Parallel Processing

October 11, 2014

References:

- Schmidberger et al. (2009). "State of the Art in Parallel Computing with R." Journal of Statistical Software, 31(1), 127. http://www.jstatsoft.org/v31/i01
- Tutorials I've prepared on parallel computing: http://statistics.berkeley.edu/computing/parallel

That first reference is a bit old, and I've pulled material from a variety of sources, often presentations, and not done a good job documenting this, so I don't have a good list of references for this topic.

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 often 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 Jaguar supercomputer at Oak Ridge National Lab, which has 18,688 nodes, each with two processors and each processor

with 6 cores, giving 224,256 total processing cores. Each node has 16 Gb of memory for a total of 300 Tb.

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.

1.1 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.

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.

With shared memory parallelism, each core is accessing the same memory so there is no need to pass messages. But 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 focus on shared memory parallelism here in this unit, though *Rmpi* will come up briefly.

1.2 Types of memory

In this course we've talked some about computer memory without distinguishing between main memory and the cache. The cache is a small amount of memory that can be accessed very quickly by the processing units, much more quickly than the main memory. The data in the cache is generally data that was previously retrieved from memory or recently computed. One principal for writing efficient code is to write code that makes effective use of the cache by using data that is already in the cache. In this course we won't discuss this further given time constraints.

Note that there are different levels of cache, with L1 cache accessed more quickly than L2 cache and in turn L3 cache.

1.3 Graphics processing units (GPUs)

GPUs were formerly a special piece of hardware used by gamers and the like for quickly rendering (displaying) graphics on a computer. They do this by having hundreds of processing units and breaking up the computations involved in an embarrassingly parallel fashion (i.e., without

inter-processor communication). For particular tasks, GPUs are very fast. Nowadays GPUs are generally built onto PC motherboards whereas previously they were on a video card.

Researchers (including some statisticians) are increasingly looking into using add-on GPUs to do massively parallel computations with inexpensive hardware that one can easily add to one's existing machine. This has promise. However, some drawbacks in current implementations include the need to learn a specific programming language (similar to C) and limitations in terms of transferring data to the GPU and holding information in the memory of the GPU.

For more information you can see this workshop material that I presented in spring 2014.

1.4 Cloud computing

Amazon (through its EC2 service) and other companies (Google and Microsoft now have similar products) 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. For some basic information on this, SCF has the following tutorial.

2 Parallelization

2.1 Overview

A lot of parallel processing follows a master-slave paradigm. There is one master process that controls one or more slave processes. The master process sends out tasks and data to the slave processes and collects results back.

One comment about parallelized code is that it can be difficult to debug because communication problems can occur in addition to standard bugs. Debugging may require some understanding of the communication that goes on between processors. If you can first debug your code on one processor, that can be helpful.

2.2 Threading

One form of shared-memory parallel processing is *threading*. Here an algorithm is implemented across multiple "light-weight" processes called *threads* in a shared memory situation. Threads are multiple paths of execution within a single process. One can write one's own code to make use

of threading, e.g., using the *openMP* protocol for C/C++/Fortran. For our purpose we'll focus on using pre-existing code or libraries that are threaded, specifically threaded versions of the BLAS.

In R, the basic strategy is to make sure that the R installation uses a threaded BLAS so that standard linear algebra computations are done in a threaded fashion. We can do this by linking R to a threaded BLAS library. Details can be found in Section A.3.1.5 of the R administration manual or talk to your system administrator.

Note that in R, the threading only helps with linear algebra operations (but the *pqR* engine seeks to change this - see my brief mention of *pqR* in Unit 6). In contrast, Matlab uses threading for a broader range of calculations.

2.2.1 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*, Apple's *VecLib* and the open source (and free) *openBLAS* (formerly *GotoBLAS*). 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. Using *top* (on a machine other than the cluster), you'll see the process using more than 100% of CPU. inconceivable! The default BLAS on the SCF Linux compute servers is *openBLAS* and on the SCF Linux cluster is *ACML*. The SCF Macs use *VecLib*, but this is not the default on Macs; ask me if you'd like more information on setting it up on your Mac.

```
require(RhpcBLASctl)

## Loading required package:  RhpcBLASctl

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

omp_set_num_threads(1)
system.time({
    X <- crossprod(Z)   # Z^t Z produces pos.def. matrix
    U <- chol(X)
})   # U^t U = X

##    user  system elapsed
##  15.929   0.072  16.073
```

```
omp_set_num_threads(4)
system.time({
    X <- crossprod(Z)
    U <- chol(X)
})

##    user  system elapsed
##  22.949   0.108   6.997
```

2.2.2 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 UNIX environment variable (VECLIB_MAXIMUM_THREADS on a Mac. This can be used in the context of R or C code that uses BLAS or your own threaded C code, but this does not work with Matlab. In the UNIX bash shell, you'd do this as follows (e.g. to limit to 3 cores) (do this before starting R):

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

2.2.3 Problems with the threaded BLAS in R

All of these problems can be alleviated by setting OMP_NUM_THREADS to 1.

