An Introduction to Parallel Processing in R, Python, C, and Matlab

Chris Paciorek
Department of Statistics
University of California, Berkeley

February 4, 2015

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 some demos on a few Statistics Ubuntu 14.04 machines. Some of this can be replicated in the BCE Virtual Machine (see *setupUbuntu.sh* for how to do some setup for this), though one can't directly see multi-node computations in that context.

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 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.

1.2 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 (from R, Python, C, and Matlab)
2. simple parallelization of embarrassingly parallel computations (in R, Python, and Matlab)
3. using openMP for shared memory (threaded) processing (in C)

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 ...
```

1.3 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 (in R and Matlab)
2. using MPI for explicit distributed memory processing (in C, R and Python)

1.4 Other type of parallel processing

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

1.4.1 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.

1.4.2 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.

2 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 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.
- 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.

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*). 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.)

3.1 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.

3.2 Using threading in R, Matlab and C/C++

3.2.1 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)
# 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
##  11.978   3.956   4.031

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

##    user  system elapsed
##   6.715   0.222   6.745
```

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

```
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
```

3.2.2 Python

On Ubuntu, if you install openBLAS, the */etc/alternatives* system will set */usr/lib/libblas.so* to point to openBLAS, and Python should use that. On my Mac, Python be default appears to use the vecLib threaded BLAS by default.

I'm not sure about other Linux variants nor about the details of linking Python to MKL or ACML. You may need to build Python yourself against a specific BLAS.

3.2.3 Matlab

Many Matlab functions are automatically threaded (not just linear algebra [which uses MKL]), so you don't need to do anything special in your code to take advantage of this. 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.

3.2.4 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 a threaded BLAS. 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). In this case, I'm assuming the system BLAS is a threaded BLAS (e.g., when openBLAS is installed on an Ubuntu machine):

```
  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, here's example compilation syntax:

```
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
```

Many more details are available from my workshop on C++ and R packages.

3.3 Important warnings about use of threaded BLAS

3.3.1 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.

3.3.2 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.

3.3.3 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.

4 Basic shared memory parallel programming in R, Python, and Matlab

4.1 Basic shared memory parallel programming in R

4.1.1 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 <- 4
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 7).

4.1.2 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.

```r
require(parallel)

## Loading required package:  parallel

nCores <- 4

############################
# 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(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   2.408

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

##    user  system elapsed
##   5.901   0.000   5.891

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

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

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

##    user  system elapsed
##   6.405   0.036   2.432
```

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.

4.1.3 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
##   3.592   0.040   1.968

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

##    user  system elapsed
##   3.238   0.008   3.244
```

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.

4.2 Basic shared memory 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.

4.2.1 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

nCores = 4

nSmp = 10000000
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, out):
    np.random.seed(i)
    # return both index and result as tuple to show how to do that
    out.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 for results...
for p in jobs:
    p.join()

results = [result_queue.get() for i in range(nCores)]


print(results)
```

4.2.2 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
```

```
nCores = 4

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 = 10000000
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 i
# 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()
```

4.3 Basic shared memory parallel programming in Matlab

4.3.1 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()*. Here is some demo code, also available in *demoParfor.m*.

```
nCores = 4;
pool = parpool(nCores);
n = 3000
```

```
nIts = 500
c = zeros(n, nIts);
parfor i = 1:nIts
     c(:,i) = eig(rand(n));
end
delete(pool);
```

4.3.2 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. Here is some demo code, also available in *demoBatch.m*.

```
feature('numThreads', 1);
nCores = 4;
pool = parpool(nCores);
% 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);
```

Functions such as *batch()*, *createJob()*, *createTask()* and *submit()* appear to be designed for jobs that you submit to a queueing system from within Matlab.

5 OpenMP for threaded C code

5.1 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.

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

// compile with:  g++ -fopenmp 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++ -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;
}
```

5.2 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:

```
// see 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("I'm the boss and control %i threads. How come they're in f
      }
   } // 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 could have done it more explicitly as:

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

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

```
// see 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 *result* 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 result = 0.0;
double tmp;
#pragma omp parallel for private (tmp, i) shared (result)
```

```
for (int i=0; i<n; i++){
   tmp = myFun(i);
   #pragma omp critical
   result += tmp;
}
```

You should also be able to use syntax like the following for the parallel for declaration (in which case you shouldn't need the `#pragma omp critical`):

```
#pragma omp parallel for reduction(+:result)
```

I believe that doing this sort of calculation where multiple threads write to the same variable may be rather inefficient given time lost in waiting to have access to *result*, but presumably this would depend on how much time is spent in *myFun()* relative to the reduction operation.

