OLCF and NICS/RDAV Tutorial: Parallel R and High Performance Computing

Prepared by Amy Szczepanski, Pragnesh Patel, and George Ostrouchov.

http://olcf.ornl.gov/ and http://rdav.nics.tennessee.edu/

1. You can download R from www.r-project.org. Installation will depend on your OS. You will also need packages multicore, foreach, doMC, doSNOW, and pnmath. All of these except the last are available at www.r-project.org. The tar file for the llast one is at http://www.stat.uiowa.edu/∼luke/R/experimental/pnmath_0.0-3.tar.gz. Download it and install from within R by install.packages(pkgs=''pnmath_0.0-3.tar.gz'').

   Launch R on your computer.

2. Last time there was a request for more ggplot2 stat examples. Here are a few to try on your own:

```
ggplot(diamonds, mapping=aes(carat, price)) + geom_point(size=1) +
     stat_bin2d(alpha=I(1/1.5))
ggplot(diamonds, mapping=aes(carat, price)) + geom_point(size=1) +
     stat_bin2d(alpha=I(1/1.5)) + coord_polar()
ggplot(diamonds, mapping=aes(carat, price)) + geom_point(size=1) +
     stat_bin2d(alpha=I(1/1.5)) + scale_x_log10() + scale_y_log10()
ggplot(diamonds, mapping=aes(price,fill=clarity)) + geom_histogram()
ggplot(diamonds, mapping=aes(carat, price)) + geom_point() + stat_density2d() +
     scale_x_log10() + scale_y_log10()
ggplot(diamonds, mapping=aes(carat, price)) + stat_binhex(bins=50) +
     scale_x_log10() + scale_y_log10()
ggplot(diamonds, mapping=aes(depth,table,color=cut)) + geom_point() +
     stat_density2d(aes(x=depth,y=table,color="1"))
ggplot(diamonds, mapping=aes(depth,table)) + stat_binhex() + facet_wrap(~cut)
```

3. There was another question about getting your data into R. This is answered in a most comprehensive way in the "R Data Import/Export" manual at http://cran.r-project.org/manuals.html.

4. Today we will be learning:

   - A bit about interpreter nature of R.
   - A bit about timing and profiling.
   - Parallelizing without rewriting.
   - Parallelizing with minor rewriting.
   - Some words about R with MPI or clusters.
   - Some words R with GPU and with large memory.

5. R is an interpreted language, like python or matlab, so anything involving the interpreter is necessarily slow compared to compiled languages. However, many functions and some operators are calls to compiled code, so they are generally just as fast as compiled code.

   The first thing to consider in speeding up an R code is conversion of any loops, especially nested loops, into matrix expressions and matrix functions. A good source of suggestions is http://manuals.bioinformatics.ucr.edu/home/programming-in-r#Progr_noloops. Consider this example:

```
myMA <- matrix(rnorm(1000000), 100000, 10)
```

```
system.time(myMAmean <- apply(myMA, 1, mean)); myMAmean[1:4]
system.time(myMAmean <- rowMeans(myMA)); myMAmean[1:4]
```

6. Nautilus is a 1024 core SMP so many of these examples will run much the same way on Nautilus as they would on your own multicore laptop.

   Lens is a cluster of 32 16-core nodes for a total of 512 cores.

   I will use 16 cores on Lens and my 2-core laptop for all my examples.

7. On lens, I am going to allocate 1 node with 16 cores to run my R scripts.

   ```
   qsub -A chargeaccount -I -X -lnodes=1:ppn=16,walltime=1:00:00
   ```

   The -I flag make the allocation interactive and puts me in a shell on the node. The -X flag allows X11 forwarding, and walltime requests 1 hour.

   To make R available, we

   ```
   module load r
   ```

8. To compare the performance of parallel and serial calculations, we will need to quantify performance. The easiest way to find out how long it takes an R function to run is with the function `system.time()`.

9. Speed-ups for free: If our installation of R is compiled with Intel's MKL library or AMD's ACML library , we'll automatically get multi-threaded implementations of standard linear algebra functions. Here is an example of an R script that demonstrates this. This script will create a matrix X filled with random numbers from a normal distribution and will calculate X^tX two ways:

   (a) Using the matrix multiplication operation `%*%` to multiply X^t by X.

