An introduction to shared memory parallelism (i.e., threading and multicore) in R, C, Matlab, and Python plus an introduction to the SCF Linux Cluster

November 20, 2013

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

For those not running jobs on the SCF network, the details of job submission and the prescriptions below for controlling the number of cores used by your jobs do not apply, but the material should still serve as an introduction to the basics of parallel programming.

1 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.

1.1 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 R package *Rmpi* implements MPI in R.

However, our cluster is not currently set up to allow for message passing between nodes, so I won't cover MPI or *Rmpi* here, though I do include the syntax for using *foreach* with *Rmpi*, as this is one way to avoid a bug that can occur with *foreach* in the context of using a threaded BLAS, with *Rmpi* usable on a single node.

1.2 Shared memory

For shared memory parallelism, each core is accessing the same memory so there is no need to pass messages. 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 (from R, C, and Matlab)
2. multicore functionality (in R, Matlab and Python)
3. general purpose threaded C programs using openMP

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 Running jobs on the department cluster and the queueing system

Both the SCF (Statistics) and EML (Economics) clusters have 8 nodes, with each node having 32 cores. Most clusters and supercomputers have many more nodes but often fewer cores per node. So we're mainly set up for shared memory parallelism.

To submit a job to the cluster, you need to create a simple shell script file containing the commands you want to run. E.g., the script, named *job.sh*, say, might contain one of the lines that follow, depending on whether you are using R, Matlab, or Python:

- `R CMD BATCH --no-save sim.R sim.Rout`
- `matlab -nodisplay -nodesktop -singleCompThread < sim.m > sim.out`
- `python sim.py > sim.out`

To submit your job:

```
$ qsub job.sh
```

Or to submit to the high priority queue (a user can only use 12 cores at any given time in the high priority queue, but jobs take priority over jobs in the low priority queue).

```
$ qsub -q high.q job.sh
```

I/O intensive jobs For jobs that read or write a lot of data to disk, it's best to read/write from the local hard disk rather than to your home directory on the SCF fileserver.

Here's an example script that illustrates staging your input data to the local hard disk on the cluster node:

```
if [ !  -d /tmp/$USER/ ]; then mkdir /tmp/$USER; fi
cp ~/projectdir/inputData /tmp/$USER # this positions an input file
on the local hard disk of the node that the job is on
R CMD BATCH --no-save sim.R sim.out # your R code should find inputData
in /tmp/$USER and write output to /tmp/$USER
cp /tmp/$USER/outputData ~/projectdir/.
```

Note that you are NOT required to stage your data from the local hard disk. By default files will be read from and written to the directory from which you submitted the job, which will presumably be somewhere in your SCF home directory. If you are reading or writing big files, you may want to stage data to the local disk to increase performance (though if I/O is a small part of your job, it may not matter). And if I/O is a big part of your job (i.e., gigabytes and particularly tens of Gb of I/O), we DO ask that you do do this.

Long-running jobs

For jobs you expect to run more than three days, you need to flag this for the system. (This allows us to better optimize sharing of resources on the cluster.) For jobs in the low queue, you would do

```
qsub -l h_rt=672:00:00 job.sh
```

and for jobs in the high queue:

```
qsub -q high.q -l h_rt=168:00:00 job.sh
```

These set the maximum run time to 28 days (low queue) and 7 days (high queue) respectively, whereas the default is 5 days.

2.1 Submitting jobs: To 'pe' or not to 'pe'

Non-parallel jobs If you have not written any explicit parallel code, you can submit your job as above.

However, we also need to make sure your job does not use more than one core by using multiple threads. By default the system limits any threaded jobs, including calls to the BLAS, to one core by having OMP_NUM_THREADS set by default to one..

