ARCHER2 for Data Scientists

Practical 4: R

Adrian Jackson

1 Introduction

In this exercise we are going to try different methods of parallelising R on ARCHER2. This will include single node parallelisation functionality (i.e. using threads or processes to use cores within a single node), and distributed memory functionality that enables the parallelisation of R programs across multiple nodes in the system.

2 R

Recall that there is an optimised R module available on ARCHER2:

```
module load cray-R
```

Now try running the following R script using R on the ARCHER2 login node:

```
n <- 8*2048
A <- matrix( rnorm(n*n), ncol=n, nrow=n )
B <- matrix( rnorm(n*n), ncol=n, nrow=n )
C <- A %*% B
```

You can run this as follows on ARCHER2 (assuming you have saved the above code into a file named `matrix.R`):

```
Rscript ./matrix.R
```

You can check the resources used by R when running on the login node using this command:

```
top -u $USER
```

If you run the R script in the background using `&`, as follows, you can then monitor your run using the `top` command. You may notice when you run your program that at points R uses many more resources than a single core can provide, as demonstrated below:

```
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND

178357 adrianj   20   0 15.542 0.014t  13064 R 10862 2.773   9:01.66 R
```

In the example above it can be seen that `10862%` of a single core is being used by R. This is an example of R using automatic parallelisation, i.e. exploiting parallelisation from the Cray libsci scientific libraries. You can experiment with controlling the automatic parallelisation using the `OMP_NUM_THREADS` variable to restrict the number of cores available to R. Try using the following values:

```
export OMP_NUM_THREADS=8
```

```
export OMP_NUM_THREADS=4

export OMP_NUM_THREADS=2
```

You may also notice that not all the R script is parallelised. Only the actual matrix multiplication is undertaken in parallel, the initialisation/creation of the matrices is done in serial.

We can also experiment with the implicit parallelism in other libraries, such as `data.table`. You will first need to install this library on ARCHER2. To do this you can simply run the following command:

```
install.packages(data.table)
```

However, if you intend to use R on the compute nodes, you will need to ensure that the library is installed on the `/work` filesystem so it is accessible from both login and compute nodes. You can do this by setting the following variable before you run R:

```
export R_LIBS_USER=/work/ta055/ta055/$USER/Rinstall
```

This instructs R to install and use libraries from the above directory. You do not need to call it `Rinstall`, it can be called any name you like, but it will need to be on the `/work` filesystem to be accessible from the compute nodes.

Once you have installed `data.table` you can experiment with the following code:

```
library(data.table)
venue_data <- data.table( ID = 1:50000000,
                Capacity = sample(100:1000, size = 50000000, replace = T),
                Code = sample(LETTERS, 50000000, replace = T),
                Country    =    rep(c("England","Scotland","Wales","Northern
Ireland"), 50000000))
system.time(venue_data[, mean(Capacity), by = Country])
```

This creates some random data in a large data table and then performs a calculation on it. Try running R with varying numbers of threads to see what impact that has on performance. Remember, you can vary the number of threads R uses by setting `OMP_NUM_THREADS=` before you run R. If you want to try easily varying the number of threads you can save the above code into a script and run it using `Rscript`, changing `OMP_NUM_THREADS` each time you run it, i.e.:

```
$> export OMP_NUM_THREADS=1
$> Rscript ./data_table_test.R
$> export OMP_NUM_THREADS=2
$> Rscript ./data_table_test.R
…
```

The `elapsed` time that is printed out when the calculation is run represents how long the script/program took to run. It's important to bear in mind that, as with the matrix multiplication exercise, not everything will be parallelised. Creating the data table is done in serial so does not benefit from the addition of more threads.

3 Loop and function parallelism

R provides a number of different functions to run loops or functions in parallel. One of the commonest functions is to use are the {`X}apply` functions:

- `apply`: Apply a function over a matrix or data frame
- `lapply`: Apply a function over a list, vector, or data frame
- `sapply`: Same as `lapply` but returns a vector
- `vapply`: Same as `sapply` but with a specified return type that improves safety and can improve speed

For example:

```
res <- lapply(1:3, function(i) {
  sqrt(i)*sqrt(i*2)
})
```

The `{X}apply` functionality supports iteration over a dataset without requiring a loop to be constructed. However, the functions outlined above do not exploit parallelism, even if there is potential for parallelisation many operations that utilise them.