6 Distributed memory

6.1 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.

6.2 Using message passing on a single node

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 MPI example

There are C (*mpicc*) and C++ (*mpic++*, *mpicxx*, *mpiCC* are synonyms) compilers for MPI programs.

Here's a basic hello world example (I'll use the MPI C++ compiler even though the code is all plain C code).

```
// see mpiHello.c
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main(int argc, char** argv) {
int myrank, nprocs, namelen;
char process_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Get_processor_name(process_name, &namelen);
printf("Hello from process %d of %d on %s\n",
myrank, nprocs, process_name);
    MPI_Finalize();
return 0;
}
```

To compile and run the code, do:

```
mpicxx mpiHello.c -o mpiHello
mpirun -np 4 mpiHello
```

This will run multiple (four in this case) processes on the machine on which it is run. Note that *mpirun* just dumbly invokes as many processes as you request without regard to the hardware (i.e., the number of cores) on your machine. Here's the output:

```
Hello from process 2 of 4 on arwen
Hello from process 3 of 4 on arwen
Hello from process 0 of 4 on arwen
Hello from process 1 of 4 on arwen
```

Note that *mpirun*, *mpiexec*, and *orterun* are synonyms under *openMPI*.

To actually write real MPI code, you'll need to go learn some of the MPI syntax. See *quad_mpi.c* and *quad_mpi.cpp*, which are example C and C++ programs (for approximating an integral via quadrature) that show some of the basic MPI functions. Compilation and running are as above:

```
mpicxx quad_mpi.cpp -o quad_mpi
mpirun -np 4 quad_mpi
```

And here's the output:

```
03 January 2013 04:02:47 PM

QUAD_MPI
C++/MPI version
Estimate an integral of f(x) from A to B.
f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )

A = 0
B = 10
N = 999999999
EXACT =       0.4993633810764567
Use MPI to divide the computation among 4 total processes,
of which one is the master and does not do core computations.
Process 2 contributed MY_TOTAL = 0.00095491
Process 1 contributed MY_TOTAL = 0.49809
Process 3 contributed MY_TOTAL = 0.000318308

Estimate =       0.4993634591634721
Error = 7.808701535383378e-08
Time = 9.904016971588135

QUAD_MPI:
Normal end of execution.
```

```
03 January 2013 04:02:56 PM
```

6.2.2 Using R in a distributed memory context

Rmpi example 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.

Here's some example code that uses actual *Rmpi* syntax (as opposed to *foreach* with Rmpi as the back-end, which we'll see in a moment), which is very similar to the MPI C syntax we've already seen. 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.

```
# example syntax of standard MPI functions

library(Rmpi)
mpi.spawn.Rslaves(nslaves = 4)

##  4 slaves are spawned successfully. 0 failed.
## master (rank 0, comm 1) of size 5 is running on: smeagol
## slave1 (rank 1, comm 1) of size 5 is running on: smeagol
## slave2 (rank 2, comm 1) of size 5 is running on: smeagol
## slave3 (rank 3, comm 1) of size 5 is running on: smeagol
## slave4 (rank 4, comm 1) of size 5 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"
##
## $slave4
## [1] "id" "n"  "x"

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

## $slave1
## [1] -2.747
##
## $slave2
## [1] 3.284 3.794
##
## $slave3
## [1]  1.2423 -1.0916 -0.1543
##
## $slave4
## [1]  0.7748 -1.1158  1.5020 -0.4823

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

## $slave3
## [1] 0
##
## $slave4
## [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>
##
## $slave4
## [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
##
## $slave4
##          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] -1.374
##
## [[2]]
## [1] 1.642 1.897
##
## [[3]]
## [1]  0.62114 -0.54582 -0.07713
##
## [[4]]
## [1]  0.3874 -0.5579  0.7510 -0.2412
```

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.

pbdR 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. I haven't had time to pull together any example code, but 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.

6.2.3 Python mpi4py

To use MPI with Python you start Python with *mpirun*:

```
mpirun -np 4 python file.py
```

I don't have much experience with *mpi4py* and was having some bugs in my basic code, so I don't have an example at the moment, but you can send, receive, broadcast, gather, etc. as with MPI itself.

mpi4py generally does not work interactively.

6.3 Using message passing in a distributed memory environment (multiple nodes)

To run on multiple machines, we need to let *mpirun* know the names of those machines. We can do this in two different ways.