However, Matlab is a bit of a renegade and its threading needs to be controlled explicitly by the user. If you are running in Matlab, you are **REQUIRED** to start Matlab with the singleCompThread flag:

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

Alternatively, from within your Matlab code you can do

```
feature('numThreads', 1)
```

Parallel jobs If you want your job to use more than one core (either by using threaded code or by explicitly parallelizing your code), you need to do the following. First, submit your job with the "`-pe smp X`" flag to qsub, where X is the number of cores (between 2 and 32) that you want. E.g.,

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

Next in your code, make sure that the total number of processes multiplied by the number of threads per process does not exceed the number of cores you request, by following the guidelines below. Note that the NSLOTS environment variable is set to the number of cores you have requested via `-pe smp`, so you can make use of NSLOTS in your code.

For jobs other than Matlab jobs, please follow these guidelines:

1. To use **more than one thread within a single process** in R or C code, set OMP_NUM_THREADS to NSLOTS in your script:

   ```
   export OMP_NUM_THREADS=$NSLOTS
   ```

   This will make available as many threads as cores that you have requested. For R jobs, this needs to be set outside of R, in the shell, before running/starting R.

2. To **run multiple processes** via explicit parallelization in your code, but **with a single thread per process**, you need to create only as many processes as you have requested. We'll see

various ways for doing this below when using R, Python or C, and you can make use of NSLOTS in your code. Since OMP_NUM_THREADS defaults to one, only a single thread per process will be used.

3. Finally, to **run multiple processes with more than one thread per process** (in R or C code), you need to make sure that the total number of threads across all processes is no more than the number of cores you have requested. Thus if you want to have H threads per process and your code starts P processes, you should request $H \times P$ cores via `-pe smp` and you should then set OMP_NUM_THREADS to H.

For Matlab jobs,

1. To use **more than one thread within a single process**, you should including the following Matlab code in your script:
   `feature('numThreads', str2num(getenv('NSLOTS')))`
   This will make available as many threads as cores that you have requested. However Matlab at this time uses 16 cores at most when threading, so there is no point in requesting more than that..

2. To **run multiple processes** in a *parfor* job or when using *parfeval()*, you should set the number of workers to $NSLOTS in *parpool()* [R2013b] or *matlabpool()* [R2013a and earlier] (see the next section for template code). By default it appears that Matlab only uses one thread per worker so your job will use no more cores than you have requested. When using *parfeval()* you can use multiple threads by setting the number of threads within the function being called, but you need to ensure that the number of threads multiplied by number of jobs does not exceed the number of cores requested.
   Note: if you're interested in combining threading with *parfor*, email consult@stat.berkeley.edu and we can look into whether there is a way to do this.
   Matlab limits you to at most 12 cores per job with *parpool()/matlabpool()*, so there is no point in requesting any more than that. [Note that on the EML we are in the process of implementing the ability to run up to 32 cores per Matlab job - email consult@econ.berkeley.edu if you have questions.]

Reserving multiple cores If the queue is heavily loaded and you request many slots via the -pe smp syntax, that many slots may not come open all at once for a long time, while jobs requesting fewer cores may slip in front of your job. You can request that the queue reserve (i.e., accumulate) cores for your job, preventing smaller jobs from preempting your job. To do so, include "-R y" in your qsub command, e.g.,

```
$ qsub -pe smp 20 -R y job.sh
```

2.2 Monitoring jobs

You can get information about jobs using *qstat*.

```
$ qstat # shows your jobs
$ qstat -u "*" # shows everyone's jobs
$ qstat -j pid # shows more information about a single job
$ qdel pid # deletes a single job
$ qdel -u username # deletes all your jobs
```

2.3 Interactive jobs

In addition to submitting batch jobs, you can also run a job from the command line on a node of the cluster.

```
qrsh -q interactive.q
```

You can then use the node as you would any other Linux compute server, but please remember to exit the session as soon as you are done so you don't unnecessarily tie up resources. The primary use of this functionality is likely to be prototyping code and debugging. However, note that cluster nodes are set up very similarly to the other Linux compute servers, so most code should run the same on the cluster as on any other SCF Linux machine.

