rslaves()

[1] 1
```

(It is not necessary to use `mpi.spawn.Rslaves` and `mpi.close.Rslaves` for every calculation, just once for each R session.)

- Hopefully the results of the two different ways of calculating `res` are the same. Use `identical` to confirm this. Are there situations when results might not be exactly identical, but 'close enough'?

Use `system.time` to get a sense for how long calculations are taking:

```
> mpi.spawn.Rslaves(nslaves = 3)

        3 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 4 is running on: gladstone
slave1 (rank 1, comm 1) of size 4 is running on: gladstone
slave2 (rank 2, comm 1) of size 4 is running on: gladstone
slave3 (rank 3, comm 1) of size 4 is running on: gladstone

> system.time(res1 <- apply(exprSet, 1, mad))

[1] 1.532 0.000 1.531 0.000 0.000

> system.time(res2 <- mpi.parApply(exprSet, 1, mad))

[1] 0.544 0.248 1.738 0.000 0.000

> identical(res1, res2)

[1] TRUE
```

This is very approximate, especially with the setup in the lab!

- How does the system time for parallel evaluation compare with that for evaluation on a single node (pay particular attention to the first and third numbers reported by `system.time`, and use the help page for `system.time` to help you interpret the results)?
- Does `system.time` offer any evidence that calculations are actually occurring on remote nodes?

In my trials, this command took only slightly longer or was even faster on one node as on several nodes.

- What sorts of factors might influence parallel evaluation time?
- If evaluating this type of command really is about as time-efficient on one as on several nodes, does this mean that parallelization is pointless? Would it be better to dissect the calculation into smaller pieces (e.g., figuring out where `mad` spends most of its time) and try to parallelize those parts, or to look for ways of combining the `mad` calculation with additional steps of analysis? How would you go about each of these steps (dissection, or combining analyses)?

Let's explore a second way of performing our calculation in parallel. The previous example sends portions of the `exprSet` 'over the wire' to each node. But actually, each node has access to a copy of `golubMerge` on its own disk. So let's just send the *commands* over the wire, and see what happens. The single node code we will try to emulate uses `sapply`:

```
> ff <- function(i) mad(exprSet[i, ])
> res1 <- sapply(1:nrow(exprSet), ff)
```

To execute in parallel, we have to load the required data:

```
> mpi.bcast.cmd(library(golubEsets))
> mpi.bcast.cmd(data(golubMerge))
> mpi.bcast.cmd(exprSet <- exprs(golubMerge))
```

We can then do our parallel evaluation sending only our function `ff` to the remote nodes:

```
> system.time(res1 <- sapply(1:nrow(exprSet), ff))

[1] 1.536 0.004 1.541 0.000 0.000

> system.time(res2 <- mpi.parSapply(1:nrow(exprSet), ff))

[1] 0.036 0.004 0.527 0.000 0.000

> identical(res1, res2)

[1] TRUE
```

On a real cluster, this calculation seems to scale very well, almost inversely proportional to the number of nodes performing calculations.

- Is it 'fair' to only include the time spent in `mpi.parSapply`? Or should we also include the time for sending the additional command? Or even distributing the `golubEsets` library to the nodes in the first place?
- Suppose the calculation really does scale inversely to the number of nodes. Sketch a graph, the x-axis indicating number of nodes in a cluster and the y axis time. Sketch a curve reflecting the total 'computation' time as a function of node number. How much do I gain by adding a constant number of nodes, say 1, to a small cluster, compared to a large cluster? Sketch a second curve reflecting communication time. If there were no network issues, communication time might increase close to logarithmically with number of nodes. Combine computation plus communication time into a single, overall computation, time. What does this say about limits to parallelization?

Reproducibility

As a final introductory exploration of Rmpi, and as an introduction to a thorny issue in parallel programming, let's explore generating random numbers. Here are some things to try:

- Start a cluster of slaves.
- Use `runif` and `mpi.remote.exec` to generate 15 random numbers on each node. Why does each node generate supposedly random numbers that are all in the same sequence?
- Figure out how to use `mpi.setup.rngstream` to generate streams of random numbers that are different on each node.
- Figure out how to use `mpi.setup.rngstream` so that each node produces a different stream of random numbers (different on each node – call the streams 'A'), but that two separate invocations of the same random number call again produces the streams 'A'.
- Now arrange to start the cluster, generate a sequence of random numbers (stream 'A'), stop the cluster, and start the cluster again (with the same number of nodes) and generate the sequence of random numbers 'A' again.
- Try stopping your cluster, and staring with a *different* number of nodes (remember that each node starts up a new instance of R, and that we only have three actual nodes available, so don't be too ambitious about your cluster size). Can you generate the sequence of random numbers 'A'? If you wanted to be able to repeat exactly the same sequence of random numbers regardless of cluster size, what might you do?

Part 2: cross-validation, bootstrap, and 'SPMD' processing

This portion of the lab tackles some useful work, in the form of an analysis (`xval`) that can require substantial computational time. We also explore a model of parallel computation in R, where each node executes the same programming, performing some redundant calculation before or after a parallel section where work is divided between nodes.

Cross-validation

The serial version of the analysis we want to do is