There are a number of mechanisms that can be used to implement parallelism using the `{X}apply` functions. One of the simplest is using the `parallel` library, and the `mclapply` function:

```
library(parallel)
res <- mclapply(1:3, function(i) {
  sqrt(i)
})
```

Try experimenting with the above functions on large numbers of iterations, both with `lapply` and `mclapply`. Can you achieve better performance using the `MC_CORES` environment variable to specify how many parallel processes R uses to complete these calculations. The default on ARCHER2 is 2 cores, but you can increase this in the same way we did for `OMP_NUM_THREADS`, i.e.:

```
export MC_CORES=16
```

Try different numbers of iterations of the functions (i.e. change `1:3` in the code to something much larger), and different numbers of parallel processes, i.e.:

```
export MC_CORES=2

export MC_CORES=8

export MC_CORES=16
```

If you have separate functions then the above approach will provide a simple method for parallelising using the resources within a single node. However, if your functionality is more loop based, then you may not wish to have to package this up into separate functions to parallelise.

The `foreach` package can be used to parallelise loops as well as functions. Consider a loop of the following form:

```
main_list <- c()
for (i in 1:3) {
   main_list <- c(main_list, sqrt(i))
}
```

This can be converted to `foreach` functionality as follows:

```
main_list <- c()
library(foreach)
foreach(i=1:3) %do% {
   main_list <- c(main_list, sqrt(i))
}
```

Whilst this approach does not significantly change the performance or functionality of the code, it does let us then exploit parallel functionality in foreach. The `%do%` can be replaced with a `%dopar%` which will execute the code in parallel.

To test this out we're going to try an example using the `randomForest` library. Firstly, we will need to install this on ARCHER2. As the version of R we have is a little too old to support the default `randomForest` library, we will need to install in a slightly different method. Run the following code in R on ARCHER2:

```
install.packages("https://cran.r-
project.org/src/contrib/Archive/randomForest/randomForest_4.6-14.tar.gz",
repos=NULL, type="source")
```

With randomForest installed we can now run the following code in R:

```
Library(foreach)
library(randomForest)
x <- matrix(runif(50000), 1000)
y <- gl(2, 500)
rf <- foreach(ntree=rep(250, 4), .combine=combine) %do%
  randomForest(x, y, ntree=ntree)
print(rf)
```

Implement the above code and run with a `system.time` to see how long it takes. Once you have done this you can change the `%do%` to a `%dopar%` and re-run. Does this provide any performance benefits?

To exploit the parallelism with `dopar` we need to provide parallel execution functionality and configure it to use extra cores on the system. One method to do this is using the `doParallel` package. Install the `doParallel` package and then add the code below to your `randomForest` example:

```
library(doParallel)
registerDoParallel(8)
```

Does this now improve performance when running the `randomForest` example? Experiment with different numbers of workers by changing the number set in `registerDoParallel(8)` to see what kind of performance you can get. Note, you may also need to change the number of clusters used in the `foreach`, i.e. what is specified in the `rep(250, 4)` part of the code, to enable more than 4 different sets to be run at once if using more than 4 workers. The amount of parallel workers you can use is dependent on the hardware you have access to, the number of workers you specify when you setup your parallel backend, and the amount of chunks of work you have to distribute with your `foreach` configuration.

It is possible to use different parallel backends for `foreach`. The one we have used in the example above creates new worker processes to provide the parallelism, but you can also use larger numbers of workers through a parallel cluster, i.e.:

```
my.cluster <- parallel::makeCluster(8)
registerDoParallel(cl = my.cluster)
```

By default `makeCluster` creates a socket cluster, where each worker is a new independent process. This can enable running the same R program across a range of systems, as it works on Linux and Windows (and other clients). However, you can also fork the existing R process to create your new workers, i.e.:

```
cl <-makeCluster(5, type="FORK")
```

This saves you from having to create the variables or objects that were setup in the R program/script prior to the creation of the cluster, as they are automatically copied to the workers when using this forking mode. However, it is limited to Linux style systems and cannot scale beyond a single node.