In some cases, you may want to specify a particular node, such as for data transfer to a particular node or to monitor a specific batch job. In this case (here for node *scf-sm01*), do:

```
qrsh -q interactive.q -l hostname=scf-sm01
```

Once on the node you can transfer files via *scp*, use *top* to monitor jobs you have running, etc.

Note that the NSLOTS environment variable is not set in interactive jobs. Please do not use more threads or cores than you requested (one by default unless you do `qrsh -q interactive.q -pe smp X`).

3 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). The default BLAS on the SCF Linux compute servers is openBLAS and on the cluster nodes is ACML. 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.)

3.1 R

On the SCF, R is linked against openBLAS (for the compute servers) and ACML (for the cluster). On the compute servers, linear algebra will be threaded by default, while on the cluster nodes, OMP_NUM_THREADS is set to one by default, so if you want to enable threading, you need to follow the instructions in the previous section of this document. If you're on another system, the R installation manual gives information on how link R to a fast BLAS and you can consult with the system administrator.

```
x <- matrix(rnorm(8000^2), 8000)
system.time({
    x <- crossprod(x)
    U <- chol(x)
})
# exit R, execute: 'export OMP_NUM_THREADS=1', and restart R
system.time({
    x <- crossprod(x)
    U <- chol(x)
})
```

3.2 Matlab

Similarly, Matlab threads certain linear algebra and other calculations by default, so if you're running Matlab and monitoring *top*, you may see a process using more than 100% of CPU. However worker tasks within a *parfor()* use only a single thread.

3.3 C/C++

To use threaded BLAS calls in a function you compile from C/C++, just make your usual BLAS or Lapack calls in your code. Then link against BLAS and Lapack (which on our system will automatically link against openBLAS or ACML). Here's an example C++ program (*testLinAlg.cpp*) and compilation goes like this (the R and Rmath links are because I use R's rnorm function):

```
  g++ -o testLinAlg testLinAlg.cpp -I/usr/share/R/include -llapack
-lblas -lRmath -lR -O3 -Wall
```

If you'd like to link against ACML's BLAS and LAPACK (not necessary on the cluster, where ACML is the default):

```
g++ -o testLinAlg testLinAlg.cpp -I/usr/share/R/include
-L/opt/acml5.2.0/gfortran64_fma4_mp/lib -lacml_mp -lgfortran
-lgomp -lrt -ldl -lm -lRmath -lR -O3 -Wall
```

That program is rather old-fashioned and doesn't take advantage of C++. One can also use the Eigen C++ template library for linear algebra that allows you to avoid the nitty-gritty of calling Lapack Fortran routines from C++. Eigen overloads the standard operators so you can write your code in a more natural way. An example is in *testLinAlgEigen.cpp*. Note that Eigen apparently doesn't take much advantage of multiple threads even when compiled with openMP, but in this basic test it was competitive with openBLAS/ACML when openBLAS/ACML was restricted to one core.

```
g++ -o testLinAlgEigen testLinAlgEigen.cpp -I/usr/include/eigen3
-I/usr/share/R/include -lRmath -lR -O3 -Wall -fopenmp
```

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 variable. 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 bash shell, you'd do this as follows (e.g. to limit to 3 cores) (do this before starting R):

```
export OMP_NUM_THREADS=3 # or "setenv OMP_NUM_THREADS 1" if using
csh/tcsh
```

Alternatively, you can set OMP_NUM_THREADS as you invoke R:

```
OMP_NUM_THREADS=3 R CMD BATCH --no-save job.R job.out
```

OMP_NUM_THREADS is set by default to 1 for jobs submitted to the cluster.

Conflict between openBLAS and some parallel functionality in R

Warning: There is some sort of conflict between forking in R and threaded BLAS that affects *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*). 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

Warning: 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

Warning: 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.