1. There is a conflict between forking in R and the threaded BLAS that in some cases affects *foreach* (when using the *multicore* and *parallel* backends), *mclapply()*, and (only if *cluster()* is set up with forking (not the default)) *par{L,S,}apply()*. The result is that if linear algebra is used within your parallel code, R hangs. This affects both *openBLAS* and *ACML* under certain circumstances, so affects the SCF Linux machines. Alternatively, you can use MPI as the parallel backend (via *doMPI* in place of *doMC* or *doParallel*). You may also be able to convert your code to use *par{L,S,}apply()* [with the default PSOCK type] and avoid *foreach* entirely.

2. 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.

2.2.4 It may not make sense to use the 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 have an embarrassingly parallel job, it is likely to be more effective to use the fixed number of multiple cores you have access to to split along the embarrassingly parallel dimension without taking advantage of the threaded BLAS (i.e., restricting each process to a single thread).

Therefore I recommend that you test any large jobs to compare performance with a single thread vs. multiple threads. Only if you see a substantive improvement with multiple threads does it make sense to have OMP_NUM_THREADS be greater than one.

2.3 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

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 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 processors. This gives us a speedup of p, which is called linear speedup (basically anytime the speedup is of the form cp for some constant c).

One difficulty is load balancing. We'd like to make sure each slave process finishes at the same time. Often we can give each process the same amount of work, but if we have a mix of faster

and slower processors, things become more difficult. To the extent it is possible to break up a job into many small tasks and have processors start new tasks as they finish off old tasks, this can be effective, but may involve some parallel programming.

Question: What do you think the tradeoffs are between breaking up a problem into many small subtasks vs. a few large subtasks?

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

2.4 Parallelization with communication

If we do not have an EP problem, we have one that involves some sort of serial calculation. As a result, different processes need to communicate with each other. There are standard protocols for such communication, with *MPI* being most common. You can use C libraries that implement these protocols. While MPI has many functions, a core of 6-10 functions (basic functions for functionality such as sending and receiving data between processes - either master-slave or slave-slave) are what we mostly need.

R provides the *Rmpi* library, which allows you to do message passing in R. It has some drawbacks, but may be worth exploring if you have a non-EP problem and don't want to learn C. Installing *Rmpi* may be tricky and on institutional machines will require you talk to your systems administrator. *Rmpi* is a basic building block for other parallel processing functionality such as the *doMPI* backend to *foreach* and for *SNOW*.

For non-EP problems, the primary question is how the speed of the computation scales with p. This will generally be much worse than $1/p$. Furthermore, as p increases, if communication must increase as well, then the speedup can be much worse. Your work can become communication-bound rather than CPU-bound. Whenever messages are sent, there is a cost that scales with the number of message sent (called *latency*) and a cost that scales with the amount of data that is passed in those messages.

The term high-performance computing (HPC) is the term associated with tools and theory for doing parallel processing involving this sort of communication.

3 Explicit parallel code in R

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

- *threading*: multiple paths of execution within a single process; the OS sees the threads as a single process, but one can think of them as 'lightweight' processes
- *forking*: child processes are spawned that are identical to the parent, but with different process IDs and their own memory

- *sockets*: some of R's parallel functionality involves creating new R processes and communicating with them via a communication technology called sockets