```
> library(MLInterfaces)
> library(golubEsets)
> data(golubMerge)
> smallG <- golubMerge[200:250, ]
> lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
+     group = as.integer(0))
> table(lk1, smallG$ALL.AML)

lk1   ALL AML
  ALL  37  10
  AML  10  15
```

This takes a subset of the `golubMerge` data set, devises an algorithm to classify gene expresssion profiles into tumor types (acute lymphoblastic leukemia, ALL, or acute myeloid leukemia, AML) using the *k*-nearest neighbor machine learning algorithm and leave-one-out cross-validation. The table summarizes how the cross-validations performed: diagonal elements represent correct classifications.

If the next paragraph becomes too complicated, use the command

```
> source("~/rsrcs/xval-setup.R")
```

to read the required set-up into your R session.

The `xval` provides provides a 'hook' that allows users to reach into the function and parallelize the computationally intensive section of code. We'll access the hook by creating a class that serves as a 'predicate' to influence code execution, and then create an object of that class:

```
> setClass("RmpiXval", representation("list"))

[1] "RmpiXval"

> cluster <- new("RmpiXval")
```

The documentation for `xval` suggests that we can define a method `xvalLoop` that will be executed in place of the `lapply` loop that performs the computationally important and parallelizeable parts of the calculation. The following method makes it possible to use `mpi.parLapply` instead of `lapply`.

```
> setMethod("xvalLoop", signature(cluster = "RmpiXval"),
+     function(cluster, ...) mpi.parLapply)

[1] "xvalLoop"
```

With these preliminaries, we're ready for our first attempt at parallelizing `xval`:

```
> mpi.bcast.cmd(library(MLInterfaces))
> lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
+     group = as.integer(0), cluster = cluster)
> table(lk1, smallG$ALL.AML)

lk1   ALL AML
  ALL  37  10
  AML  10  15
```

- Hopefully the results of the single-processor and parallel results agree. Do they?
- We discussed `mpi.parLapply` above. When it is used in `xval`, the effect is to forward `exprs(smallG)` to each of the nodes in the cluster. How would you expect this solution to scale? Any ideas about how to make this more efficient?

'Single program, multiple data' model

Before starting this section, execute the command

```
> source("~/rsrsc/batch.R")
```

The program flow in the previous example is that the manager starts to evaluate `xval`. The manager 'massages' input data, and at a critical point forwards data and work to other nodes in the cluster.

A different model is to start `xval` on all nodes, and for all nodes to input and massage data. At the critical section of code marked by `xvalLoop`, each node performs only its own work, and arranges to communicate results between all nodes. This model of computation is potentially very efficient, avoiding communication of large data betweeen nodes. We will now explore how to implement this solution.

Consider the following commands:

```
> bcast.Rcmd(ff <- function() {
+     f <- function(i) mpi.comm.rank()
+     sapply(1:20, f)
+ })

function ()
{
    f <- function(i) mpi.comm.rank()
    sapply(1:20, f)
}

> bcast.Rcmd(ff())

 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

So far not very interesting! One way to think the function `ff` is as a small program. `bcast.Rcmd` is a function I wrote for this practical, available in the file `~/rsrsc/batch.R`. It evaluates the single program on all nodes, including the master node. The first `bcast.Rcmd` assigns the value `ff` to all nodes on the cluster. The second `bcast.Rcmd` evaluates `ff` on all nodes. `bcast.Rcmd` returns the value of the program as run on the manager node.

We now introduce a 'wrapper' that changes the way evaluation of `f` works.

```
> bcast.Rcmd(ff <- function() {
+     f <- function(i) mpi.comm.rank()
+     ff <- wrap(f, -1)
+     sapply(1:20, ff)
+ })

function ()
{
    f <- function(i) mpi.comm.rank()
    ff <- wrap(f, -1)
    sapply(1:20, ff)
}

> bcast.Rcmd(ff())

 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
```

The `wrap` function (also written for this practical) takes a function and 'wraps' it so that successive calls to the function only sometimes compute a new value. The other times, it returns a previously-stored psuedo-value. Computations occur in a round-robin fashion – first node 1, then node 2, etc. The manager node never evaluates the function, but instead collects and coordinates the results of the worker nodes. The end result is that the manager node collects the round-robin evaluations from all nodes, returning something approaching a useful result.

