Unit 7: Parallel Processing

October 28, 2017

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 Overview

1.1 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.

1.2 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: computational tasks 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.
- 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.

1.3 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.

1.3.1 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.

1.3.2 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 won't cover distributed memory parallelization in this Unit, but we will touch on a flavor of it via Spark in Unit 8.

1.4 Some other approaches to parallel processing

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 2016, I gave a workshop on using GPUs.

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

1.4.2 Spark and Hadoop

Spark and Hadoop are systems for implementing computations in a distributed memory environment, using the MapReduce approach. We'll see this in the next unit.

1.4.3 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.

2 Threading, particularly for linear algebra

2.1 What is 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*; may be available for educational use for free
- *OpenBLAS*; open source and free
- AMD's *ACML*; 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.)

On the SCF, R is linked against OpenBLAS.

2.2 Using threading

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:

```
## be careful here - I'm having problems with
## this package causing R to crash...
# require(RhpcBLASctl)

## alternatively just start R multiple times having
## set OMP_NUM_THREADS outside of R

Z <- matrix(rnorm(5000^2), 5000)

## blas_set_num_threads(4)
system.time({
X <- crossprod(Z) # Z^t Z produces pos.def. matrix
U <- chol(X)      # U^t U = X
})
#   user  system elapsed
#  7.216   1.096   2.219


## blas_set_num_threads(1)
system.time({
X <- crossprod(Z)
U <- chol(X)
})
#   user  system elapsed
#  6.360   0.204   6.563
```

Here the elapsed time indicates that using four threads gave us a three fold (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 (*RhpcBLASctl*) that can control the number of threads from within R.

2.3 Setting the number of threads (cores used)

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
```

2.4 Important warnings about use of threaded BLAS

2.4.1 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 3 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).

2.4.2 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 (we'll see when this is the case later in the unit), 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.

2.5 Using an optimized BLAS on your own machine(s)

To use an optimized BLAS with R, talk to your systems administrator, see Section A.3 of the R Installation and Administration Manual or see these instructions to use vecLib BLAS from Apple's Accelerate framework on your own Mac.

3 Basic parallelized loops/maps/apply

All of the functionality discussed here 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

3.1 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).

Question: Suppose you have n tasks to do where $n \gg p$. How should you divide up the n tasks amongst the p cores?

In the next sections, we'll see a few approaches in R for dealing with EP problems.

3.2 Parallel for loops with foreach

A simple way to exploit parallelism in R is to use the *foreach* package to do a for loop in parallel.

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. (foreach can also use *Rmpi* or *SNOW* to access cores in a distributed memory setting; please see the tutorial on distributed parallel processing mentioned above.)

Here we'll parallelize leave-one-out cross-validation for a random forest model. An iteration involves holding out a data point, fitting the model with all the other data points, and then predicting the held-out point. First, here's the code for doing a cross-validation prediction and for generating some fake data.

```
library(randomForest)

## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.

looFit <- function(i, Y, X, loadLib = FALSE) {
    if(loadLib)
        library(randomForest)
```

```
    out <- randomForest(y = Y[-i], x = X[-i, ], xtest = X[i, ])
    return(out$test$predicted)
}

set.seed(1)
## training set
n <- 500
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)
```

Now here's how we use foreach to do the computation in parallel:

```
require(parallel) # one of the core R packages
require(doParallel)

## Loading required package:  doParallel
## Loading required package:  foreach
## Loading required package:  iterators

library(foreach)

nCores <- 4
registerDoParallel(nCores)

nSub <- 30  # do only first 30 for illustration

result <- foreach(i = 1:nSub) %dopar% {
        cat('Starting ', i, 'th job.\n', sep = '')
        output <- looFit(i, Y, X)
        cat('Finishing ', i, 'th job.\n', sep = '')
        output # this will become part of the out object
}
print(result[1:5])

## [[1]]
```

```
##           1
## -0.5365734
##
## [[2]]
##           2
## -0.4030276
##
## [[3]]
##          3
## 0.7258196
##
## [[4]]
##         4
## 1.052046
##
## [[5]]
##         5
## 1.387056
```

(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.)

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, as we do here using `.combine = c`.

You can debug by running serially using `%do%` rather than `%dopar%`.

3.3 Parallel apply functionality

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 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 *cluster()*, which uses the PSOCK mechanism as in the SNOW package - starting new jobs via Rscript and communicating via a technology called sockets.