First, we can pass the machine names directly, replicating the name if we want multiple processes on a single machine.

```
mpirun --host arwen,arwen,radagast -np 3 mpiHello
```

Here's the output:

```
Hello from processor 0 of 3 on arwen
Hello from processor 2 of 3 on radagast
Hello from processor 1 of 3 on arwen
```

```
mpirun --host arwen,arwen,radagast -np 3 quad_mpi
```

Alternatively, we can create a file with the relevant information. Here we'll specify two processes on *arwen* for every one process on *radagast*.

```
echo 'arwen slots=2' >> .hosts
echo 'radagast slots=1' >> .hosts
mpirun -machinefile .hosts -np 3 mpiHello
mpirun -machinefile .hosts -np 12 mpiHello # this seems to just
recycle what is in .hosts
```

Here's the output from the last command that requested 12 processes:

```
Hello from processor 1 of 12 on arwen
Hello from processor 9 of 12 on arwen
Hello from processor 10 of 12 on arwen
Hello from processor 3 of 12 on arwen
Hello from processor 0 of 12 on arwen
Hello from processor 6 of 12 on arwen
Hello from processor 7 of 12 on arwen
Hello from processor 4 of 12 on arwen
Hello from processor 5 of 12 on radagast
Hello from processor 8 of 12 on radagast
Hello from processor 11 of 12 on radagast
Hello from processor 2 of 12 on radagast
```

To limit the number of threads for each process, we can tell *mpirun* to export the value of OMP_NUM_THREADS to the processes.

```
export OMP_NUM_THREADS=2
mpirun -machinefile .hosts -np 3 -x OMP_NUM_THREADS quad_mpi
```

In these examples, I illustrated with a C program, but one would similarly use the -machinefile flag when starting R or Python.

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

6.4.1 foreach in R

Just as we used *foreach* in a shared memory context, we can use it in a distributed memory context as well, and R will handle all everything behind the scenes for you.

Start R through the *mpirun* command as discussed above, either as a batch job or for interactive use:

```
mpirun -machinefile .hosts -np 1 R CMD BATCH --no-save file.R file.out
mpirun -machinefile .hosts -np 1 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 number of hosts listed in `.hosts`, so your R code will be more portable. The reason for the `-np 1` flag in the *mpirun* invocation is that the worker processes are started within the R code rather than by *mpirun*.

```
## invoke R as:
## mpirun -machinefile .hosts -np 1 R CMD BATCH --no-save file.R file.out

library(Rmpi)
library(doMPI)

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


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

results <- foreach(i = 1:200) %dopar% {
  out = mean(rnorm(1e7))
}

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.

If you specified -np with more than one process then as with the C-based MPI job above, you can control the threading via OMP_NUM_THREADS and the -x flag to *mpirun*. Note that this only works when the R processes are directly started by *mpirun*, which they are not if you set -np 1. The *maxcores* argument to *startMPIcluster()* does not seem to function (perhaps it does on other systems).

Using foreach + doMPI on a single node One reason to use Rmpi on a single node is that for machines using openBLAS (all of the compute servers except for the cluster nodes), there is a conflict between the *multicore*/*parallel*'s forking functionality and openBLAS that can cause *foreach* to hang when used with the *doParallel* or *doMC* parallel back ends.

6.4.2 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("arwen.berkeley.edu", 4),
    rep("treebeard.berkeley.edu", 2),
    rep("beren.berkeley.edu", 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
```

6.4.3 Python

Start Python through the *mpirun* command as discussed above:

```
mpirun -machinefile .hosts -np 4 python file.py > file.out
```

```
## put the following code in file.py and run:
## mpirun -np 4 python file.py

from mpi4py import MPI
```

```
import numpy as np

comm = MPI.COMM_WORLD

# simple print out Rank & Size
id = comm.Get_rank()
print "Of ", comm.Get_size() , " workers, I am number " , id, "."

def f(id, n):
    np.random.seed(id)
    return np.mean(np.random.normal(0, 1, n))

n = 1000000
result = f(id, n)


output = comm.gather(result, root = 0)

if id == 0:
    print output
```

6.4.4 Matlab

To use Matlab across multiple nodes, you need to have the Matlab Distributed Computing Server (DCS). In this case one can set up Matlab so that parfor will distribute its work across multiple nodes.

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}.

7.1.1 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 <- 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*.

7.1.2 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 <- 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:  methods
## Loading required package:  pkgmaker
## Loading required package:  registry

library(doParallel)

## Loading required package:  parallel

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
```

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 that it doesn't have the functionality that R and Matlab do with regard to ensuring independent streams. 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)
```