Unit 8: Parallel Processing

October 6, 2018

References:

- Tutorial on basic parallel processing: https://github.com/berkeley-scf/tutorial-parallel-basics
- Tutorial on distributed parallel processing: https://github.com/berkeley-scf/tutorial-parallel-distributed.

This unit will be fairly Linux-focused as most serious parallel computation is done on systems where some variant of Linux is running. The single-machine parallelization discussed here should work on Macs, but only some of the approaches are likely to work on Windows machines.

1 Some scenarios for parallelization

- You need to fit a single statistical/machine learning model, such as a random forest or regression model, to your data.
- You need to fit three different statistical/machine learning models to your data.
- You are running a prediction method on 10 cross-validation folds, possibly using multiple statistical/machine learning models to do prediction.
- You are running an ensemble prediction method such as *SuperLearner* or *Bayesian model averaging* over 10 cross-validation folds, with 30 statistical/machine learning methods used for each fold.
- You are running stratified analyses on a very large dataset (e.g., running regression models once for each subgroup within a dataset).
- You are running a simulation study with n=1000 replicates. Each replicate involves fitting 10 statistical/machine learning methods.

Given you are in such a situation, can you do things in parallel? Can you do it on your laptop or a single computer? Will it be useful (e.g., faster or provide access to sufficient memory) to use multiple computers, such as multiple nodes in a Linux cluster?

All of the functionality discussed in this Unit applies ONLY if the iterations/loops of your calculations can be done completely separately and do not depend on one another; i.e., you can do the computation as separate processes without communication between the processes. This scenario is called an *embarrassingly parallel* computation

1.1 Embarrassingly parallel (EP) problems

An EP problem is one that can be solved by doing independent computations as separate processes without communication between the processes. You can get the answer by doing separate tasks and then collecting the results. Examples in statistics include

1. simulations with many independent replicates
2. bootstrapping
3. stratified analyses
4. random forests
5. cross-validation.

The standard setup is that we have the same code running on different datasets. (Note that different processes may need different random number streams, as we will discuss in the Simulation Unit.)

To do parallel processing in this context, you need to have control of multiple processes. Note that on a shared system with queueing/scheduling software set up, this will generally mean requesting access to a certain number of processors and then running your job in such a way that you use multiple processors.

In general, except for some modest overhead, an EP problem can ideally be solved with $1/p$ the amount of time for the non-parallel implementation, given p cores. This gives us a speedup of p, which is called linear speedup (basically anytime the speedup is of the form kp for some constant k).

2 Overview of parallel processing

2.1 Computer architecture

Computers now come with multiple processors for doing computation. Basically, physical constraints have made it harder to keep increasing the speed of individual processors, so the chip industry is now putting multiple processing units in a given computer and trying/hoping to rely on implementing computations in a way that takes advantage of the multiple processors.

Everyday personal computers usually have more than one processor (more than one chip) and on a given processor, often have more than one core (multi-core). A multi-core processor has multiple processors on a single computer chip. On personal computers, all the processors and cores share the same memory.

Supercomputers and computer clusters generally have tens, hundreds, or thousands of 'nodes', linked by a fast local network. Each node is essentially a computer with its own processor(s) and memory. Memory is local to each node (distributed memory). One basic principle is that communication between a processor and its memory is much faster than communication between processors with different memory. An example of a modern supercomputer is the Edison supercomputer at Lawrence Berkeley National Lab, which has 5,586 nodes, each with two processors and each processor with 12 cores, giving 134,064 total processing cores. Each node has 64 GB of memory for a total of 357 TB of memory.

There is little practical distinction between multi-processor and multi-core situations. The main issue is whether processes share memory or not. In general, I won't distinguish between cores and processors. We'll just focus on the number of cores on given personal computer or a given node in a cluster.

2.2 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*: instances of a program(s) executing on a machine; multiple processes may be executing at once. A given program may start up multiple processes at once. Ideally we have no more processes than cores on a node.
- *tasks*: individual units of computation; one or more tasks might be executed by a given process on a given core.

- *threads*: 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 the same number of cores as we have processes and threads combined.
- *forking*: child processes are spawned that are identical to the parent, but with different process IDs 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.
- *scheduler*: a program that manages users' jobs on a cluster.
- *load-balanced*: when all the cores that are part of a computation are busy for the entire period of time the computation is running.

2.3 Distributed vs. shared memory

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 for the rest of this document) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. You can think of each node as a separate computer connected by a fast network.

