An Introduction to Parallel Processing in R, Including Use on Clusters and in the Cloud

Chris Paciorek
Department of Statistics
University of California, Berkeley

February 27, 2015

1 How to follow and try out this material

Note that my examples here will be silly toy examples for the purpose of keeping things simple and focused on the parallelization approaches.

I will do demos on an Ubuntu Linux virtual machine (VM), on the biostat cluster, and on Amazon's AWS.

We'll use the BCE Virtual Machine. You can run this on your laptop (see BCE Virtual Machine for installation instructions), and I encourage you to do so to follow along. Note: to allow for parallelization, before starting the BCE VM, go to Machine > Settings, select System and increase the number of processors.

We'll also use BCE as the basis for the virtual machines we start on Amazon's AWS.

2 Overview of parallel processing computers

There are two basic flavors of parallel processing (leaving aside GPUs): distributed memory and shared memory. With shared memory, multiple processors (which I'll call cores) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. You can think of each node as a separate computer connected by a fast network.

2.1 Some useful terminology:

- *cores*: We'll use this term to mean the different processing units available on a single node.

- *nodes*: We'll use this term to mean the different computers, each with their own distinct memory, that make up a cluster or supercomputer.
- *processes*: computational tasks executing on a machine. A given program may start up multiple processes at once. Ideally we have no more processes than cores on a node.
- *thread*s: multiple paths of execution within a single process; the OS sees the threads as a single process, but one can think of them as 'lightweight' processes. Ideally when considering the processes and their threads, we would have no more processes and threads combined than cores on a node.
- *forking*: child processes are spawned that are identical to the parent, but with different process id's and their own memory.
- *sockets*: some of R's parallel functionality involves creating new R processes (e.g., starting processes via *Rscript*) and communicating with them via a communication technology called sockets.

2.2 Shared memory

For shared memory parallelism, each core is accessing the same memory so there is no need to pass information (in the form of messages) between different machines. But in some programming contexts one needs to be careful that activity on different cores doesn't mistakenly overwrite places in memory that are used by other cores.

Some of the shared memory parallelism approaches that we'll cover are:

1. threaded linear algebra
2. simple parallelization of embarrassingly parallel computations

Threading Threads are multiple paths of execution within a single process. Using *top* to monitor a job that is executing threaded code, you'll see the process using more than 100% of CPU. When this occurs, the process is using multiple cores, although it appears as a single process rather than as multiple processes. In general, threaded code will detect the number of cores available on a machine and make use of them. However, you can also explicitly control the number of threads available to a process.

For most threaded code (that based on the openMP protocol), the number of threads can be set by setting the OMP_NUM_THREADS environment variable (VECLIB_MAXIMUM THREADS on a Mac). E.g., to set it for four threads in bash:

```
export OMP_NUM_THREADS=4
```

Matlab is an exception to this. Threading in Matlab can be controlled in two ways. From within your Matlab code you can set the number of threads, e.g., to four in this case:

```
feature('numThreads', 4)
```

To use only a single thread, you can use 1 instead of 4 above, or you can start Matlab with the *singleCompThread* flag:

```
matlab -singleCompThread ...
```

2.3 Distributed memory

Parallel programming for distributed memory parallelism requires passing messages between the different nodes. The standard protocol for doing this is MPI, of which there are various versions, including *openMPI*. The Python package *mpi4py* implements MPI in Python and the R package *Rmpi* implements MPI in R.

Some of the distributed memory approaches that we'll cover are:

1. simple parallelization of embarrassingly parallel computations
2. using MPI for explicit distributed memory processing

2.4 Other type of parallel processing

We won't cover either of the following in this material.

2.4.1 GPUs

GPUs (Graphics Processing Units) are processing units originally designed for rendering graphics on a computer quickly. This is done by having a large number of simple processing units for massively parallel calculation. The idea of general purpose GPU (GPGPU) computing is to exploit this capability for general computation.

In spring 2014, I gave a workshop on using GPUs. One easy way to use a GPU is on an Amazon EC2 virtual machine.

2.4.2 Spark and Hadoop

Spark and Hadoop are systems for implementing computations in a distributed memory environment, using the MapReduce approach. In fall 2014 I gave a workshop on Spark. One easy way to use Spark is on a cluster of Amazon EC2 virtual machines.

3 Basic suggestions for parallelizing your code

The easiest situation is when your code is embarrassingly parallel, which means that the different tasks can be done independently and the results collected. When the tasks need to interact, things get much harder. Much of the material here is focused on embarrassingly parallel computation.

- If you can do your computation on the cores of a single node using shared memory, that will be faster than using the same number of cores (or likely even than somewhat more cores) across multiple nodes. Similarly, jobs with a lot of data/high memory requirements that one might think of as requiring Hadoop may in some cases be much faster if you can find a single machine with a lot of memory.
- If you have nested loops, you generally only want to parallelize at one level of the code. That said, there may be cases in which it is helpful to do both. Keep in mind whether your linear algebra is being threaded. Often you will want to parallelize over a loop and not use threaded linear algebra. (That said, if you have multiple nodes, you might have one task per node and use threaded linear algebra to exploit the cores on each node.)
- Often it makes sense to parallelize the outer loop when you have nested loops.
- You generally want to parallelize in such a way that your code is load-balanced and does not involve too much communication.
  - If you have very few tasks, particularly if the tasks take different amounts of time, often some processors will be idle and your code poorly load-balanced.
  - If you have very many tasks and each one takes little time, the communication overhead of starting and stopping the tasks will reduce efficiency.