More generally, if you have an embarrassingly parallel job, it is likely to be more effective to use the fixed number of multiple cores you have access to so as to split along the embarrassingly parallel dimension without taking advantage of the threaded BLAS (i.e., restricting each process to a single thread).

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.

4 Parallel programming in R

4.1 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 or (our focus here) 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; we'll discuss this a bit more in the second part of the workshop.

```
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)
library(iterators)

taskFun <- function() {
    mn <- mean(rnorm(1e+07))
    return(mn)
}
nCores <- 8  # manually for non-cluster machines
# nCores <- as.numeric(Sys.getenv('NSLOTS')) # for use on cluster
registerDoParallel(nCores)
# registerDoMC(nCores) # alternative to registerDoParallel cl <-
# startMPIcluster(nCores); registerDoMPI(cl) # when using Rmpi as the
# back-end

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.

4.2 parallel apply, vectorization (parallel package)

The *parallel* package has the ability to parallelize the various *apply()* functions (apply, lapply, sapply, etc.) and parallelize vectorized functions. The *multicore* package also has this ability and *parallel* is built upon *multicore*. *Parallel* is a core R package so we'll explore the functionality in that setting. 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)
nCores <- 8  # manually for non-cluster machines
# nCores <- as.numeric(Sys.getenv('NSLOTS')) # for use on cluster
#
# using sockets
#
# ?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(1e+06))
    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))
system.time(res2 <- sapply(input, testFun))
res <- parLapply(cl, input, testFun)

################################ using forking
```

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

Now let's consider parallel evaluation of a vectorized function. This will often only be worthwhile on very long vectors and for computationally intensive calculations. (The *Matern()* call here is more time-consuming than *exp()*).

```
require(parallel)
nCores <- 8
# nCores <- as.numeric(Sys.getenv('NSLOTS')) # for use on cluster
cl <- makeCluster(nCores)
library(fields)
ds <- runif(6e+06, 0.1, 10)
ds_exp <- pvec(ds, exp, mc.cores = nCores)
# here's a more computationally intensive function
system.time(corVals <- pvec(ds, Matern, 0.1, 2, mc.cores = nCores))
system.time(corVals <- Matern(ds, 0.1, 2))
```

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. If you use such a package, you should submit your job with *-pe smp* and set the number of cores/CPUs used to be $NSLOTS (syntax will vary depending which package is being used).

5 Parallel programming in Matlab

5.1 Threaded operations

Many Matlab functions are automatically threaded, so you don't need to do anything special in your code to take advantage of this. Just make sure to follow the instructions in the previous section about setting the number of threads based on the number of cores you have requested.

5.2 Parallel for loops

To run a loop in parallel in Matlab, you can use the *parfor* construction. Note that once again, this only makes sense if the iterations operate independently of one another. Before running the parfor you need to start up a set of workers using *parpool()* [R2013b] or *matlabpool()* [R2013a and earlier]. If you're doing this on the cluster, you must set the *parpool* size to the number of slots requested via the *-pe smp* flag to your qsub submission. It appears that Matlab only uses one thread per worker, so you can set the pool size to the number of slots requested.

```
NSLOTS = str2num(getenv('NSLOTS')); # if running on the cluster
% NSLOTS = 8; # otherwise choose how many cores you want to use
pool = parpool(NSLOTS); # matlabpool(NSLOTS) in R2013a and earlier
n = 3000
nIts = 500
c = zeros(n, nIts);
parfor i = 1:nIts
     c(:,i) = eig(rand(n));
end
delete(pool);
```