Here is a slightly more extensive example, where a bootstrap calculation is distributed across nodes in a cluster.

```
> bcast.Rcmd(ff <- function() {
+     library(boot)
+     ratio <- function(d, w) sum(d$x * w)/sum(d$u * w)
+     ratiow <- wrap(ratio, -1)
+     boot(city, ratiow, R = 999, stype = "w")
+ })

function ()
{
    library(boot)
    ratio <- function(d, w) sum(d$x * w)/sum(d$u * w)
    ratiow <- wrap(ratio, -1)
    boot(city, ratiow, R = 999, stype = "w")
}

> bcast.Rcmd(ff())

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
```

```
boot(data = city, statistic = ratiow, R = 999, stype = "w")


Bootstrap Statistics :
    original     bias    std. error
t1* 1.520313 0.03311055   0.2092155
```

Notice that this is changed only a very little from the single-processor evaluation.

Here is a sketch of the `wrap` function:

```
> wrap <- function(func, pseudo, comm = 1) {
+     iter <- -1
+     sz <- mpi.comm.size(comm) - 1
+     rank <- mpi.comm.rank(comm)
+     tag <- list(result = 1)
+     if (rank == 0)
+         function(...) {
+             iter <<- iter + 1
+             mpi.recv.Robj(iter%%sz + 1, tag$result, comm)
+         }
+     else function(...) {
+         iter <<- iter + 1
+         if (iter%%sz + 1 == rank) {
+             result <- func(...)
+             mpi.send.Robj(result, 0, tag$result, comm)
+         }
+         else result <- pseudo
+         result
+     }
+ }
> mpi.bcast.Robj2slave(wrap)
```

This uses several features of the R language, and as an exercise it would be useful to discuss these with others in the lab.

- In R, a function can be treated just like any other object. What is the return value of `wrap`?
- R is *lexically scoped*. This means that the return values of `wrap` ‘remember’ the environment in which they were defined. So the return value of `wrap` can access the variables `iter, sz, rank` and `tag`.
- A line near then end of `wrap` reads `iter <<- iter + 1`. The R operator `<<-` causes assignment, but unlike the usual assignment statement `<-` it looks for a variable (in this case named `iter`) to assign to in the *environment* where the function is defined. So the effect is to change the value of the `iter` that is defined in the very first line of `wrap`

- `wrap` is really quite incomplete. What issues can you see with it, and how migth it be improved?

A parallel `lapply`

`lapply` provides a very useful framework for embarassingly parallel problems.

```
> args(lapply)

function (X, FUN, ...)
NULL
```

The argument `X` represents the 'work' to be done, `FUN` the program that needs to be evaluated on each piece of work, and `...` the data on which the program is to be evaluated. Here is a batch program that uses `lapplys`, an `lapply`-like alternative, to perform a simple parallel computation:

```
> res <- bcast.Rcmd(lapplys(1:20, function(i) mpi.comm.rank()))
> unlist(res)

 [1] 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

The 'work' is a list from 1 to 20. The function to be performed by each node is to determine the rank of the node. `lapplys` ensures that work gets distributed approximately evenly between nodes, and collates results from each node into a single list (`unlist(res)` simplifies results for compact presentation).

As a more complicated example, here is a way to perform cross-validation in batch mode.

```
> bcast.Rcmd(ff <- function() {
+     library(MLInterfaces)
+     setClass("BatchXval", representation("list"))
+     setMethod("xvalLoop", signature(cluster = "BatchXval"),
+         function(cluster, ...) lapplys)
+     cluster <- new("BatchXval")
+     data(golubMerge)
+     smallG <- golubMerge[200:250, ]
+     lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
+         group = as.integer(0), cluster = cluster)
+     table(lk1, smallG$ALL.AML)
+ })

function ()
{
    library(MLInterfaces)
    setClass("BatchXval", representation("list"))
    setMethod("xvalLoop", signature(cluster = "BatchXval"), function(cluster,
        ...) lapplys)
    cluster <- new("BatchXval")
    data(golubMerge)
    smallG <- golubMerge[200:250, ]
    lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
        group = as.integer(0), cluster = cluster)
    table(lk1, smallG$ALL.AML)
}
```

```
> bcast.Rcmd(ff())
```

```
lk1   ALL AML
  ALL  37  10
  AML  10  15
```

- Describe what steps the flow of execution of this program. Which node(s) read data from disk? How much communication is likely occurring between nodes?
- Based on experience earlier in the lab, how do you think this program performs as more nodes are used in the computation?

Let's take a closer look at `lapplys`.