2.3.1 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.

We'll cover two types of shared memory parallelism approaches in this unit:

- threaded linear algebra
- multicore functionality

Threading Threads are multiple paths of execution within a single process. If you are monitoring CPU usage (such as with *top* in Linux or Mac) and watching 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.

Note that this is a different notion than a processor that is hyperthreaded. With hyperthreading a single core appears as two cores to the operating system.

2.3.2 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. The *pbdR* packages for R also implement MPI as well as distributed linear algebra (linear algebra calculations across nodes). In addition, there are various ways to do simple parallelization of multiple computational tasks (across multiple nodes) that use MPI and other tools on the back-end without users needing to understand them. We'll only cover distributed memory parallelization very briefly in this Unit, but we covered a flavor of it via Spark in Unit 7.

2.4 Some other approaches to parallel processing

2.4.1 GPUs

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

Most researchers don't program for a GPU directly but rather use software (often machine learning software such as Tensorflow, PyTorch, or Caffe) that has been programmed to take advantage of a GPU if one is available.

2.4.2 Spark and Hadoop

Spark and Hadoop are systems for implementing computations in a distributed memory environment, using the MapReduce approach, as we saw.

2.4.3 Cloud computing

Amazon (Amazon Web Services' EC2 service), Google (Google Cloud Platform's Compute Engine service) and Microsoft (Azure) offer computing through the cloud. The basic idea is that they rent out their servers on a pay-as-you-go basis. You get access to a virtual machine that can run various versions of Linux or Microsoft Windows server and where you choose the number of processing cores you want. You configure the virtual machine with the applications, libraries, and data you need and then treat the virtual machine as if it were a physical machine that you log into as usual. You can also assemble multiple virtual machines into your own virtual cluster and use platforms such as Spark on the cloud provider's virtual machines.

3 Parallelization strategies

Some of the considerations that apply when thinking about how effective a given parallelization approach will be include:

- the amount of memory that will be used by the various processes,
- the amount of communication that needs to happen – how much data will need to be passed between processes,
- the latency of any communication - how much delay/lag is there in sending data between processes or starting up a worker process, and
- to what extent do processes have to wait for other processes to finish before they can do their next step.

The following are some basic principles/suggestions for how to parallelize your computation.

- Should I use one machine/node or many machines/nodes?
  - If you can do your computation on the cores of a single node using shared memory, that will be faster than using the same number of cores (or even somewhat more cores) across multiple nodes. Similarly, jobs with a lot of data/high memory requirements that one might think of as requiring Spark or Hadoop may in some cases be much faster if you can find a single machine with a lot of memory.
  - That said, if you would run out of memory on a single node, then you'll need to use distributed memory.
- What level or dimension should I parallelize over?
  - 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.

- How do I balance communication overhead with keeping my cores busy?

  - 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.

- Should multiple tasks be pre-assigned to a process (i.e., a worker) (sometimes called *prescheduling*) or should tasks be assigned dynamically as previous tasks finish?

  - Basically if you have many tasks that each take similar time, you want to preschedule the tasks to reduce communication. If you have few tasks or tasks with highly variable completion times, you don't want to preschedule, to improve load-balancing.
  - For R in particular, some of R's parallel functions allow you to say whether the tasks should be prescheduled. E.g., the *mc.preschedule* argument in *mclapply()*. For *parLapply()* the documentation would suggest *parLapplyLB()* is the way to do this but there appears to be a bug in *parLapplyLB()* such that no load-balancing is done.

4 Illustrating the principles in specific case studies

4.1 Scenario 1: one model fit

Scenario: You need to fit a single statistical/machine learning model, such as a random forest or regression model, to your data.

4.1.1 Scenario 1A:

A given method may have been written to use parallelization and you simply need to figure out how to invoke the method for it to use multiple cores.

For example the documentation for the *randomForest* package doesn't indicate it can use multiple cores, but the *ranger* package can – note the *num.threads* argument.

```
args(ranger::ranger)
```

```
## function (formula = NULL, data = NULL, num.trees = 500, mtry = NULL,
##     importance = "none", write.forest = TRUE, probability = FALSE,
##     min.node.size = NULL, replace = TRUE, sample.fraction = ifelse(repla
##         1, 0.632), case.weights = NULL, class.weights = NULL,
##     splitrule = NULL, num.random.splits = 1, alpha = 0.5, minprop = 0.1,
##     split.select.weights = NULL, always.split.variables = NULL,
##     respect.unordered.factors = NULL, scale.permutation.importance = FAL
##     keep.inbag = FALSE, holdout = FALSE, quantreg = FALSE, num.threads =
##     save.memory = FALSE, verbose = TRUE, seed = NULL, dependent.variable
##     status.variable.name = NULL, classification = NULL)
## NULL
```

4.1.2 Scenario 1B:

If a method does linear algebra computations on large matrices/vectors, R can call out to parallelized linear algebra packages (the BLAS and LAPACK).