5.3 Manually parallelizing individual tasks

You can also explicitly program parallelization, managing the individual parallelized tasks. Here is some template code for doing this. We'll submit our jobs to a pool of workers so that we have control over how many jobs are running at once. Note that here I submit 6 jobs that call the same function, but the different jobs could call different functions and have varying inputs and outputs. Matlab will run as many jobs as available workers in the pool and will queue the remainder, starting them as workers in the pool become available. (So my example is a bit silly in that I have 8 workers but only 6 jobs.)

```
feature('numThreads', 1);
NSLOTS = str2num(getenv('NSLOTS')); % if running on the cluster
% NSLOTS = 8; % otherwise choose how many cores you want to use
pool = parpool(NSLOTS); % matlabpool(NSLOTS) in R2013a and earlier
% assume you have test.m with a function, test, taking two inputs
% (n and seed) and returning 1 output
n = 10000000;
job = cell(1,6);
job{1} = parfeval(pool, @test, 1, n, 1);
job{2} = parfeval(pool, @test, 1, n, 2);
job{3} = parfeval(pool, @test, 1, n, 3);
job{4} = parfeval(pool, @test, 1, n, 4);
job{5} = parfeval(pool, @test, 1, n, 5);
job{6} = parfeval(pool, @test, 1, n, 6);

% wait for outputs, in order
```

```
output = cell(1, 6);
for idx = 1:6
  output{idx} = fetchOutputs(job{idx});
end

% alternative way to loop over jobs:
for idx = 1:6
  jobs(idx) = parfeval(pool, @test, 1, n, idx);
end

% wait for outputs as they finish
output = cell(1, 6);
for idx = 1:6
  [completedIdx, value] = fetchNext(jobs);
  output{completedIdx} = value;
end

delete(pool);
```

And if you want to run threaded code in a given job, you can do that by setting the number of threads within the function called by *parfeval()*. See the *testThread.m* file in the demo code files.

```
NSLOTS = str2num(getenv('NSLOTS')); % if running on the cluster
% NSLOTS = 8; % otherwise choose how many cores you want to use
pool = parpool(NSLOTS); % matlabpool(NSLOTS) in R2013a and earlier
n = 5000;
nJobs = 2;
pool = parpool(nJobs);
% pass number of threads as number of slots divided by number of jobs
% testThread() function should then do:
% feature('numThreads', nThreads);
% where nThreads is the name of the relevant argument to testThread()
jobt1 = parfeval(pool, @testThread, 1, n, 1, NSLOTS/nJobs);
jobt2 = parfeval(pool, @testThread, 1, n, 2, NSLOTS/nJobs);
jobt3 = parfeval(pool, @testThread, 1, n, 3, NSLOTS/nJobs);
jobt4 = parfeval(pool, @testThread, 1, n, 4, NSLOTS/nJobs);
```

```
output1 = fetchOutputs(jobt1);
output2 = fetchOutputs(jobt2);
output3 = fetchOutputs(jobt3);
output4 = fetchOutputs(jobt4);

delete(pool);
```

Note that functions such as *batch()*, *createJob()*, *createTask()* and *submit()* appear to be designed for jobs that you submit to a queueing system from within Matlab, which is not how our cluster is set up, so don't use these on the cluster without emailing consult@stat.berkeley.edu.

6 Parallel programming in Python

There are a number of approaches to parallelization in Python. We'll cover two of the basic approaches here. Note that Python has something called the Global Interpreter Lock that interferes with threading in Python. The approaches below make use of multiple processes.

6.1 Multiprocessing package

We'll first see how to use the multiprocessing package to do multi-core calculations. First we'll use the *Pool.map()* method to iterate in a parallelized fashion, as the Python analog to *foreach* or *parfor*. *Pool.map()* only supports having a single argument to the function being used, so we'll use list of tuples, and pass each tuple as the argument.

```
import multiprocessing as mp
import numpy as np

# on cluster, when using qsub:
import os
nCores = int(os.environ['NSLOTS'])
# nCores = 8 # otherwise choose how many cores you want to use

nSmp = 100000
m = 40
def f(input):
   np.random.seed(input[0])
   return np.mean(np.random.normal(0, 1, input[1]))
```

```
# create list of tuples to iterate over, since
# Pool.map() does not support multiple arguments
inputs = [(i, nSmp) for i in xrange(m)]
inputs[0:2]
pool = mp.Pool(processes = nCores)
results = pool.map(f, inputs)
print(results)
```

