3 Parallel computing

Parallel computing divides calculations into smaller problems and solves these simultaneously using multiple computing elements (hereafter "workers"). Parallel computing, once prominently used only in supercomputing (e.g. Hager & Wellein, 2010), is today commonplace as most desktop computers and laptops have CPUs with multiple "cores". Parallel computing has also become relatively straightforward to implement, but is not implemented by default in R (Knaus *et al.*, 2009). In the next sections we provide an introduction to parallel programming. The benefits of parallelism can be great, however before implementing parallel-programming or investing in hardware, a handful of basic rules are worth reviewing.

1. A common misconception is that when more processes are run in parallel the faster the problem will be solved (Hager & Wellein, 2010). This misconception is easily seen by applying Amdahl's Law (Fig. 3 in the maintext), which shows that the possible speed-up is asymptotically related to the amount of parallel processes. Even when a large fraction of the code can be run in parallel the maximum speed-up is limited (and each additional worker offers diminishing returns). Applying both profiling and Amdahl's law can identify the optimal amount of parallel processes beforehand. We provide this functionality in the package *aprof*.

2. Parallel computation incurs overhead. Initializing a parallel computation takes some time, including communication between devices or across nodes to copy code and data. When individual jobs are not reasonably intensive compared to the time it takes to initiate the parallelisation (i.e. the ratio of computation to communication is low), repeatedly starting parallel threads will deteriorate overall performance. This sometimes means that one should not use all workers available, especially for smaller repetitive tasks.

3. Avoid over-parallelisation, and do not assign more workers than available. If the machine is used for additional tasks besides calculation, leave some resources (i.e. physical cores) for other processes. If not, the system may become unstable.

4. In most computing devices, memory (RAM) is shared among parallel processes (Schmidberger *et al.*, 2009). Therefore, each of N processes running in parallel will only receive 1/N of the available memory. If memory is not available within a thread, threads will automatically attempt to "borrow" memory from another process. While parallel workers wait for memory to become available, overhead is drastically increased to the point that execution of the problem in serial may be faster. One can use R functions as *object.size* to determine memory usage in an algorithm.

5. Independence of random number sequences must be ensured for valid scientific results (e.g. L'Ecuyer, 2012). A crucial consideration when conduct-

ing operations that rely on random numbers is that all parallel calculations have unique sequences that will not overlap and can be reproduced. Specific techniques are available that ensure independence of random number sequences within R (e.g. L'Ecuyer, 1999, section 3.4)).

3.1 Parallel computing in R

Programming in parallel in short, involves starting a main process, often called the parent process, which (1) initializes the calculations, (2) divides them and sends these as "jobs" to "child" processes and (3) waits for the "child" processes to return results once they have completed.

R has multiple packages to execute parallel code. These packages include *foreach*, *multicore* and *snow* to name a few (Weston & Computing, 2013; Urbanek, 2011; Tierney *et al.*, 2013), and some packages for parallel calculations on a graphics processing unit or "GPU" (e.g. gputools Buckner *et al.*, 2013). In this tutorial we will use the package *parallel* which is shipped with the core version of R (since R version 2.14.0). In this document we will focus on parallel computing on a single machine (i.e. a laptop) with two or more cores (these techniques will also work on servers with many more cores). We do not focus on parallel computing via clusters or using several computers connected via ethernet, though the reader should be aware that these operations are readily available and easy to implement in R (see e.g. the *snow* package Tierney *et al.*, 2013).

Take special care when using programs (e.g. a custom BLAS library, pqR) that automatically make use of parallel operations, as these can easily cause the proliferation of jobs far beyond the amount of workers if used within parallel algorithms.

3.2 Parallel calculations using forking (Unix systems including Mac)

One of the easiest ways of conducting parallel computations in R is through forking. However, this can only be done on unix machines (essentially anything but Windows). In the next section (3.5) we give an example for windows. Windows users should not skip this part, as we address some key issues for parallel computing. They may, however, skip running most of the code examples. In these sections, we use so-called implicit parallelism (this is nice as most of the setup is done by the system and the user does not need to worry about it). So-called "explicit parallelism" is also possible in R, and here the user will have more control over the process. However, we do not discuss explicit parallelism here. More important considerations for parallel computing are given in the main document.