   (b) Using the statistics function `crossprod()`.

   Then it calls the `all.equal()` function to make sure that it got the same answer from all three calculations.

   ```
   its = 2500
   dim = 1750
   X = matrix(rnorm(its*dim),its, dim)

   # BLAS matrix mult
   system.time({C1 = t(X) %*% X})

   # BLAS matrix mult
   system.time({C2 = crossprod(X)})

   print(all.equal(C,C1,C2))
   ```

10. In addition to using `system.time()` we can use `Rprof` to profile our code. Here is the same code, slightly rewritten.

    ```
    its = 2500
    dim = 1750
    X = matrix(rnorm(its*dim),its, dim)

    my.cross.prod <- function(X)
    {
    ```

```
    C = matrix(0, ncol(X), ncol(X))
    for(i in 1:nrow(X))
    {
        C = C + X[i,] %o% X[i,]
    }
    return(C)
}

Rprof("matrix-mult.out")
C  = my.cross.prod(X)

C1 = t(X) %*% X

C2 = crossprod(X)
Rprof(NULL)

print(all.equal(C,C1,C2))
quit(save="no")
```

After we run this code, we have a fille called **matrix-mult.out** that has the timing information in it. We would use `R CMD Rprof matrix-mult.out` to see which functions we spend the most time in.

11. Another easy way to get a speed-up is to use the pnmath package in R. This package takes many of the standard math functions in R and replaces them with multi-threaded versions, using OpenMP. Here's a quick example of some code that you could run in an interactive R session to try this out. It creates two vectors of random numbers (uniform distribution) and applies some math functions to them. Some functions get more of a speed-up than others with pnmath.

```
v1 <- runif(1000)
v2 <- runif(100000000)

system.time(qtukey(v1,2,3))
system.time(exp(v2))
system.time(sqrt(v2))

library(pnmath)
system.time(qtukey(v1,2,3))
system.time(exp(v2))
system.time(sqrt(v2))
```

12. Here are some timings of some R functions using pnmath on 4 and 8 cores on Nautilus:

Function	Four cores	Eight cores
`sqrt()`	0.207	0.199
`exp()`	0.069	0
`dnorm()`	0.055	0.001
`lgamma()`	0.013	0.013
`dpois()`	0	0
`df()`	0.019	0.009
`pt()`	0.004	0.002
`qchisq()`	0.04	0.02
`psigamma()`	0.007	0.004
`qbeta()`	0.062	0.031
`qnchisq()`	5.356	2.683
`ptukey()`	61.436	30.602
`qtukey()`	334.185	165.328

13. When using pnmath, we can set the number of threads in our R script with `setNumPnmathThreads(8)`.

14. Some methods of running R in parallel require us to write our code in a certain way. We'll look at loop parallelism next. Once by applying a function to every item in a list and then at parallel `for()` loops. The pnmath package is still experimental and interacts badly with the `fork()` mechanism (our next topic), so I will restart my R session to unload the pnmath library.

15. The multicore package has a function `mclapply()` that allows us to apply a function to every item in a list (**m**ulti**c**ore **l**ist **apply**). It also has the function `pvec()` that allows us to apply a function to every element of a vector. We load this package with `library(multicore)`. We can use the following commands to check how many cores are available, to set the number of cores to use, or to see how many cores we are using:

    ```
    library(multicore)
    multicore:::detectCores()
    options(cores = 8)
    getOption('cores')
    ```

    Note that multicore relies on `fork()` and spawns new processes.

16. A simple example with `mclapply()` or `pvec()`:

    ```
    x <- mclapply(1:1000, sqrt)
    y <- pvec(1:1000, sqrt)
    ```

    The main difference between these two has to do with assumptions on how the calculation can be applied to the vector.