We can also manually dispatch the jobs as follows. However, this method will not manage the processes such that only as many jobs are being done as there are cores, so you would need to manually manage that.

```
# set up a shared object to store results
result_queue = mp.Queue()

def f(i, n, results):
   np.random.seed(i)
   # return both index and result as tuple to show how to do that
   results.put((i, np.mean(np.random.normal(0, 1, n))))

jobs = [] # list of processes
nProc = nCores # don't have more processes than cores available
for i in range(nCores):
   p = mp.Process(target = f, args = (i, nSmp, result_queue))
   jobs.append(p)
   p.start()
# wait until each child finishes
for p in jobs:
   p.join()
results = [result_queue.get() for i in range(nCores)]
print(results)
```

6.2 pp package

Here we create a server object and submit jobs to the server object, which manages the farming out of the tasks. Note that this will run interactively in iPython or as a script from UNIX, but will

not run interactively in the base Python interpreter (for reasons that are unclear to me). Also note that while we are illustrating this as basically another parallelized for loop, the individual jobs can be whatever calculations you want, so the *f()* function could change from job to job.

```
import numpy
import pp

# on cluster, when using qsub:
import os
nCores = int(os.environ['NSLOTS'])

job_server = pp.Server(ncpus = nCores, secret = 'mysecretphrase')
# set 'secret' to some passphrase (you need to set it but
#   what it is should not be crucial)
job_server.get_ncpus()

nSmp = 1000000
m = 40
def f(i, n):
   numpy.random.seed(i)
   return (i, numpy.mean(numpy.random.normal(0, 1, n)))

# create list of tuples to iterate over
inputs = [(i, nSmp) for i in xrange(m)]
# submit and run jobs
jobs = [job_server.submit(f, invalue, modules =
  ('numpy',)) for invalue in inputs]
# collect results (will have to wait for longer tasks to finish)
results = [job() for job in jobs]
print(results)
job_server.destroy()
```

7 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.

7.1 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}.

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 <- 4
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*.

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 <- 4
library(doMPI, quietly = TRUE)
cl <- startMPIcluster(nslaves)
```

```
##  4 slaves are spawned successfully. 0 failed.

registerDoMPI(cl)
result <- foreach(i = 1:20, .options.mpi = list(seed = 0)) %dopar% {
    out <- mean(rnorm(1000))
}
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 <- 4
library(doRNG, quietly = TRUE)

## Loading required package:  registry

library(doParallel)
registerDoParallel(nCores)
result <- foreach(i = 1:20, .options.RNG = 0) %dorng% {
    out <- mean(rnorm(1000))
}

## Warning:  error 'Interrupted system call' in select

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

## Warning:  error 'Interrupted system call' in select
```

```
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))
}

## Warning:  error 'Interrupted system call' in select

registerDoRNG(seed = 0)
result2 <- foreach(i = 1:20) %dopar% {
    out <- mean(rnorm(1000))
}

## Warning:  error 'Interrupted system call' in select

identical(result, result2)

## [1] TRUE
```

7.2 Python

Python uses the Mersenne-Twister generator. If you're using the RNG in *numpy/scipy*, you can set the seed using *{numpy,scipy}.random.seed()*. The advice I'm seeing online in various Python forums is to just set separate seeds, so it appears the Python community is not very sophisticated about this issue. There is a function *random.jumpahead()* that allows you to move the seed ahead as if a given number of random numbers had been generated, but this function will not be in Python 3.x, so I won't suggest using it.

7.3 Matlab

Matlab also uses the Mersenne-Twister. We can set the seed as: `rng(seed)`, with seed being a non-negative integer.

Happily, like R, we can set up independent streams, using either of the Combined Multiple Recursive ('mrg32k3a') and the Multiplicative Lagged Fibonacci ('mlfg6331_64') generators. Here's an example, where we create the second of the 5 streams, as if we were using this code in the second of our parallel processes. The `'Seed',0` part is not actually needed as that is the default.