I'm happy to help discuss specific circumstances, so just email consult@stat.berkeley.edu. The new Berkeley Research Computing (BRC) initiative is also providing consulting on efficient parallelization strategies. If my expertise is not sufficient, I can help you get assistance from BRC.

Static vs. dynamic assignment of tasks Some of R's parallel functions allow you to say whether the tasks can be divided up and allocated to the workers at the beginning or whether tasks should be assigned individually as previous tasks complete. E.g., the *mc.preschedule* argument in *mclapply()* and the *.scheduling* argument in *parLapply()*.

Basically if you have many tasks that each take similar time, you want to preschedule to reduce communication. If you have few tasks or tasks with highly variable completion times, you don't want to preschedule, to improve load-balancing.

4 Threaded linear algebra and the BLAS

The BLAS is the library of basic linear algebra operations (written in Fortran or C). A fast BLAS can greatly speed up linear algebra relative to the default BLAS on a machine. Some fast BLAS libraries are Intel's *MKL*, AMD's *ACML*, and the open source (and free) *openBLAS* (formerly *GotoBLAS*). For the Mac, there is *vecLib* BLAS. All of these BLAS libraries are now threaded - if your computer has multiple cores and there are free resources, your linear algebra will use multiple cores, provided your program is linked against the specific BLAS and provided OMP_NUM_THREADS is not set to one. (Macs make use of VECLIB_MAXIMUM_THREADS rather than OMP_NUM_THREADS.)

4.1 Fixing the number of threads (cores used)

In general, if you want to limit the number of threads used, you can set the OMP_NUM_THREADS environment variable on UNIX machine. This can be used in the context of R or C code that uses BLAS or your own threaded C code, but this does not work with Matlab. In the UNIX shell, you'd do this as follows (e.g. to limit to 3 cores):

```
export OMP_NUM_THREADS=3 # bash
setenv OMP_NUM_THREADS 3 # tcsh
```

If you are running R, you'd need to do this in your shell session before invoking R.

4.2 Using threading in R

Threading in R is limited to linear algebra, for which R calls external BLAS and LAPACK libraries.

Here's some code that when run in an R executable linked to a threaded BLAS illustrates the speed of using a threaded BLAS:

```
require(RhpcBLASctl)

## Loading required package:  RhpcBLASctl

# I use RhpcBLASctl to control threading for purpose of demo
# but one can also set OMP_NUM_THREADS in the shell before invoking R
x <- matrix(rnorm(5000^2), 5000)

blas_set_num_threads(4)
system.time({
x <- crossprod(x)
U <- chol(x)
})
```

```
##    user  system elapsed
##  11.630   4.010   3.875

blas_set_num_threads(1)
system.time({
x <- crossprod(x)
U <- chol(x)
})

##    user  system elapsed
##   6.613   0.195   6.621
```

The R installation manual gives information on how to link R to a fast BLAS. On Ubuntu, if you install openBLAS as follows, the */etc/alternatives* system will set */usr/lib/libblas.so* to point to openBLAS. By default on Ubuntu, R will use the system BLAS, so it will as a result use openBLAS.

```
# to install openBLAS
sudo apt-get install -y libopenblas-base

ls -l /usr/lib/libblas.so
## lrwxrwxrwx 1 root root 28 Jan  8 19:11 /usr/lib/libblas.so ->
##     /etc/alternatives/libblas.so
ls -l /etc/alternatives/libblas.so
## lrwxrwxrwx 1 root root 33 Jan 13 16:01 /etc/alternatives/libblas.so ->
##     /usr/lib/openblas-base/libblas.so
```

To use a fast, threaded BLAS enabled on your own Mac, do the following:

```
R_VERSION=3.1

cd /Library/Frameworks/R.framework/Versions/3.1/${R_VERSION}/lib
cp libRblas.dylib libRblas.dylib.backup
ln -s /System/Library/Frameworks/Accelerate.framework/Versions/\
Current/Frameworks/vecLib.framework/Versions/Current/libBLAS.dylib
```

You should be able to use OMP_NUM_THREADS to control the number of threads used for linear algebra on the Biostat cluster.

4.2.1 Important warnings about use of threaded BLAS

Conflict between openBLAS and some parallel functionality in R There are conflicts between forking in R and threaded BLAS that in some cases affect *foreach* (when using the *multicore* and *parallel* backends), *mclapply()*, and (only if *cluster()* is set up with forking (not the default)) *par{L,S,}apply()*. The result is that if linear algebra is used within your parallel code, R hangs. This affects (under somewhat different circumstances) both ACML and openBLAS.

To address this, before running an R job that does linear algebra, you can set OMP_NUM_THREADS to 1 to prevent the BLAS from doing threaded calculations. Alternatively, you can use MPI as the parallel backend (via *doMPI* in place of *doMC* or *doParallel* – see Section 6). You may also be able to convert your code to use *par{L,S,}apply()* [with the default PSOCK type] and avoid *foreach* entirely.