3.3 Number of workers

One thing that needs to be ensured in parallel computing is that we avoid over-parallelization. We should not assign more jobs then there are "workers", or we will surely paralyse our code. Therefore it is important to find out how many physical workers are available (for details see the main document). In R we can do this using the following code:

R Example 3.1.

```
## OnlineRcode40.R

require(parallel)

## Loading required package:  parallel

ncores<-detectCores()
print(ncores)

## [1] 8
```

We see that the computer on which this document was made, has 8 cores. However we should watch out, as we are working on a multi-threaded machine. However, we don't want to use all the cores when running these codes, as we are using our machine for more than just computation (e.g. writing this document). This is why we adjust the output of *detectCores*, below, to ensure we don't use all the resources (physical cores and dual-threaded capabilities) for computation. We do this to ensure stability of the system by leaving enough computing power for the other things we need to do. If you are using a dedicated machine (e.g. on a server) you can of course assign all the workers available to you (though even then it may be smart to leave some idle).

R Example 3.2.

```
## OnlineRcode41.R

require(parallel)
ncores<-detectCores()/2
print(ncores)

## [1] 4
```

3.4 Random numbers

Another thing we need to consider before we start is that bootstrapping relies on random number generation. We therefore need to ensure that the sequences of random number generated by R, within each child process, are truly independent. We therefore need each parallel stream to have a separate random seed. We must also consider that not all seeds are equally good as pseudo Random Number Generators (RNG) - as they are called - are typically also periodic, meaning that sequences will eventually repeat. If seeds are not chosen well, streams with each parallel calculation may overlap and will therefore no longer be independent. Lastly, we would also like our random numbers to be repeatable when we keep the same seed. To make sure all this takes place we change the random number generator in use to the L'Ecuyer RNG (L'Ecuyer, 1999) with $RNGkind("L'Ecuyer - CMRG")$, which is a random number generator specifically designed for use in parallel computations. This ensures that we will have independent and reproducible random numbers.

R Example 3.3.

```
## OnlineRcode42.R

## set RNG to "L'Ecuyer-CMRG"
RNGkind("L'Ecuyer-CMRG" )
```

Now that we know how many workers we have available and we know which RNG to use, let's build our first parallel algorithm to test if we can reproduce random numbers. We will generate random values, in parallel, from a normal distribution using *rnorm*. To do this, we will utilize the function *mcparallel* in R. You will see that, as we are using "implicit parallelism", its usage is remarkably simple. Note that when you build parallel algorithms you should avoid using any GUI elements (e.g. graph plotting, printing) as they may lead to problems. The code for windows users is given in 3.5.

R Example 3.4.

```
## OnlineRcode43.R

require(parallel)

## number of workers
ncores<-2

## make an object to save the output from each child process
```

```
children<-vector("list" , ncores)

# change the random number generator
RNGkind("L'Ecuyer-CMRG" )

## set an initial seed
set.seed(20130808)

## The following will make runs from mcparallel reproducible
mc.reset.stream()

## initialize each child process
for(i in 1:ncores){
        children[[i]]<-mcparallel(rnorm(4))
}

# collect results
randomnumbers<-parallel::mccollect(children)
```

Let's see if we can reproduce these by rerunning the code with the original seed:

R Example 3.5.

```
## OnlineRcode44.R

## reset an initial seed
set.seed(20130808)
mc.reset.stream()

## re-initialize each child process
    for(i in 1:ncores){
        children[[i]]<-mcparallel(rnorm(4))
        }

## collect results
randomnumbers2<-parallel::mccollect(children)
```

Now lets see if the random numbers are identical:

```
## OnlineRcode45.R

print(randomnumbers)

## $`6598`
## [1] -0.6172  0.1962  0.6680 -1.0425
##
## $`6599`
## [1]  0.4541 -0.1770  0.3327 -0.4369

print(randomnumbers2)

## $`6600`
## [1] -0.6172  0.1962  0.6680 -1.0425
##
## $`6601`
## [1]  0.4541 -0.1770  0.3327 -0.4369
```

Success! We now know how to conduct a parallel computation via forking, and ensure we have reproducible results. Let's apply this to our bootstrap problem and see what our gains are.

3.4.1 The parallel bootstrapping algorithm via forking

Building a parallel algorithm to conduct our bootstrap, will be relatively easy. However, we will need to find a way to split our calculations. In this case the most logical place to split the calculations into equal parts would be to divide the number of re-samples among workers. The following is one way to do this:

R Example 3.6.

```
## OnlineRcode46.R

subR<-1000

## Set the number of workers we will use
ncores<-3

## Split the jobs
splitR<-table(cut(1:subR,ncores,labels=F))
print(splitR)

##
```

```
##   1   2   3
## 334 333 333
```

As can be seen, we have now produced an equal split among 3 workers. Next, let's start our parallel bootstrap - by sending our (almost) equal job splits to the different children. As each of the children are independent processes we opt to not use a profiler to time the execution of this process. Instead we record the start and end times. The equivalent of the below code for windows users is given in section 3.5. Note that the windows code is slightly simplier, and also works well on *unix systems.