Once you have finished using a parallel cluster you should shut it down to free up computational resources, using `stopCluster`, i.e.:

```
stopCluster(cl)
```

When using clusters without the forking approach, you need to distribute objects and variables from the main process to the workers using the `clusterExport` function, i.e.:

```
library(parallel)
variableA <- 10
variableB <- 20
mySum <- function(x) variableA + variableB + x
cl <- makeCluster(4)
res <- try(parSapply(cl=cl, 1:40, mySum))
```

The program above will fail because `variableA` and `variableB` are not present on the cluster workers. Try the above on ARCHER2 and see what result you get.

To fix this issue you can modify the program using `clusterExport` to send `variableA` and `variableB` to the workers, prior to running the `parSapply` i.e.:

```
clusterExport(cl=cl, c('variableA', 'variableB'))
```

4 Distributed parallelism

The parallel library has the functionality to build clusters across nodes in an arbitrary system, using `makePSOCKcluster`. You can provide the information of the hosts to use and construct a cluster using this information as follows:

```
hosts <- c( rep("localhost",8), rep("192.168.0.10", 8) )
cl <- makePSOCKcluster(names=hosts)
clusterCall(cl, rnorm, 7)
clusterCall(cl, system, "hostname")
stopCluster(cl)
```

However, if you have a batch system and parallel libraries already installed on your system, such as ARCHER2, then it is possible to exploit this functionality using some R libraries that provide batch system integration and communication functionality.

Using the library `snow` you can setup a parallel cluster that will use MPI to distribute functions across all the processes available to a job. For parallel functions snow provides alternative to the {`x}apply` functions, just as the parallel library does, however rather than being `mc{x}apply` they are `par{X}apply`. An example would be:

```
library(snow)
cl <- makeCluster( mpi.universe.size(), type="MPI" )
clusterExport(cl, c('data'))
results <- parLapply( cl, c(25,25,25,25), fun=parallel.function )
…
stopCluster(cl)
mpi.exit()
```

The library `doSNOW` provides the same functionality, but to distribute loops across processes in a cluster using MPI by providing a parallel backend for `%dopar%` functionality, i.e.:

```
library(foreach)
library(doSNOW)
…
cl <- makeCluster( mpi.universe.size(), type='MPI' )
clusterExport(cl,c('data'))
registerDoSNOW(cl)
…
stopCluster(cl)
mpi.exit()
```

Try implementing the previous example where we parallelised randomForest using `%dopar%` with the `doSNOW` parallel backend, and the example where use used `mclapply` using the `snow` `parLapply` functionality, as described above. Try using a range of different numbers of workers to see how performance improves. To run your code you will need to submit a job to the ARCHER2 backend and specify a number of MPI workers to use for your parallel work. Below is an example batch script which will use 2 nodes, and 128 workers per node, parallelising across 256 workers in total (assuming your code is within a file called `my_code.R`:

```
#!/bin/bash
#SBATCH --job-name=snow
#SBATCH --nodes=2
#SBATCH --tasks-per-node=128
#SBATCH --time=0:5:0
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --hint=nomultithread

module load cray-R

export R_LIBS_USER=/work/ta055/ta055/$USER/Rinstall
export PATH=$PATH/work/ta055/ta055/$USER/Rinstall/snow/
export OMP_NUM_THREADS=1

srun RMPISNOW -q < ./my_code.R
```

Once you have completed these exercises there are a few more involved exercises you can try. Firstly, try to use a `RMPISNOW` cluster to parallelise the following application code:

```
sayhello <- function()
{
  info <- Sys.info()[c("nodename", "machine")]
  paste("Hello from", info[1], "with CPU type", info[2])
}
```

This will require adding the functionality to create and MPI cluster, and then run function with `clusterCall`, which will issue it to the available workers in the cluster. Remember to also stop the cluster at the end of the program. If you struggle to implement this, an example implementation is provided in the repository in the file `sample_data_snow.R`.