```
require(parallel)
nCores <- 4
### using sockets
#
## ?clusterApply
cl <- makeCluster(nCores) # by default this uses sockets

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

input <- seq_len(nSub) # same as 1:nSub but more robust

# need to load randomForest package within function
# when using par{L,S}apply
system.time(
        res <- parSapply(cl, input, looFit, Y, X, TRUE)
)

##    user  system elapsed
##   0.000   0.004  16.569

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

##    user  system elapsed
##  56.016   0.024  56.043

res <- parLapply(cl, input, looFit, Y, X, TRUE)
```

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*.

mclapply() is an alternative that uses forking to start up the worker processes.

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

##    user  system elapsed
##  59.000   0.052  15.863
```

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 parallel package directly.

3.4 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.

3.5 Loading packages and accessing global variables within your parallel tasks

Whether you need to explicitly load packages and export global variables from the master process to the parallelized worker processes depends on the details of how you are doing the parallelization.

With foreach with the *doParallel* backend, parallel apply-style statements (starting the cluster via *makeForkCluster()*, instead of the default *makeCluster()*), and *mclapply()*, packages and global variables in the main R process are automatically available to the worker tasks without any work on your part. This is because all of these approaches fork the original R process, thereby creating worker processes with the same state as the original R process. Interestingly, this means that global variables in the forked worker processes are just references to the objects in memory in the original R process. So the additional processes do not use additional memory for those objects (despite what is shown in *top*) and there is no time involved in making copies. However, if you modify objects in the worker processes then copies are made, as we've seen to be generally the case with R.

Let's experiment with that with foreach where foreach uses forking by default on the machine I'm running this on. Note that this seems to indicate no copy is made until a change is made, but also note what seems strange about the new address of 'x'. I'm not sure what is going on with regard to the latter.

```r
library(parallel) # one of the core R packages
library(doParallel)  # loads foreach as a dependency

nCores <- 4
registerDoParallel(nCores)

library(pryr)
x <- rnorm(10)
address(x)

## [1] "0x4e12f98"

result <- foreach(i = 1:3) %dopar% {
    set.seed(i)
    tmp <- address(x) # original address
    x[3] <- rnorm(1)
    out <- c(orig = tmp, new = address(x))
}
result

## [[1]]
##        orig         new
## "0x4e12f98" "0x4bb8e90"
##
## [[2]]
##        orig         new
## "0x4e12f98" "0x4bb8e90"
##
## [[3]]
##        orig         new
## "0x4e12f98" "0x4bb8e90"

## note when this is run manually in R, 'x' is not copied
## when x[3] is modified; not sure why the different behavior
address(x)

## [1] "0x4e12f98"
```

```
x[3] <- 2.1
address(x)

## [1] "0x4a79398"
```

In contrast, when processes are not forked, we can see that a copy is being made for each process from the very beginning:

```
cl <- makeCluster(4)
cl  # no forking

## socket cluster with 4 nodes on host 'localhost'

registerDoParallel(cl)

x <- rnorm(10)
library(pryr)
address(x)

## [1] "0x4a75ff8"

result <- foreach(i = 1:3, .packages = 'pryr') %dopar% {
    address(x)  # print(.Internal(inspect)) doesn't print to screen
}
result

## [[1]]
## [1] "0x26507f8"
##
## [[2]]
## [1] "0x1f7a7f8"
##
## [[3]]
## [1] "0x17e77f8"
```

In contrast, with parallel apply-style statements when starting the cluster using the default *makeCluster()* (which sets up a so-called *PSOCK* cluster, starting the R worker processes via Rscript), one needs to load packages within the code that is executed in parallel. In addition one needs to use *clusterExport()* to tell R which objects in the global environment should be available

to the worker processes. This involves making as many copies of the objects as there are worker processes, so one can easily exceed the physical memory (RAM) on the machine if one has large objects, and the copying of large objects will take time.

4 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.