Conflict between threaded BLAS and R profiling There is also a conflict between threaded BLAS and R profiling, so if you are using *Rprof()*, you may need to set OMP_NUM_THREADS to one. This has definitely occurred with openBLAS; I'm not sure about other threaded BLAS libraries.

Speed and threaded BLAS In many cases, using multiple threads for linear algebra operations will outperform using a single thread, but there is no guarantee that this will be the case, in particular for operations with small matrices and vectors. Testing with openBLAS suggests that sometimes a job may take more time when using multiple threads; this seems to be less likely with ACML. This presumably occurs because openBLAS is not doing a good job in detecting when the overhead of threading outweights the gains from distributing the computations. You can compare speeds by setting OMP_NUM_THREADS to different values. In cases where threaded linear algebra is slower than unthreaded, you would want to set OMP_NUM_THREADS to 1.

Therefore I recommend that you test any large jobs to compare performance with a single thread vs. multiple threads. Only if you see a substantive improvement with multiple threads does it make sense to have OMP_NUM_THREADS be greater than one.

5 Basic shared memory parallel programming in R

5.1 foreach

A simple way to exploit parallelism in R when you have an embarrassingly parallel problem (one where you can split the problem up into independent chunks) is to use the *foreach* package to

do a for loop in parallel. For example, bootstrapping, random forests, simulation studies, cross-validation and many other statistical methods can be handled in this way. You would not want to use *foreach* if the iterations were not independent of each other.

The *foreach* package provides a *foreach* command that allows you to do this easily. *foreach* can use a variety of parallel "back-ends". It can use *Rmpi* to access cores in a distributed memory setting as discussed in Section 6 or the *parallel* or *multicore* packages to use shared memory cores. When using *parallel* or *multicore* as the back-end, you should see multiple processes (as many as you registered; ideally each at 100%) when you look at *top*. The multiple processes are created by forking or using sockets; this is discussed a bit more later in this document.

```
require(parallel) # one of the core R packages
require(doParallel)
# require(multicore); require(doMC) # alternative to parallel/doParallel
# require(Rmpi); require(doMPI) # to use Rmpi as the back-end
library(foreach)

taskFun <- function(){
        mn <- mean(rnorm(10000000))
        return(mn)
}
nCores <- 2
registerDoParallel(nCores)
# registerDoMC(nCores) # alternative to registerDoParallel
# cl <- startMPIcluster(nCores); registerDoMPI(cl) # when using Rmpi as the

out <- foreach(i = 1:100) %dopar% {
        cat('Starting ', i, 'th job.\n', sep = '')
        outSub <- taskFun()
        cat('Finishing ', i, 'th job.\n', sep = '')
        outSub # this will become part of the out object
}
```

The result of *foreach* will generally be a list, unless *foreach* is able to put it into a simpler R object. Note that *foreach* also provides some additional functionality for collecting and managing the results that mean that you don't have to do some of the bookkeeping you would need to do if writing your own for loop.

You can debug by running serially using *%do%* rather than *%dopar%*. Note that you may

need to load packages within the *foreach* construct to ensure a package is available to all of the calculations.

Caution: Note that I didn't pay any attention to possible danger in generating random numbers in separate processes. More on this issue in the section on RNG (Section 9).

5.2 parallel apply (parallel package)

The *parallel* package has the ability to parallelize the various *apply()* functions (*apply*, *lapply*, *sapply*, etc.) and parallelize vectorized functions. It's a bit hard to find the vignette for the parallel package because parallel is not listed as one of the contributed packages on CRAN.

First let's consider parallel apply.

```
require(parallel)

## Loading required package:  parallel

nCores <- 2

################################
# using forking (mclapply)
################################

system.time(
        res <- mclapply(input, testFun, mc.cores = nCores)
)

## Error:  object 'input' not found

## Timing stopped at: 0.001 0.001 0

#############################
# using sockets (parLapply)
#############################

# ?clusterApply
cl <- makeCluster(nCores) # by default this uses the PSOCK
#  mechanism as in the SNOW package - starting new jobs via Rscript
#  and communicating via sockets
nSims <- 60
```

```
input <- seq_len(nSims) # same as 1:nSims but more robust
testFun <- function(i){
        mn <- mean(rnorm(1000000))
        return(mn)
}
# clusterExport(cl, c('x', 'y')) # if the processes need objects
#   (x and y, here) from the master's workspace
system.time(
        res <- parSapply(cl, input, testFun)
)

##    user  system elapsed
##   0.003   0.000   4.459

system.time(
        res2 <- sapply(input, testFun)
)

##    user  system elapsed
##   4.950   0.043   4.993

res <- parLapply(cl, input, testFun)
```

Note that some R packages can directly interact with the parallelization packages to work with multiple cores. E.g., the *boot* package can make use of the *multicore* package directly.