The BLAS is the library of basic linear algebra operations (written in Fortran or C). A fast BLAS can greatly speed up linear algebra in R relative to the default BLAS that comes with R. Some fast BLAS libraries are

- Intel's *MKL*; available for educational use for free
- *OpenBLAS*; open source and free
- *vecLib* for Macs; provided with your Mac

In addition to being fast when used on a single core, all of these BLAS libraries are 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 threaded BLAS installed on your machine and provided the environment variable OMP_NUM_THREADS is not set to one. (Macs make use of VECLIB_MAXIMUM_THREADS rather than OMP_NUM_THREADS.)

Threading in R is limited to linear algebra, provided R is linked against a threaded BLAS.

Here's some code that illustrates the speed of using a threaded BLAS:

```
library(RhpcBLASctl)
x <- matrix(rnorm(5000^2), 5000)

blas_set_num_threads(4)
```

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

##    user  system elapsed
##   8.316   2.260   2.692

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

##    user  system elapsed
##   6.360   0.036   6.399
```

Here the elapsed time indicates that using four threads gave us a two-three times (2-3x) speedup in terms of real time, while the user time indicates that the threaded calculation took a bit more total processing time (combining time across all processors) because of the overhead of using multiple threads.

Note that the code also illustrates use of an R package that can control the number of threads from within R, but you could also have set OMP_NUM_THREADS before starting R.

To use an optimized BLAS with R, talk to your systems administrator, see Section A.3 of the R Installation and Administration Manual (https://cran.r-project.org/manuals.html), or see these instructions to use vecLib BLAS from Apple's Accelerate framework on your own Mac: http://statistics.berkeley.edu/computing/blas.

It's also possible to use an optimized BLAS with Python's *numpy* and *scipy* packages, on either Linux or using the Mac's vecLib BLAS. Details will depend on how you install Python, numpy, and scipy.

4.2 Scenario 2: three different prediction methods on your data

Scenario: You need to fit three different statistical/machine learning models to your data.

What are some options?

- use one core per model

- if you have rather more than three cores, apply the ideas here combined with Scenario 1 above - with access to a cluster and parallelized implementations of each model, you might use one node per model

```
library(parallel)
n <- 10000000
system.time({
        p <- mcparallel(mean(rnorm(n)))
        q <- mcparallel(mean(rgamma(n, shape = 1)))
        s <- mcparallel(mean(rt(n, df = 3)))
        res <- mccollect(list(p,q, s))
})

##    user  system elapsed
##   2.832   0.180   2.545

system.time({
        p <- mean(rnorm(n))
        q <- mean(rgamma(n, shape = 1))
        s <- mean(rt(n, df = 3))
})

##    user  system elapsed
##   5.036   0.060   5.098
```

Why might this not have shown a perfect three-fold speedup?

You could also have used tools like *foreach* and *parLapply* here as well, as we'll discuss next.

4.3 Scenario 3: 10-fold CV and 10 or fewer cores

Scenario: You are running a prediction method on 10 cross-validation folds.

Here I'll illustrate parallel looping, using this simulated dataset and basic use of *randomForest()*.

```
library(randomForest)

## randomForest 4.6-14
## Type rfNews() to see new features/changes/bug fixes.
```

```
cvFit <- function(foldIdx, folds, Y, X, loadLib = FALSE) {
    if(loadLib)
        library(randomForest)
    out <- randomForest(y = Y[folds != foldIdx],
                        x = X[folds != foldIdx, ],
                        xtest = X[folds == foldIdx, ])
    return(out$test$predicted)
}

set.seed(23432)
## training set
n <- 1000
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)
nFolds <- 10
folds <- sample(rep(seq_len(nFolds), each = n/nFolds), replace = FALSE)
```

4.3.1 Using a parallelized for loop with *foreach*

The foreach package provides a *foreach* command that allows you to do this easily. foreach can use a variety of parallel "back-ends". For our purposes, the main one is use of the *parallel* package to use shared memory cores. When using *parallel* as the back-end, you should see multiple processes (as many as you registered; ideally each at 100%) when you monitor CPU usage. The multiple processes are created by forking or using sockets.

Note that *foreach* also provides functionality for collecting and managing the results to avoid some of the bookkeeping you would need to do if writing your own standard for loop. The result of *foreach* will generally be a list, unless we request the results be combined in different way, using the *.combine* argument.