```
> lapplys <- function(X, FUN, ..., comm = 1) {
+     rank <- mpi.comm.rank(comm) + 1
+     n <- mpi.comm.size(comm)
+     tasks <- 1:length(X)
+     mywork <- X[split(tasks, cut(tasks, n))[[rank]]]
+     result <- lapply(mywork, FUN, ...)
+     allgather.Robj(result, comm)
+ }
> mpi.bcast.Robj2slave(lapplys)
```

The function has nearly the same signature as `lapply`. The first several lines make `lapplys` slightly different on each node that it runs on. The line `mywork<-...` divides `X` into approximately evenly sized lists, and selects a different list for each node. The next line evaluates `FUN` for a subset of the original `X`, with the subset differing depending on the the node. The final line is an Rmpi extension that collates results from different nodes.

`lapplys` makes use of an extension to Rmpi

```
> allgather.Robj <- function(obj = NULL, comm = 1) {
+     obj <- as.integer(charToRaw(serialize(obj, NULL)))
+     sz <- mpi.allgather(length(obj), 1, integer(mpi.comm.size(comm)),
+         comm)
+     objs <- mpi.allgatherv(obj, 1, integer(sum(sz)),
+         sz, comm)
+     as.list(unlist(lapply(split(as.raw(objs), rep(1:length(sz) -
+           1, sz)), unserialize), recursive = FALSE, use.names = FALSE))
+ }
> mpi.bcast.Robj2slave(allgather.Robj)
```

Allgather is an MPI concept, where objects from all nodes are collated and redistributed to all nodes. The `allgather.Robj` is an extension to this, and is included here to illustrate how MPI routines can be included in R functions for easy evaluation.

Finally, `bcast.Rcmd` is defined on the manager node to broadcast commands to all workers, and to evaluate the same commands on the manager.

```
> bcast.Rcmd <- function(cmd = NULL, rank = 0, comm = 1) {
+     if (mpi.comm.rank(comm) == rank) {
+         cmdp <- deparse(substitute(cmd), width.cutoff = 500)
+         cmdp <- paste(cmdp, collapse = "\"\"/")
+         mpi.bcast(x = nchar(cmdp), type = 1, rank = rank,
+             comm = comm)
+         mpi.bcast(x = cmdp, type = 3, rank = rank, comm = comm)
+         eval(cmd, parent.frame())
+     }
+     else {
+         charlen <- mpi.bcast(x = integer(1), type = 1,
+             rank = rank, comm = comm)
+         if (is.character(charlen))
+             parse(text = "break")
+         else {
+             out <- mpi.bcast(x = .Call("mkstr", as.integer(charlen),
+                 PACKAGE = "Rmpi"), type = 3, rank = rank,
+                 comm = comm)
+             parse(text = unlist(strsplit(out, "\"\"/")))
+         }
+     }
+ }
```

This command makes use of several R language concepts, including the ability to deparse and serialize R objects (including functions), and to evaluate R objects in environments different from the calling environment.

Conclusions

We have covered alot of ground in the lab. Here is a brief summary of the highlights:

1. There are at present few 'out of the box' solutions for parallel programming in R; you'll have to grapple with at least some details of parallelization.

2. Not all parts of a script can be usefully parallelized.

3. Communication costs can be important, and for large data sets can outweigh much of the benefit of faster computation.

4. Even without communication costs, the decrease in compute time is inversely proportional to the number of nodes available: adding 1 node to a 1 node cluster doubles potential computational throughput, but adding 1 node to a 10 node cluster matters hardly at all.

5. Repeatability is essential for scientific research, and can be a thorny issue in a parallel environment.

6. Rmpi provides some functionality for parallel programming, including an interface to lower-level MPI functions.

7. Useful new parallel techniques introduced in the lab rely on executing a series of commands as a single program on the remote and manager nodes.

8. This technique is useful in conjunction with a 'wrapper' that determines which nodes engage in expensive computations, and `lapplys`, an `lapply`-like function for distributing work evenly between nodes.

Needless to say, we have barely scratched the surface of possibility. The algorithms presented here are only a fraction of those useful in parallel computation. They lack any error checking or recovery, and make no attempt to check-point or otherwise ensure against system failure. These are the directions for exploration.