5.3 Using mcparallel() to manually parallelize individual tasks

One can use *mcparallel()* in the *parallel* package to send different chunks of code to different processes. Here we would need to manage the number of tasks so that we don't have more tasks than available cores.

```
library(parallel)
n <- 10000000
system.time({
        p <- mcparallel(mean(rnorm(n)))
        q <- mcparallel(mean(rgamma(n, shape = 1)))
        res <- mccollect(list(p,q))
```

```
})

##    user  system elapsed
##   0.785   0.003   1.365

system.time({
        p <- mean(rnorm(n))
        q <- mean(rgamma(n, shape = 1))
})

##    user  system elapsed
##   2.061   0.020   2.080
```

Note that *mcparallel()* also allows the use of the *mc.set.seed* argument as with *mclapply()*.

Note that on the cluster, one should create only as many parallel blocks of code as were requested when submitting the job.

5.4 Running on the Biostat cluster

Here's how you would submit an R job that uses shared memory on a single node (in this case requesting four cores of a maximum of 8 on a given core):

```
qsub -pe smp 4 job.sh
```

job.sh would have the various lines starting with #$ as in the biostat cluster documentation and your R command would look like:

```
R CMD BATCH file.R file.out
```

You can make use of the UNIX environment variable NSLOTS in your code to programmatically control the number of the number of cores your code uses (e.g., with foreach, mclapply, parLapply, etc.). To read this variable in R do:

```
ncores <- Sys.getenv('NSLOTS')
```

6 Distributed memory

6.1 Use of MPI as a black box for embarrassingly parallel computation

6.1.1 foreach in R on a single node

One reason to use Rmpi on a single node is that in some contexts the threaded BLAS (particularly openBLAS) conflicts with *multicore*/*parallel*'s forking functionality. This can cause *foreach*

to hang when used with the *doParallel* or *doMC* parallel back ends (also with *mclapply()*) and OMP_NUM_THREADS set to a value greater than one.

Here's R code for using *Rmpi* as the back-end to *foreach*.

```
library(Rmpi)
library(doMPI)

nCores = 2

cl = startMPIcluster(nCores)

registerDoMPI(cl)
clusterSize(cl) # just to check

nIts <- 20

results <- foreach(i = 1:nIts) %dopar% {
  out = mean(rnorm(1e7))
}

print(unlist(results))

closeCluster(cl)

mpi.quit()
```

A caution concerning Rmpi/doMPI: when you invoke *startMPIcluster()*, all the slave R processes become 100% active and stay active until the cluster is closed. In addition, when *foreach* is actually running, the master process also becomes 100% active. So using this functionality involves some inefficiency in CPU usage. This inefficiency is not seen with a sockets cluster (see next) nor when using other Rmpi functionality - i.e., starting slaves with *mpi.spawn.Rslaves()* and then issuing commands to the slaves.

6.1.2 foreach in R on multiple nodes

For this, we need to start R through the *mpirun* command so that inter-node communication is handled by MPI.

```
mpirun -machinefile .hosts -np 4 R CMD BATCH --no-save file.R file.out
mpirun -machinefile .hosts -np 4 R --no-save
```

Here's R code for using *Rmpi* as the back-end to *foreach*. If you call *startMPIcluster()* with no arguments, it will start up one fewer worker processes than the value indicated by the np flag, so your R code will be more portable. Also you can leave off the -np and mpirun will start up as many processes as there are lines in the .hosts file.

```
library(Rmpi)
library(doMPI)

cl = startMPIcluster()  # by default will start one fewer slave
# than elements in .hosts


registerDoMPI(cl)
clusterSize(cl) # just to check

nIts <- 20

results <- foreach(i = 1:nIts) %dopar% {
  out = mean(rnorm(1e7))
}

closeCluster(cl)

mpi.quit()
```

Note that the above should work with -np 1 as well (and does on the biostat cluster). However -np 1 does not work in the Amazon cluster nor the SCF (though it has in the past). I'm in the process of investigating this.

6.1.3 Sockets in R example

One can also set up a cluster via sockets. You just need to specify a character vector with the machine names as the input to *makeCluster()*.

```
# multinode example with PSOCK cluster

library(parallel)

machineVec = c(rep("master", 2),
    rep("node001", 2),
    rep("node002", 2))
cl = makeCluster(machineVec)

n = 1e7
clusterExport(cl, c('n'))
fun = function(i)
  out = mean(rnorm(n))

result <- parSapply(cl, 1:20, fun)

stopCluster(cl) # not strictly necessary
```

6.2 MPI basics

There are multiple MPI implementations, of which *openMPI* and *mpich* are very common.

In MPI programming, the same code runs on all the machines. This is called SPMD (single program, multiple data). As we saw above, one invokes the same code (same program) multiple times, but the behavior of the code can be different based on querying the rank (ID) of the process. Since MPI operates in a distributed fashion, any transfer of information between processes must be done explicitly via send and receive calls (e.g., *MPI_Send*, *MPI_Recv*, *MPI_Isend*, and *MPI_Irecv*). (The "MPI_" is for C code; C++ just has *Send*, *Recv*, etc.)

The latter two of these functions (*MPI_Isend* and *MPI_Irecv*) are so-called non-blocking calls. One important concept to understand is the difference between blocking and non-blocking calls. Blocking calls wait until the call finishes, while non-blocking calls return and allow the code to continue. Non-blocking calls can be more efficient, but can lead to problems with synchronization between processes.