```
thisStream = 2;
totalNumStreams = 5;
seed = 0;
cmrg1 = RandStream.create('mrg32k3a', 'NumStreams', totalNumStreams,
   'StreamIndices', thisStream, 'Seed', seed);
RandStream.setGlobalStream(cmrg1);
randn(5, 1)
```

8 Explicit parallel programming in R: mcparallel and forking

Before we get into some functionality, let's define some terms more explicitly.

- *threading*: 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
- *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

Now let's discuss some functionality in which one more explicitly controls the parallelization.

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

Using mcparallel to dispatch blocks of code to different processes

First 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 <- 1e+07
system.time({
    p <- mcparallel(mean(rnorm(n)))
    q <- mcparallel(mean(rgamma(n, shape = 1)))
    res <- mccollect(list(p, q))
})
system.time({
    p <- mean(rnorm(n))
    q <- mean(rgamma(n, shape = 1))
})
```

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

Explicitly forking code in R

The *fork* package and *fork()* function in R provide an implementation of the UNIX *fork* system call for forking a process.

```
library(fork)
{
    # this set of braces is REQUIRED, unless you pass a function
    # to the slave argument of fork()
    pid <- fork(slave = NULL)
    if (pid == 0) {
        cat("Starting child process execution.\n")
        tmpChild <- mean(rnorm(1e+07))
        cat("Result is ", tmpChild, "\n", sep = "")
        save(tmpChild, file = "child.RData")
        cat("Finishing child process execution.\n")
        exit()
    } else {
        cat("Starting parent process execution.\n")
        tmpParent <- mean(rnorm(1e+07))
        cat("Finishing parent process execution.\n")
        wait(pid)  # wait til child is finished so can read in
        # updated child.RData below
```

```
    }
}
load("child.RData")
print(c(tmpParent, tmpChild))
```

Note that if we were really running the above code, we'd want to be careful about the RNG. As it stands, it will use the same random numbers in both child and parent processes.

9 OpenMP for C: compilation and parallel for

It's straightforward to write threaded code in C and C++ (as well as Fortran). The basic approach is to use the *openMP* protocol. Here's how one would parallelize a loop in C/C++ using an *openMP* compiler directive. As with *foreach* in R, you only want to do this if the iterations do not depend on each other.

```
// testOpenMP.cpp
#include <iostream>
using namespace std;

// compile with:  g++ -fopenmp -L/usr/local/lib
//                  testOpenMP.cpp -o testOpenMP

int main(){
  int nReps = 20;
  double x[nReps];
  #pragma omp parallel for
  for (int i=0; i<nReps; i++){
    x[i] = 0.0;
    for ( int j=0; j<1000000000; j++){
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

We would compile this program as follows

```
$ g++ -L/usr/local/lib -fopenmp testOpenMP.cpp -o testOpenMP
```

The main thing to be aware of in using *openMP* is not having different threads overwrite variables used by other threads. In the example above, variables declared within the `#pragma` directive will be recognized as variables that are private to each thread. In fact, you could declare 'int i' before the compiler directive and things would be fine because *openMP* is smart enough to deal properly with the primary looping variable. But big problems would ensue if you had instead written the following code:

```
int main(){
  int nReps = 20;
  int j;  // DON'T DO THIS !!!!!!!!!!!!
  double x[nReps];
  #pragma omp parallel for
  for (int i=0; i<nReps; i++){
    x[i] = 0.0;
    for (j=0; j<1000000000; j++){
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

Note that we do want x declared before the compiler directive because we want all the threads to write to a common x (but, importantly, to different components of x). That's the point!