```
library(doParallel)  # uses parallel package, a core R package

## Loading required package:  foreach
## Loading required package:  iterators
```

```
nCores <- 4
registerDoParallel(nCores)

result <- foreach(i = seq_len(nFolds)) %dopar% {
        cat('Starting ', i, 'th job.\n', sep = '')
        output <- cvFit(i, folds, Y, X)
        cat('Finishing ', i, 'th job.\n', sep = '')
        output # this will become part of the out object
}

length(list)

## [1] 1

result[[1]][1:5]

##          3          9         12         27         29
##  0.4122723  2.9946570 -0.2290716  2.5510170  1.0017545
```

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.

(Note that the printed statements from *cat* are not showing up in the creation of this document but should show if you run the code.)

4.3.2 Alternatively using parallel apply statements

The *parallel* package has the ability to parallelize the various *apply* functions (*apply*, *lapply*, *sapply*, etc.). It's a bit hard to find the vignette for the parallel package (http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf) because *parallel* is not listed as one of the contributed packages on CRAN (it gets installed with R by default).

We'll consider parallel *lapply* and *sapply*. These rely on having started a cluster using *makeCluster*, which starts new jobs via *Rscript* and communicates via a technology called sockets.

```
library(parallel)
nCores <- 4
cl <- makeCluster(nCores)
```

```
## clusterExport(cl, c('x', 'y')) # if the processes need objects
## from main R workspace (not needed here as no global vars used)

input <- seq_len(nFolds)

## need to load randomForest package within function
## when using par{L,S}apply, the last) argument being
## set to TRUE causes this to happen (see cvFit())
system.time(
        res <- parSapply(cl, input, cvFit, folds, Y, X, loadLib = TRUE)
)

##    user  system elapsed
##   0.004   0.004  14.046

system.time(
        res2 <- sapply(input, cvFit, folds, Y, X)
)

##    user  system elapsed
##  42.880   0.004  42.884
```

Here the miniscule user time is probably because the time spent in the worker processes is not counted at the level of the overall master process that dispatches the workers.

For help with these functions and additional related parallelization functions (including *parApply()*), see the help on *clusterApply*.

Now suppose you have 4 cores (and therefore won't have an equal number of tasks per core). The approach in the next scenario should work better.

4.4 Scenario 4: parallelizing over prediction methods

Scenario: parallelizing over prediction methods or other cases where execution time varies

If you need to parallelize over prediction methods or in other contexts in which the computation time for the different tasks varies widely, you want to avoid having the parallelization tool group the tasks in advance, because some cores may finish a lot more quickly than others. In many cases, this sort of prescheduling or 'static' allocation of tasks to workers is the default.

mclapply() is an alternative to *parSapply/parLapply* that uses forking to start up the worker processes. Here we see how to use *mclapply* to avoid prescheduling in a toy example.

```
library(parallel)

nCores <- 4

## specifically designed to be slow when have four cores and
## and use prescheduling, because
## the slow tasks all assigned to one worker
n <- rep(c(1e7, 1e5, 1e5, 1e5), 4)


fun <- function(n) {
    cat("working on ", n, "\n")
    mean(rnorm(n))
}

system.time(
        res <- mclapply(n, fun, mc.cores = nCores)
)

##    user  system elapsed
##  13.856   0.040   3.545

system.time(
        res <- mclapply(n, fun, mc.cores = nCores, mc.preschedule = FALSE)
)

##    user  system elapsed
##   6.416   0.460   1.127
```

4.5 Scenario 5: 10-fold CV across multiple methods with many more than 10 cores

Scenario: You are running an ensemble prediction method such as SuperLearner or Bayesian model averaging on 10 cross-validation folds, with many statistical/machine learning methods.

Here you want to take advantage of all the cores you have available, so you can't just parallelize over folds. There are a couple ways we can deal with such nested parallelization.

4.5.1 Scenario 5A: nested parallelization

One can always flatten the looping, either in a for loop or in similar ways when using apply-style statements.