In addition to send and receive calls to transfer to and from specific processes, there are calls that send out data to all processes (*MPI_Scatter*), gather data back (*MPI_Gather*) and perform reduction operations (*MPI_Reduce*).

Debugging MPI/*Rmpi* code can be tricky because communication can hang, error messages from the workers may not be seen or readily accessible and it can be difficult to assess the state of the worker processes.

First let's get a feel for MPI in the simple context of a single node. Of course it's a bit silly to do this in reality on a single node on which one can take advantage of shared memory. There's not much reason to use MPI on a single node as there's a cost to the message passing relative to shared memory, but it is useful to be able to test the code on a single machine without having to worry about networking issues across nodes.

6.2.1 Using Rmpi to write code for distributed computation

R users can use *Rmpi* to interface with MPI. To use *Rmpi*, you can simply start R as you normally do by invoking a command-line R session or using R CMD BATCH. To use Rmpi across multiple nodes you'll need to invoke R with mpirun as in the example of foreach with doMPI on multiple nodes above.

Here's some example code that uses actual *Rmpi* syntax (as opposed to *foreach* with Rmpi as the back-end). This code runs in a master-slave paradigm where the master starts the slaves and invokes commands on them. It may be possible to run *Rmpi* in a context where each process runs the same code based on invoking with Rmpi, but I haven't investigated this further.

To start your R job do the following. Here one requests just a single process, as R will manage the task of spawning slaves.

```
mpirun -machinefile .hosts -np 1 R CMD BATCH --no-save file.R file.Rout
```

```
# example syntax of standard MPI functions

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: smeagol
## slave1 (rank 1, comm 1) of size 4 is running on: smeagol
## slave2 (rank 2, comm 1) of size 4 is running on: smeagol
## slave3 (rank 3, comm 1) of size 4 is running on: smeagol

n = 5
mpi.bcast.Robj2slave(n)
mpi.bcast.cmd(id <- mpi.comm.rank())
mpi.bcast.cmd(x <- rnorm(id))
```

```
mpi.remote.exec(ls(.GlobalEnv), ret = TRUE)

## $slave1
## [1] "id" "n"  "x"
##
## $slave2
## [1] "id" "n"  "x"
##
## $slave3
## [1] "id" "n"  "x"

mpi.bcast.cmd(y <- 2 * x)
mpi.remote.exec(print(y))

## $slave1
## [1] -0.9796
##
## $slave2
## [1]  1.4292 -0.4196
##
## $slave3
## [1] -0.1208  2.6639 -0.6288

objs <- c('y', 'z')
# next command sends value of objs on _master_ as argument to rm
mpi.remote.exec(rm, objs)

## $slave2
## [1] 0

mpi.remote.exec(print(z))

## $slave1
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
```

```
## <simpleError in print(z): object 'z' not found>
##
## $slave2
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>
##
## $slave3
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>
```

```
# collect results back via send/recv
mpi.remote.exec(mpi.send.Robj(x, dest = 0, tag = 1))
```

```
## $slave1
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  93190  5.0     350000 18.7   347671 18.6
## Vcells 184733  1.5     786432  6.0   637831  4.9
##
## $slave2
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  93190  5.0     350000 18.7   347671 18.6
## Vcells 184734  1.5     786432  6.0   637831  4.9
##
## $slave3
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  93190  5.0     350000 18.7   347671 18.6
## Vcells 184736  1.5     786432  6.0   637831  4.9
```

```
results = list()
for(i in 1:(mpi.comm.size()-1)){
  results[[i]] = mpi.recv.Robj(source = i, tag = 1)
```

```
}

print(results)

## [[1]]
## [1] -0.4898
##
## [[2]]
## [1]  0.7146 -0.2098
##
## [[3]]
## [1] -0.06038  1.33193 -0.31440
```

CAUTION: for some reason this syntax is not distributing workers to the other nodes listed in the hosts file on the Amazon cluster and the SCF, but it should work on the Biostat cluster. This has worked in the past for me, so I'm not sure what is going on, but it may be a difference in the MPI version or something in the latest Ubuntu (Ubuntu 14 vs. Ubuntu 12).

Note that if you do this in interactive mode, some of the usual functionality of command line R (tab completion, scrolling for history) is not enabled and errors will cause R to quit. This occurs because passing things through *mpirun* causes R to think it is not running interactively.

Note on supercomputers Note: in some cases a cluster/supercomputer will be set up so that *Rmpi* is loaded and the worker processes are already started when you start R. In this case you wouldn't need to load *Rmpi* or use *mpi.spawn.Rslaves()*. You can always check *mpi.comm.size()* to see if the workers are already set up.

6.2.2 Using *pbdR* for distributed computation

There is a relatively new effort to enhance R's capability for distributed memory processing called *pbdR*. *pbdR* is designed for SPMD processing in batch mode. *pbdR* provides the following capabilities:

- an alternative to Rmpi for interfacing with MPI
- the ability to do some parallel apply-style computations
- the ability to do distributed linear algebra by interfacing to ScaLapack

Personally, I think the last of the three is the most exciting as its a functionality not readily available in R or even more generally in other readily-accessible software.

pbdR is installed on the nodes in our example EC2 cluster, as detailed in Section X.

