Parallel Processing here at the School of Statistics

Charles J. Geyer
School of Statistics
University of Minnesota

http://www.stat.umn.edu/~charlie/parallel/

- batch processing
- R package `multicore`
- R package `rlecuyer`
- R package `snow`
- grid engine (CLA)
- clusters (MSI)

Batch Processing

This is really old stuff (from 1975). But not everyone knows it.

If you do the following at a unix prompt

`nohup nice -n 19` *some job* `&`

where "*some job*" is replaced by an actual job, then

- the job will run in background (because of `&`).
- the job will not be killed when you log out (because of `nohup`).
- the job will have low priority (because of `nice -n 19`).

Batch Processing (cont.)

For example, if `foo.R` is a plain text file containing R commands, then

```
nohup nice -n 19 R CMD BATCH --vanilla foo.R &
```

executes the commands and puts the printout in the file `foo.Rout`. And

```
nohup nice -n 19 R CMD BATCH --no-restore foo.R &
```

executes the commands, puts the printout in the file `foo.Rout`, and saves all created R objects in the file `.RData`.

Batch Processing (cont.)

```
nohup nice -n 19 R CMD BATCH foo.R &
```

is a *really bad idea!* It reads in all the objects in the file `.RData` (if one is present) at the beginning. So you have no idea whether the results are reproducible.

Always use `--vanilla` or `--no-restore` except when debugging.

Batch Processing (cont.)

This idiom has nothing to do with R. If `foo` is a compiled C or C++ or Fortran main program that doesn't have command line arguments (or a shell, Perl, Python, or Ruby script), then

```
nohup nice -n 19 foo &
```

runs it. And

```
nohup nice -n 19 foo < foo.in > foo.out &
```

runs it taking input from the file `foo.in` and placing output in the file `foo.out`. Regular output and error messages are interspersed and not necessarily in order.

```
nohup nice -n 19 foo < foo.in > foo.out 2> foo.err &
```

puts the error messages in a separate file.

Batch Processing (cont.)

Don't omit the `nice -n 19`. If you omit it, and we notice it, you'll be in trouble. Or if we got up on the wrong side of bed that morning, we'll just kill your jobs.

Batch Processing (cont.)

We've got lots of computers, and each one has eight processors (so eight jobs can run simultaneously).

That allows a lot of parallel processing without knowing anything more than how to background a job.

Multicore Package

Now we start talking about parallel processing within one R job using the `multicore` package (available from CRAN) and installed on our computers. If you do the following in R

```
library(multicore)
setup statements
for (isplit in 1:nsplit) {
    parallel(some R expression involving isplit)
}
out <- collect()
```

then the expressions that are arguments to `parallel` will execute in parallel and `out` will be a list of length `nsplit` containing the results.

Multicore Package (cont.)

Each invocation of the `parallel` makes a copy of the entire R process using the unix fork command and runs it independently of the main R process.

Call the main R process the "parent" process and the forked copies the "child" processes (unix terminology).

The parent process keeps going when a child process is forked, eventually spawning `nsplit` child processes running in parallel. When the `collect` function is invoked, the parent process waits until all the child processes finish and collects their results into a list, which is the value of the `collect` function.

Multicore Package (cont.)

There is no special setup for the child processes, the unix fork command in effect copies the whole state of the process (it actually uses copy-on-write, no copying is done until the value of an object changes). No child process has any effect on the parent process except via the `collect` function.

Multicore Package (cont.)

If that's too complicated, you can just think it's magic, and it really works!

There are options to `parallel` and `collect` that allow waiting for some but not all child processes.

You can start as many child processes as you want, but there's no point in having more than 8 running simultaneously (or however many processors you have in the computer if you are not using our linux workstations).

Multicore Package (cont.)

There is an alternative to `parallel` and `collect` as a way to make child processes and gets results from them (from the R help page)

```
mclapply(X, FUN, ..., mc.preschedule = TRUE,
    mc.set.seed = TRUE, mc.silent = FALSE,
    mc.cores = getOption("cores"))
```

Multicore Package (cont.)

Warning! Child processes all compete for the memory in the computer. If they are all trying to use most of the memory in the computer, there will be no speed up from parallel processing. In fact the computer may slow to unusability.

If the problem is compute cycles, `multicore` works great.

If the problem is memory, `multicore` is no help!

Multicore Package (cont.)

Note on the help for the `parallel` function

> Windows operating system lacks the `fork` system call so it cannot be used with `multicore`.

Multicore Package (cont.)

The optional argument `mc.set.seed = TRUE` given to the `parallel` function will make each child process use a different random number stream.

But the method used,

```
set.seed(Sys.getpid())
```

invoked in each child process, has no theoretical guarantees.