```r
## original code: multiple loops
for(fold in 1:n) {
  for(method in 1:M) {
     ### code here
  }
}
## revised code: flatten the loops
output <- foreach(idx = 1:(n*M)) %dopar% {
   fold <- idx %/% M + 1
   method <- idx %% M + 1
   ### code here
}
```

Alternatively, *foreach* supports nested parallelization as follows:

```r
output <- foreach(fold = 1:n) %:%
  foreach(method = 1:M) %dopar% {
     ## code here
}
```

The '%:%' basically causes the nesting to be flattened, with n*M total tasks run in parallel.

4.5.2 Scenario 5B: Parallelizing across multiple nodes

If you have access to multiple machines networked together, including a Linux cluster, you can use *foreach* across multiple nodes.

The *doSNOW* backend has the advantage over *doMPI* in that it doesn't need to have MPI installed on the system. MPI can be tricky to install and keep working, so this is an easy approach to using foreach across multiple machines.

Simply start R as you usually would.

Here's R code for using doSNOW as the back-end to foreach. Make sure to use the `type = "SOCK"` argument or doSNOW will actually use MPI behind the scenes.

```r
library(doSNOW)

## Specify the machines you have access to and
##    number of cores to use on each:
machines = c(rep("beren.berkeley.edu", 1),
    rep("gandalf.berkeley.edu", 1),
    rep("arwen.berkeley.edu", 2))

## On Savio and other clusters using the SLURM scheduler:
## machines <- system('srun hostname', intern = TRUE)

cl = makeCluster(machines, type = "SOCK")
cl

registerDoSNOW(cl)

fun = function(i, n = 1e6)
  out = mean(rnorm(n))

nTasks <- 120

print(system.time(out <- foreach(i = 1:nTasks) %dopar% {
        outSub <- fun(i)
        outSub # this will become part of the out object
}))

stopCluster(cl)
```

To use parallel variations on apply/sapply/lapply:

```r
library(parallel)

machines = c(rep("beren.berkeley.edu", 1),
    rep("gandalf.berkeley.edu", 1),
    rep("arwen.berkeley.edu", 2))
cl = makeCluster(machines)
```

```
cl

n = 1e7
## copy global variable to workers
clusterExport(cl, c('n'))

fun = function(i)
  out = mean(rnorm(n))

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

result[1:5]

stopCluster(cl)
```

4.6 Scenario 6: Stratified analysis on a very large dataset

Scenario: You are doing stratified analysis on a very large dataset and want to avoid unnecessary copies.

In many of R's parallelization tools, if you try to parallelize this case on a single node, you end up making copies of the original dataset, which both takes up time and eats up memory. Here R copies 'data' when it passes it in as the argument to the calls to *testfun()*.

```
library(parallel)

testfun <- function(i, data, ids) {
    return(mean(data[ids[[i]], 2]))
}

nStrata <- 100
n <- 3e7
data <- cbind(rep(seq_len(nStrata), each = n/nStrata), rnorm(n))
object.size(data)

ids <- lapply(seq_len(nStrata), function(i) which(data[,1] == i))
```

```r
## in terminal monitor memory use with: watch -n 0.1 free -h

cl <- makeCluster(4)
parSapply(cl, seq_len(nStrata), testfun, data, ids)
stopCluster(cl)
```

However, if you are working on a single machine (i.e., with shared memory) you can avoid this by using parallelization strategies that fork the original R process (i.e., make a copy of the process) and use the big data objects in the global environment (yes, this violates the usual programming best practices of not using global variables). So here we avoid copying the original dataset.

```r
testfun_global <- function(i) {
    return(mean(data[ids[[i]], 2]))
}

## in terminal monitor memory use with: watch -n 0.1 free -h

## Forked processes do not make copies of 'data'
cl <- makeCluster(4, type = "FORK")
parSapply(cl, seq_len(nStrata), testfun_global)
stopCluster(cl)

## To contrast, we see what happens if not using
## a fork-based cluster.
## Note that this would also require exporting the global variable(s).