Let's see some basic examples of using pbdR.

One starts pbdR via mpirun as follows:

```
mpirun -machinefile .hosts -np 4 Rscript file.R > file.out
```

Interfacing with MPI Here's an example of distributing an embarrassingly parallel calculation (estimating an integral via Monte Carlo - in this case estimating the value of π).

```
library(pbdMPI, quiet = TRUE )
init()

if(comm.rank()==0) {
    x <- matrix(rnorm(1e6*50), 1e6)
}

sm <- comm.timer(pbdApply(x, 1, mean, pbd.mode = 'mw', rank.source = 0))
if(comm.rank()==0) {
    print(sm)
}

finalize()
```

Parallel apply Here's some basic syntax for doing a distributed *apply()* on a matrix that is on one of the workers (i.e., the matrix is not distributed).

Distributed linear algebra And here's how you would set up a distributed matrix and do linear algebra on it. Note that when working with large matrices, you would generally want to construct the matrices (or read from disk) in a parallel fashion rather than creating the full matrix on one worker.

```
library(pbdDMAT, quiet = TRUE )
init()

n <- 4096*2
```

```
library(RhpcBLASctl)
blas_set_num_threads(1)

init.grid()

if(comm.rank()==0) print(date())

# pbd allows for parallel I/O, but here
# we keep things simple and distribute
# an object from one process
if(comm.rank() == 0) {
    x <- rnorm(n^2)
    dim(x) <- c(n, n)
} else x <- NULL
dx <- as.ddmatrix(x)

timing <- comm.timer(sigma <- crossprod(dx))

if(comm.rank()==0) {
    print(date())
    print(timing)
}

timing <- comm.timer(out <- chol(sigma))

if(comm.rank()==0) {
    print(date())
    print(timing)
}


finalize()
```

6.3 Running on the Biostat cluster

Here's how you would run distributed R jobs based on mpirun on the Biostat cluster

```
qsub -pe orte 12 job.sh
```

job.sh would have the various lines starting with #$ as in the biostat cluster documentation.

For Rmpi/doMPI your R command would look like this, where R will start the worker processes (hence the -np 1)

```
mpirun -np 1 R CMD BATCH file.R file.out
```

or for pbdR where mpirun starts the workers in SPMD fashion:

```
mpirun -np 12 Rscript file.R > file.out
```

7 Using R on Cloud VMs and Cloud-based Clusters

For Amazon-based machines, we'll use the BCE image as our starting point and add software to it.

WARNING: do not share your Amazon authentication info (AWS_ACCESS_KEY_ID or AWS_SECRET_ACCESS_KEY) in any public setting, such as a public Github account. A user in my class did this and we ended up with thousands of dollars in charges from hackers starting up instances (probably to mine Bitcoin).

7.1 Starting an Amazon EC2 Instance

Point-and-click via the EC2 console To start up an EC2 virtual machine based on BCE, follow the instructions at bce.berkeley.edu/install.html.

To logon, you'll need to make use of SSH keys. Here's how to get things going with BCE such that you can logon as the oski user. You obtain the <IP_address> by clicking on the "Connect" button on the EC2 console.

```
ssh -i ~/.ssh/your_private_key_file ubuntu@<IP_address> sudo bash modify-fo
ssh -i ~/.ssh/your_private_key_file ubuntu@<IP_address> sudo bash add-paral
# Now you can ssh in directly as the oski user.
ssh -i ~/.ssh/your_private_key_file oski@<IP_address>
```

[show Aaron's approach]

Once you've logged into the instance, copy the files modify-for-aws.sh and add-parallel-tools.sh to the instance and run them:

```
bash modify-for-aws.sh
bash add-parallel-tools.sh
```

Now you can logon to the instance as the oski user (or just do "su - oski" if you are logged on as root) and do your work.

Once you are logged in to the VM, running parallel R jobs on a single node should be exactly the same as we've already seen on a BCE VM running on your laptop.

By scripting Aaron Culich of Berkeley Research Computing recommends the following scripted approach.

7.2 Starting a Cluster of EC2 Instances

We'll use a tool called Starcluster to start EC2 clusters. This manages setting up the networking between the nodes. You'll need Starcluster installed on your local machine.

The file *starcluster.config* contains some basic settings you can use to start up an EC2 cluster based on BCE. In particular it has the AMI IDs for the BCE EC2 images. You'll need to put your own AWS authentication information in the file (or set these as environment variables in your shell session). Rename the file *config* and put it in a directory called *.starcluster* in your home directory. Then you can start an EC2 cluster and login as follows:

```
starcluster start -c bce mycluster

starcluster put mycluster modify-for-aws.sh .
starcluster sshmaster mycluster bash modify-for-aws.sh

starcluster put -u oski mycluster add-parallel-tools.sh .
starcluster put -u oski mycluster add-parallel-starcluster.sh .

starcluster sshmaster -u oski mycluster sudo bash add-parallel-tools.sh
starcluster sshmaster -u oski mycluster sudo bash add-parallel-starcluster.s
```