We can also be explicit about what is shared and what is private to each thread:

```
int main(){
  int nReps = 20;
  int i, j;
  double x[nReps];
  #pragma omp parallel for private(i,j) shared(x, nReps)
  for (i=0; i<nReps; i++){
    x[i] = 0.0;
    for (j=0; j<1000000000; j++){
```

```
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

As discussed, when running on the cluster, you are required to use the *-pe smp* parallel environment flag and to set OMP_NUM_THREADS to NSLOTS .

10 OpenMP for C/C++: more advanced topics

The goal here is just to give you a sense of what is possible with *openMP*.

The OpenMP API provides three components: compiler directives that parallelize your code (such as `#pragma omp parallel for`), library functions (such as *omp_get_thread_num()*), and environment variables (such as OMP_NUM_THREADS)

OpenMP constructs apply to structured blocks of code.

Here's a basic "Hello, world" example that illustrates how it works:

```
// helloWorldOpenMP.cpp
#include <stdio.h>
#include <omp.h> // needed when using any openMP functions
//                                like omp_get_thread_num()

void myFun(double *in, int id){
// this is the function that would presumably do the heavy lifting
}

int main()
{
   int nthreads, myID;
   double* input;
   /* make the values of nthreads and myid private to each thread */
   #pragma omp parallel private (nthreads, myID)
   { // beginning of block
      myID = omp_get_thread_num();
      printf("Hello, I am thread %d\n", myID);
```

```
      myFun(input, myID);  // do some computation on each thread
      /* only master node print the number of threads */
      if (myid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   } // end of block
   return 0;
}
```

The parallel directive starts a team of threads, including the master, which is a member of the team and has thread number 0. The number of threads is determined in the following ways - here the first two options specify four threads:

1. `#pragma omp parallel NUM_THREADS (4)` // set 4 threads for this parallel block
2. `omp_set_num_threads(4)` // set four threads in general
3. the value of the OMP_NUM_THREADS environment variable
4. a default - usually the number of cores on the compute node

Note that in `#pragma omp parallel for`, there are actually two instructions, 'parallel' starts a team of threads, and 'for' farms out the iterations to the team. In our *parallel for* invocation, we have done it more explicitly as:

```
#pragma omp parallel
#pragma omp for
```

We can also explicitly distribute different chunks of code amongst different threads:

```
// sectionsOpenMP.cpp
#pragma omp parallel // starts a new team of threads
{
   Work0(); // this function would be run by all threads.
   #pragma omp sections // divides the team into sections
   {
      // everything herein is run only once.
      #pragma omp section
      { Work1(); }
```

```
      #pragma omp section
      {
         Work2();
         Work3();
      }
      #pragma omp section
      { Work4(); }
   }
} // implied barrier
```

Here Work1, {Work2 + Work3} and Work4 are done in parallel, but Work2 and Work3 are done in sequence (on a single thread).

If one wants to make sure that all of a parallized calculation is complete before any further code is executed you can insert

```
#pragma omp barrier
```

Note that a `#pragma for` statement includes an implicit barrier as does the end of any block specified with `#pragma omp parallel`

You can use '`nowait`' if you explicitly want to prevent threads from waiting at an implicit barrier: e.g., `#pragma omp parallel sections nowait` or `#pragma omp parallel for nowait`

One should be careful about multiple threads writing to the same variable at the same time (this is an example of a *race condition*). In the example below, if one doesn't have the `#pragma omp critical` directive two threads could read the current value of '`sum`' at the same time and then sequentially write to sum after incrementing their local copy, which would result in one of the increments being lost. A way to avoid this is with the `critical` directive (for single lines of code you can also use `atomic` instead of `critical`):

```
// see criticalOpenMP.cpp
double sum = 0.0;
double tmp;
#pragma omp parallel for private (tmp, i) shared (sum)
for (int i=0; i<n; i++){
   tmp = myFun(i);
   #pragma omp critical
   sum += tmp;
}
```