cl <- makeCluster(4)
## This errors because 'data' not available on workers
parSapply(cl, seq_len(nStrata), testfun_global)
clusterExport(cl, 'data')
parSapply(cl, seq_len(nStrata), testfun_global)
stopCluster(cl)
```

Parallelizing across nodes requires copying any big data across machines (one can't fork processes across nodes), which will be slow.

4.7 Scenario 7: Simulation study with n=1000 replicates: parallel random number generation

We won't cover this in class, though I will mention the issue in the simulation unit when we talk about random number generation.

In the previous example, we set the random number seed to different values for each bootstrap sample. One danger in setting the seed like that is that the random numbers in the different bootstrap samples could overlap somewhat. This is probably somewhat unlikely if you are not generating a huge number of random numbers, but it's unclear how safe it is.

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.

Scenario: You are running a simulation study with n=1000 replicates.

Each replicate involves fitting 20 statistical/machine learning methods.

Here, unless you really have access to multiple hundreds of cores, you might as well just parallelize across replicates.

However, you need to think about random number generation. If you have overlap in the random numbers the replications may not be fully independent.

In R, the *rlecuyer* package deals with this. The L'Ecuyer algorithm has a period of $2^{191}$, which it divides into subsequences of length $2^{127}$.

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

```r
library(parallel)
library(rlecuyer)

nSims <- 250
taskFun <- function(i){
        val <- runif(1)
        return(val)
}
```

```
nCores <- 4
RNGkind()

## [1] "Mersenne-Twister" "Inversion"

cl <- makeCluster(nCores)
iseed <- 1
clusterSetRNGStream(cl = cl, iseed = iseed)
RNGkind() # clusterSetRNGStream sets RNGkind as L'Ecuyer-CMRG

## [1] "Mersenne-Twister" "Inversion"

## but it doesn't show up here on the master
res <- parSapply(cl, 1:nSims, taskFun)
## now redo with same master seed to see results are the same
clusterSetRNGStream(cl = cl, iseed = iseed)
res2 <- parSapply(cl, 1:nSims, taskFun)
identical(res,res2)

## [1] TRUE

stopCluster(cl)
```

5 Additional details and topics

5.1 Limitations in Windows

Forking in R is not possible on Windows, so parallelization on Windows would generally need to use approaches based on sockets and not based on forking.

5.2 Setting the number of threads (cores used) in threaded code (including parallel linear algebra in R)

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 the bash shell:

```
export OMP_NUM_THREADS=4
```

Do this before starting your R or Python session or before running your compiled executable.

Alternatively, you can set OMP_NUM_THREADS as you invoke your job, e.g., here with R:

```
OMP_NUM_THREADS=4 R CMD BATCH --no-save job.R job.out
```

5.3 Important warnings about use of threaded BLAS

5.3.1 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.

More generally, if you are using the parallel tools in Section 4 to simultaneously carry out many independent calculations (tasks), it is likely to be more effective to use the fixed number of cores available on your machine so as to split up the tasks, one per core, without taking advantage of the threaded BLAS (i.e., restricting each process to a single thread).

5.3.2 Conflicts between openBLAS and various R functionality

In the past, I've seen various issues arising when using threaded linear algebra. In some cases when the parallelization uses forking, I have seen cases where R hangs and doesn't finish the linear algebra calculation.

I've also seen a conflict between threaded linear algebra and R profiling (recall the discussion of profiling in Unit 4).

Some solutions are to set OMP_NUM_THREADS to 1 to prevent the BLAS from doing threaded calculations or to use parallelization approaches that avoid forking.

5.4 Other packages for parallelization in R

5.4.1 The *future* package

The *future* package and related packages (*future.apply*, *future.batchtools*) are an entire system for doing computations in parallel in an integrated system where you can write your code once and then deploy it in multiple different parallelization contexts by simply changing some initial code that controls how the parallelization is done.

5.4.2 The *partools* package

partools is a new package developed by Norm Matloff at UC-Davis. He has the perspective that Spark/Hadoop are not the right tools in many cases when doing statistics-related work and has developed some simple tools for parallelizing computation across multiple nodes, also referred to as *Snowdoop*. The tools make use of the key idea in Hadoop of a distributed file system and distributed data objects but avoid the complications of trying to ensure fault tolerance, which is critical only on very large clusters of machines.

5.4.3 *pbdR*

pbdR is an effort to enhance R's capability for distributed memory processing called pbdR (http://r-pbd.org). For an extensive tutorial, see the pbdDEMO vignette (https://github.com/wrathematics/pbdDEMO/blob/ guide.pdf?raw=true). *pbdR* is designed for SPMD processing in batch mode, which means that you start up multiple processes in a non-interactive fashion using mpirun. The same code runs in each R process so you need to have the code behavior depend on the process ID.

pbdR provides the following capabilities:

- the ability to do some parallel apply-style computations (this section),
- the ability to do distributed linear algebra by interfacing to *ScaLapack*, and
- an alternative to *Rmpi* for interfacing with MPI.

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