When you start the cluster via `starcluster start`, you should see logging information that looks like this, followed by some instructions about how to restart, stop, and login to your cluster.

```
StarCluster - (http://star.mit.edu/cluster) (v. 0.95.6)
Software Tools for Academics and Researchers (STAR)
Please submit bug reports to starcluster@mit.edu
```

```
>>> Validating cluster template settings...
>>> Cluster template settings are valid
>>> Starting cluster...
>>> Launching a 4-node cluster...
>>> Creating security group @sc-mycluster...
Reservation:r-e344eee9
>>> Waiting for instances to propagate...
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Waiting for cluster to come up... (updating every 30s)
>>> Waiting for all nodes to be in a 'running' state...
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Waiting for SSH to come up on all nodes...
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Waiting for cluster to come up took 1.591 mins
>>> The master node is ec2-52-10-188-152.us-west-2.compute.amazonaws.com
>>> Configuring cluster...
>>> Running plugin starcluster.clustersetup.DefaultClusterSetup
>>> Configuring hostnames...
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Creating cluster user: oski (uid: 1001, gid: 1001)
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Configuring scratch space for user(s): oski
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Configuring /etc/hosts on each node
4/4 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Starting NFS server on master
>>> Configuring NFS exports path(s):
/home
>>> Mounting all NFS export path(s) on 3 worker node(s)
3/3 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 100
>>> Setting up NFS took 0.070 mins
>>> Configuring passwordless ssh for root
>>> Configuring passwordless ssh for oski
>>> Configuring cluster took 0.227 mins
>>> Starting cluster took 1.860 mins
```

The other commands after the `starcluster start` customize the EC2 cluster with some

parallel R (and Python) tools, as well as MPI. In particular note that I've created a *.hosts* file for use with *mpirun*. The host file just contains lines with *master*, *node001*, *node002*, ..., which are the internal names of the nodes in the cluster that Starcluster has set.

8 Efficiency and Profiling

8.1 General tips on efficient R code

See Section 1 of Unit 6 from Statistics 243 on efficient R code.

Here are a few high-level principles:

1. Write vectorized code, not for loops or apply-style statements.
2. Pre-allocate storage rather than allocating into a position in a vector or list that is beyond the length of that vector/list.
3. Make sure R is linked to a fast BLAS.
4. To get one or more values from a vector or list, use numeric indices rather than looking up/subsetting by name or boolean values.

8.2 Benchmarking and profiling

Again my class notes (see above link) provide more details on the topics below.

The system.time() function allows you to time R code. The rbenchmark package provides a nice wrapper function, benchmark(), that eases timing of code and comparison of timing of different code chunks.

The Rprof() function allows you to see how much time is spent in different parts of your code, helping to identify bottlenecks. Rprof() works as follows. At set time intervals, it queries the R process to see what function is being evaluated and writes this info to disk. Then when the code is finished you can call summaryRprof() to collect those statistics. Note that if your code runs very quickly, you may not get enough queries accumulated to have useful information. See the 'interval' argument to Rprof. In this case you may be able to write a loop that repeatedly evaluates your code.

I haven't used it, but Hadley Wickham's lineprof package provides a nice interface that allows you to see timing (as with Rprof()) but also memory use by lines of your code. Some details are in his book.

9 RNG

The key thing when thinking about random numbers in a parallel context is that you want to avoid having the same 'random' numbers occur on multiple processes. On a computer, random numbers are not actually random but are generated as a sequence of pseudo-random numbers designed to mimic true random numbers. The sequence is finite (but very long) and eventually repeats itself. When one sets a seed, one is choosing a position in that sequence to start from. Subsequent random numbers are based on that subsequence. All random numbers can be generated from one or more random uniform numbers, so we can just think about a sequence of values between 0 and 1.

The worst thing that could happen is that one sets things up in such a way that every process is using the same sequence of random numbers. This could happen if you mistakenly set the same seed in each process, e.g., using *set.seed(mySeed)* in R on every process.

The naive approach is to use a different seed for each process. E.g., if your processes are numbered $id = 1, \ldots, p$, with *id* unique to a process, using *set.seed(id)* on each process. This is likely not to cause problems, but raises the danger that two (or more sequences) might overlap. For an algorithm with dependence on the full sequence, such as an MCMC, this probably won't cause big problems (though you likely wouldn't know if it did), but for something like simple simulation studies, some of your 'independent' samples could be exact replicates of a sample on another process. Given the period length of the default generators in R, Matlab and Python, this is actually quite unlikely, but it is a bit sloppy.

One approach to avoid the problem is to do all your RNG on one process and distribute the random deviates, but this can be infeasible with many random numbers.

More generally to avoid this problem, the key is to use an algorithm that ensures sequences that do not overlap.

9.1 Ensuring separate sequences in R

In R, there are two packages that deal with this, *rlecuyer* and *rsprng*. We'll go over *rlecuyer*, as I've heard that *rsprng* is deprecated (though there is no evidence of this on CRAN) and *rsprng* is (at the moment) not available for the Mac.

The L'Ecuyer algorithm has a period of 2^{191}, which it divides into subsequences of length 2^{127}.