Rlecuyer Package

The `rlecuyer` package (available from CRAN) makes available multiple parallel random number streams that do have theoretical guarantees.

```
library(rlecuyer)
.lec.SetPackageSeed(c(42, 66, 101, 123454, 7, 54321))
nstream <- 8
stream.names <- LETTERS[1:nstream]
.lec.CreateStream(stream.names)
```

creates `nstream` streams named `"A"` through `"H"`

Rlecuyer Package (cont.)

With that setup

```
library(multicore)
foo <- function(name) {
     .lec.CurrentStream(name)
     result <- runif(5)
     .lec.CurrentStreamEnd()
     return(result)
}
for (i in 1:nstream)
    parallel(foo(stream.names[i]), name = stream.names[i])
out <- collect()
```

produces different random vectors in each component of `out`

Rlecuyer Package (cont.)

Random number seeds to not propagate back to parent process when using `multicore`.

No surprise. *Nothing* propagates back except what is returned in the list returned by the `collect` function.

So running the loop again will produce the *same* random numbers as before (probably not what is wanted).

Snow Package

The `snow` package (Simple Network of Workstations, available from CRAN) provides parallel processing via clusters of processes running on different computers.

Snow Package (cont.)

```
library(snow)
machines <- c("hyland", "dogleg", "crab", "sugar",
    "strike", "pool", "pool", "pool")
cl <- makeSOCKcluster(machines)
invisible(clusterEvalQ(cl, library(nice)))
invisible(clusterEvalQ(cl, set.my.priority(19)))
```

some R statements doing some work on the cluster

```
stopCluster(cl)
```

sets up a cluster of eight processes on five machines (three on `pool`), nices all eight processes, and stops the cluster.

Don't forget the `stopCluster` command. It cleans up after you!

Snow Package (cont.)

If your work involves random numbers, then *some R statements doing some work on the cluster* on the preceding slide expands as follows

```
library(rlecuyer)
clusterSetupRNG(cl)
```
some R statements doing some work on the cluster

Snow Package (cont.)

The cluster consists of eight processes (the *slave* processes) on five different machines. They are all controlled by the *master* process which is executing the statements shown here.

Snow Package (cont.)

The main tool for getting work done on the cluster is the function `clusterEvalQ`, which we have already seen used to nice the slave processes. This function evaluates the same R expression on each slave and returns the results to the master process as a list with one component for each slave. For example,

```
out1 <- clusterEvalQ(cl, runif(5))
out2 <- clusterEvalQ(cl, runif(5))
```

Makes two different lists of vectors of uniform random numbers, all of which are different. The ones done on different slaves are different because `clusterSetupRNG` sets up a different stream of random numbers on each slave, and the ones in different calls (`out1` and `out2`) are different because the slaves persist until `stopCluster` is called.

Snow Package (cont.)

There are a bunch of other functions used to do work on the cluster (from the R help page)

```
clusterSplit(cl, seq)
clusterCall(cl, fun, ...)
clusterApply(cl, x, fun, ...)
clusterApplyLB(cl, x, fun, ...)
clusterExport(cl, list)
clusterMap(cl, fun, ..., MoreArgs = NULL, RECYCLE = TRUE)
```

Snow Package (cont.)

`snow` solves one problem we had with `multicore`, that it was hard to propagate the state of random number generators back to the master process.

`snow` solves another problem we had with `multicore` which doesn't work well with more child processes than the number of cores in one box.

`snow` solves yet another problem we had with `multicore` with all child processes competing for memory in the same box.

`snow` can use all the cores in all the machines in the department simultaneously. But everyone will get mad at you if you try!

Logging in Without Typing a Password

In order to avoid having to type your password once for each slave process started you need to set up `ssh` so that you can log in without typing your password (this is still completely secure).

Step 1. Use

```
ssh-keygen -t dsa
```

to set up an encryption key. Type a long passphrase that you can remember when asked.

If successful, files

```
~/.ssh/id_dsa
~/.ssh/id_dsa.pub
```

will be created.

Logging in Without Typing a Password (cont.)

Step 2. Copy the file

`~/.ssh/id_dsa.pub`

into

`~/.ssh/authorized_keys`

on all machines you want to log into this authentication method.

For machines inside the department, which all share the same home directory (via NFS)

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

does the job.

Logging in Without Typing a Password (cont.)

Steps 1 and 2 get done once.

Step 3. This gets done each time you want to do `ssh` without a password, once per unix login.

```
ssh-add
```

and type your passphrase. It will be remembered (but not written down anywhere) until you log out of unix.

Now `ssh` won't ask for a password.

Sun Grid Engine (CLA)

CLA has a bunch of servers running the Sun Grid Engine that is supposed to make parallelism via batch processing easier.

It's not clear that it does, and they don't have as many processors available as the School of Statistics does (last time I checked, Glen and I tried it more than a year ago).

Massive Clusters (MSI)

The Minnesota Supercomputing Institute (MSI) located across Northrop mall in Walter Library has some very large clusters, some of which an U of M faculty can apply to get time on. So once your application is working here using `snow` it can be moved to a bigger cluster in Walter. (I haven't tried this.)