R Example 3.7.

```
## OnlineRcode47.R

## load the parallel package
require(parallel)

## save the start time
tp0 <- structure(.Internal(Sys.time()))

## initialize a list where we can store the id of each child
children<-vector('list" , ncores)

## send the division of work in splitR to each of the cores
for(i in 1:ncores){
        children[[i]] <- mcparallel(fastBoot(subBioData,
                                    splitR[i]))
}

## Wait for the child processes named in "children" to finish
results <- mccollect(children)

## Record end time
tp1 <- structure(.Internal(Sys.time()))

## Calculate execution time
tp <- tp1-tp0
```

Which gives us:

```
print(tp)
[1]  5.529923
```

That's it. We completed a parallel execution of the bootstrap and the final execution time of our program was about 5.5 seconds. This means that we gained a relative speed-up of 280% above our fastest serial code!

3.5 Alternative parallel bootstrapping algorithm (including windows)

Adapting the above guidelines for windows is straight forward.

R Example 3.8.

```
## OnlineRcode48.R

require(parallel)

## number of workers
ncores<-2

## set an original seed
set.seed(20130808)
mclapply(rep(4,ncores),rnorm)

## [[1]]
## [1] -0.6172  0.1962  0.6680 -1.0425
##
## [[2]]
## [1]  0.4541 -0.1770  0.3327 -0.4369

## reset the original seed
set.seed(20130808)
mclapply(rep(4,ncores),rnorm)

## [[1]]
## [1] -0.6172  0.1962  0.6680 -1.0425
##
## [[2]]
## [1]  0.4541 -0.1770  0.3327 -0.4369
```

Again, as in section 3.4, we succeeded in generating repeatable random numbers that are independent with each parallel process. Next let's finish the parallel bootstrap for windows. The following deploys the same parallel bootstrap as we did before:

R Example 3.9.

```
## OnlineRcode49.R

## load the parallel package
require(parallel)

##initialize the problem
subR<-1000

## Set the number of workers we will use
ncores<-3

## Split the jobs
splitR<-table(cut(1:subR,ncores,labels=F))

## save the start time
tp0 <- structure(.Internal(Sys.time()))

## send the division of work in splitR to each of the cores
results <- mclapply(splitR,function(X)
                fastBoot(subBioData,X))

## Record end time
tp1 <- structure(.Internal(Sys.time()))

## Calculate execution time
tp <- tp1-tp0
```

3.6 The end gains

The final question that needs to be answered is, in the end, how much did we gain when we conduct the full problem? That being 10 000 bootstrap resamples on the full dataset with 750 000 records. We have summarized the full run time of each program in table 3.1, here we see that in the end we have sped-up execution of our problem by a factor of ≈ 18.8 (from 1 hour 3 minutes to 3.35 minutes). This is a significant improvement.

3.7 Closing remarks

Parallel computation via cloud computing and computing clusters are also becoming common at many institutions (e.g. Harvard University's Odyssey Cluster), increasing the value to learn parallelisation techniques for the future. Be-

Program	NaiveBoot	LessNaiveBoot	DatatableBoot	Parallel
Execution time	1 hour and 3 minutes	38 minutes	12 minutes	3.35 minutes

Table 3.1: The end gains. Required execution time for each of the program versions to conduct 10 000 bootstrap resamples on a dataset of 750 000 records.

yond this, it is now possible for scientists to implement massively parallelised code on graphics cards (or graphics processing units, GPUs), which have hundreds to thousands of processors, and are relatively cheap (starting at only a few hundred dollars). New developments, such as the CUDA platform and programming model (Nvidia, 2007), which enables execution of C, C++ and Fortran code on GPUs, are freely available to scientists. Such prospects show that we are only starting to scrape the surface of computationally feasibility.

Most scientists use desktop or laptop computers (Hannay *et al.*, 2009), and in recent years most of these computers have become capable of parallel computing (Wilson, 1995). Coincidentally, many computationally intensive problems in the biological science rely on so-called 'embarrassingly parallel computations' (Grama, 2003), where a very large fraction of all calculations can be completed in parallel. Consequently, parallel computing is potentially the most useful technique highlighted here for certain problems, and as we show in our here (see 2.1), implementing code in parallel is also one of the simplest ways of speeding up an analysis. We hope that we have shown here that there is no reason not to take advantage of all the computing power available at our fingertips today.