3.1 *foreach*

A simple way to exploit parallelism in R when you have an EP problem 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 when MPI is available or (our focus here) the *parallel* or *multicore* packages to use shared memory cores. When using *parallel* or *multicore* as the back-end, you should see multiple processes (as many as you registered; ideally each at 100%) when you look at *top*. The multiple processes are generally created by forking.

```
require(parallel)  # one of the core R packages
require(doParallel)
# require(multicore); require(doMC) # alternative to parallel/doParallel
# require(Rmpi); require(doMPI) # when Rmpi is available as the back-end
library(foreach)
library(iterators)

taskFun <- function() {
    mn <- mean(rnorm(1e+07))
    return(mn)
}
nCores <- 4  # set based on the machine to be used
registerDoParallel(nCores)
# registerDoMC(nCores) # alternative to registerDoParallel cl <-
# startMPIcluster(nCores); registerDoMPI(cl) # when using Rmpi as the
# back-end
out <- foreach(i = 1:100, .combine = c) %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. Here I've explicitly told *foreach* to combine the results with *c()* (*cbind()* and *rbind()* are other common choices), but it will often be smart enough to figure it out on its own. 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* code block to ensure a package is available to all of the calculations.

Warning (repeated from before): There are sometimes conflicts between *foreach* and the threaded BLAS, so before running an R job that does linear algebra within a call to *foreach*, you may need to set *OMP_NUM_THREADS* to 1 to prevent the BLAS from doing threaded calculations.

3.2 Parallel apply and vectorization (parallel package)

The *parallel* package has the ability to (1) parallelize the various *apply()* functions (*apply()*, *lapply()*, *sapply()*, etc.) and (2) parallelize vectorized functions, among other things. The *multicore* package also has this ability and *parallel* is built upon *multicore*. *parallel* is a core R package so we'll explore the functionality in that setting. Here's the vignette for the parallel package – it's hard to find because *parallel* is not listed as one of the contributed packages on CRAN.

First let's consider parallel *apply()*.

```
require(parallel)
nCores <- 4
### using sockets ?clusterApply
cl <- makeCluster(nCores)  # by default this uses sockets
nSims <- 60
testFun <- function(i) {
    mn <- mean(rnorm(1e+06))
    return(mn)
}
# if the processes need objects (x and y, here) from the master's workspace
# clusterExport(cl, c('x', 'y'))
system.time(res <- parSapply(cl, 1:nSims, testFun))
```

```
system.time(res2 <- sapply(1:nSims, testFun))
myList <- as.list(1:nSims)
res <- parLapply(cl, myList, testFun)

### using forking
system.time(res <- mclapply(seq_len(nSims), testFun, mc.cores = nCores))
```

In *mclapply()*, there is an option, *mc.preschedule*, that if set to TRUE (the default), causes the jobs to be divided in advance amongst the cores. For individual tasks with high variation in completion time, setting this to FALSE is a good idea. Why? Similarly, there are 'load-balancing' versions of *par{S,L}apply()*: *par{L,S}applyLB()*.

Now let's consider parallel evaluation of a vectorized function. *exp()* is not a great example, because it's quite fast anyway, so I also show an example with *Matern()*, which calculates correlation as a parameterized function of distance (e.g., time lag) in a vectorized fashion.

```
require(parallel)
nCores <- 4
x <- rnorm(1e+07)
expx <- pvec(x, exp, mc.cores = nCores)
library(fields)
ds <- runif(6e+06, 0.1, 10)
system.time(corVals <- pvec(ds, Matern, 0.1, 2, mc.cores = nCores))
system.time(corVals <- Matern(ds, 0.1, 2))
```

Note that some R packages can directly interact with the parallelization packages to work with multiple cores. E.g., the *boot* package can make use of the *multicore* package directly.

3.3 Explicit parallel programming in R: mcparallel and forking

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

3.3.1 Using mcparallel to dispatch blocks of code to different processes

First one can use *mcparallel()* in the *parallel* package to send different chunks of code to different processes.

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

3.3.2 Explicitly forking code in R

The *fork* package and *fork()* function in R provide an implementation of the UNIX *fork* system call for forking a process. Note that the code here does not handle passing information back from the child very well. One approach is to use sockets – the help page for *fork()* has a bit more information.

```
library(fork)
# mode 1 of how to use fork()
pid <- fork(slave = myfun)
# mode 2 of how to use fork() this set of braces is REQUIRED when you don't
# pass a function to the slave argument of fork()
{
    pid <- fork(slave = NULL)
    if (pid == 0) {
        cat("Starting child process execution.\n")
        tmpChild <- mean(rnorm(1e+07))
        cat("Result is ", tmpChild, "\n", sep = "")
        save(tmpChild, file = "child.RData")  # clunky
        cat("Finishing child process execution.\n")
        exit()
    } else {
        cat("Starting parent process execution.\n")
        tmpParent <- mean(rnorm(1e+07))
```

```
        cat("Finishing parent process execution.\n")
        wait(pid)  # wait til child is finished so can read
        # in updated child.RData below
    }
}
load("child.RData")  # clunky
print(c(tmpParent, tmpChild))
```

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