9.1.1 With the parallel package

Here's how you initialize independent sequences on different processes when using the *parallel* package's parallel apply functionality (illustrated here with *parSapply()*.

```
require(parallel)
require(rlecuyer)
nSims <- 250
testFun <- function(i){
        val <- runif(1)
        return(val)
}

nSlots <- 2
RNGkind()
cl <- makeCluster(nSlots)
iseed <- 0
# ?clusterSetRNGStream
clusterSetRNGStream(cl = cl, iseed = iseed)
RNGkind() # clusterSetRNGStream sets RNGkind as L'Ecuyer-CMRG
# but it doesn't show up here on the master
res <- parSapply(cl, 1:nSims, testFun)
clusterSetRNGStream(cl = cl, iseed = iseed)
res2 <- parSapply(cl, 1:nSims, testFun)
identical(res,res2)
stopCluster(cl)
```

If you want to explicitly move from stream to stream, you can use *nextRNGStream()*. For example:

```
RNGkind("L'Ecuyer-CMRG")
seed <- 0
set.seed(seed) ## now start M workers
s <- .Random.seed
for (i in 1:M) {
        s <- nextRNGStream(s)
        # send s to worker i as .Random.seed
}
```

When using *mclapply()*, you can use the *mc.set.seed* argument as follows (note that *mc.set.seed* is TRUE by default, so you should get different seeds for the different processes by default), but one

needs to invoke `RNGkind("L'Ecuyer-CMRG")` to get independent streams via the L'Ecuyer algorithm.

```
require(parallel)
require(rlecuyer)
RNGkind("L'Ecuyer-CMRG")
res <- mclapply(seq_len(nSims), testFun, mc.cores = nSlots,
    mc.set.seed = TRUE)
# this also seems to reset the seed when it is run
res2 <- mclapply(seq_len(nSims), testFun, mc.cores = nSlots,
    mc.set.seed = TRUE)
identical(res,res2)
```

The documentation for *mcparallel()* gives more information about reproducibility based on *mc.set.seed*.

9.1.2 With foreach

Getting independent streams One question is whether *foreach* deals with RNG correctly. This is not documented, but the developers (Revolution Analytics) are well aware of RNG issues. Digging into the underlying code reveals that the *doMC* and *doParallel* backends both invoke *mclapply()* and set *mc.set.seed* to TRUE by default. This suggests that the discussion above r.e. *mclapply()* holds for *foreach* as well, so you should do `RNGkind("L'Ecuyer-CMRG")` before your foreach call. For *doMPI*, as of version 0.2, you can do something like this, which uses L'Ecuyer behind the scenes:

```
cl <- makeCluster(nSlots)
setRngDoMPI(cl, seed=0)
```

Ensuring reproducibility While using *foreach* as just described should ensure that the streams on each worker are are distinct, it does not ensure reproducibility because task chunks may be assigned to workers differently in different runs and the substreams are specific to workers, not to tasks.

For *doMPI*, you can specify a different RNG substream for each task chunk in a way that ensures reproducibility. Basically you provide a list called *.options.mpi* as an argument to *foreach*, with *seed* as an element of the list:

```
nslaves <- 2
library(doMPI, quietly = TRUE)
cl <- startMPIcluster(nslaves)

##  2 slaves are spawned successfully. 0 failed.

registerDoMPI(cl)
result <- foreach(i = 1:20, .options.mpi = list(seed = 0)) %dopar% {
        out <- mean(rnorm(1000))
}

## Warning:  closing unused connection 6 (<-localhost:11286)
## Warning:  closing unused connection 5 (<-localhost:11286)

result2 <- foreach(i = 1:20, .options.mpi = list(seed = 0)) %dopar% {
        out <- mean(rnorm(1000))
}
identical(result, result2)

## [1] TRUE
```

That single seed then initializes the RNG for the first task, and subsequent tasks get separate substreams, using the L'Ecuyer algorithm, based on *nextRNGStream()*. Note that the *doMPI* developers also suggest using the *chunkSize* option (also specified as an element of *.options.mpi*) when using *seed*. See `?"doMPI-package"` for more details.

For other backends, such as *doParallel*, there is a package called *doRNG* that ensures that *foreach* loops are reproducible. Here's how you do it:

```
rm(result, result2)
nCores <- 2
library(doRNG, quietly = TRUE)

## Loading required package:  methods
## Loading required package:  pkgmaker
## Loading required package:  registry

library(doParallel)
registerDoParallel(nCores)
result <- foreach(i = 1:20, .options.RNG = 0) %dorng% {
```

```
        out <- mean(rnorm(1000))
}
result2 <- foreach(i = 1:20, .options.RNG = 0) %dorng% {
        out <- mean(rnorm(1000))
}
identical(result, result2)

## [1] TRUE
```

Alternatively, you can do:

```
rm(result, result2)
library(doRNG, quietly = TRUE)
library(doParallel)
registerDoParallel(nCores)
registerDoRNG(seed = 0)
result <- foreach(i = 1:20) %dopar% {
        out <- mean(rnorm(1000))
}
registerDoRNG(seed = 0)
result2 <- foreach(i = 1:20) %dopar% {
        out <- mean(rnorm(1000))
}
identical(result,result2)

## [1] TRUE
```