17. A slightly more complicated example, comparing `mclapply()` to `lapply()`:

    ```
    library(ggplot2)
    library(multicore)

    rdata <- function(n, par)
    {
      x <- 2*runif(n) - 1
      m <- 0 + 1*x + 2*x^2 + 3*x^3
      y <- exp(m)
      z <- round(exp(m + rnorm(n, sd=par)), 0)
      data.frame(list(y=y, z=z, x1=x, x2=x^2, x3=x^3))
    }
    ```

```
fitmodel <-  function(dat)
  {
    mod <- glm(z ~ x1 + x2 + x3, data=dat, family=poisson)
    mod$coefficients
  }

obs <- lapply(rep(10000, 100), rdata, par=2)

ggplot(data=obs[[1]]) + geom_point(aes(x1, z)) +
  geom_line(aes(x1, y), color="red") + scale_y_log10()

system.time(results <- lapply(obs, fitmodel))

system.time(resultsmc <- mclapply(obs, fitmodel))
```

18. This package also has a `parallel()` and `collect()` construct.

19. The next level of complication is the foreach package. With this package, you can have a `for()` loop run in parallel. It requires you to tell R which method of parallelization you want to use. There are many options, including multicore (which we just saw), SMP, Rmpi, snow, and others. You tell R that you want the loop to run in parallel by specifying `%dopar%`. For it to run in serial, you would say `%do%`. Here is an illustration of the multicore and snow options.

```
library(doMC)
library(foreach)

registerDoMC()

foreach(i=1:16) %dopar%
{
    test <- rnorm(100000)
    loess.smooth(test, test)
}
```

```
library(doSNOW)
library(foreach)

registerDoSNOW(makeCluster(2, type = "SOCK"))

foreach(i=1:16) %dopar%
{
    test <- rnorm(100000)
    loess.smooth(test, test)
}
```

20. Other ways of parallelizing code in R include snow (**s**imple **n**etwork **o**f **w**orkstations) and Rmpi. With snow, you define a **cluster** and then can use functions such as `clusterApply()` to have each node in a cluster apply a function to an item from a list.

21. There is a GSoC project in the works to use OpenMP directives in R. OpenMP in FORTRAN and in C directs the compiler on how to produce parallel SMP code. In a similar way, this GSoC project develops a pre-compiler for R to use OpenMP directives to produce R code instrumented with some of the tools we just discussed.

22. Although there is a 64-bit version of R, there is a problem: The indexing is limited to 32-bit integers. One way to deal with this is to split up the data across processors with separate R sessions that communicate via MPI.

23. Rmpi is pretty much what it sounds like. For some clusters, a combination of spreading loop iterations out to the nodes and using pnmath so that each node runs multithreaded math operations can be effective. Here is a simple Hello World example:

```
library(Rmpi, quietly=TRUE)

rank <- mpi.comm.rank(0)
size <- mpi.comm.size(0)
```

```
cat("Hello World from", rank, ":", size, "\n")

mpi.quit(save = "no")
```

I saved this script as `HelloWorldBatch.R` and I will run it with `orterun Rscript HelloWorldBatch.R`

24. The gputools package provides R interfaces to handful common statistical algorithms. They are implemented using mixture of CUDA language, CUBLAS library and CULA library. It contains many other functions, including hierarchical clustering, SVM training, SVD, Least-squares fit, linear modeling, and many others. Less-communicative algorithms seeing speedups over 20 times on data set of moderate size, but speed up factors vary with CPU, memory configurations and, of course, GPU. Here is an example:

```
library(gputools)
matA <- matrix(runif(3*2), 3, 2)
matB <- matrix(runif(3*4), 3, 4)
gpuCrossprod(matA, matB) # Perform Matrix Cross-product with a GPU

numVectors <- 5
dimension <- 10
Vectors <- matrix(runif(numVectors*dimension), >numVectors, dimension)
gpuDist(Vectors, "euclidean")
gpuDist(Vectors, "maximum")
gpuDist(Vectors, "manhattan")
gpuDist(Vectors, "minkowski", 4)
```

25. For working with large data on SMP machines, consider the packages bigmemory, biganalytics, bigalgebra, and bigtabulate. The data structures may be allocated to shared memory, allowing separate processes on the same computer share access to single copy of the date set. The data structures may also be file-backend allowing users to easily manage and analyze data sets larger than available RAM and share them across nodes of a cluster. This generally needs a parallel file system for efficiency.

26. Summary of these packages:

    bigmemory: supports the creation, manipulation and storage of large matrices.

    bigalgebra: provides linear algebra functionality with large matrices.

    biganalytics: extends the functionality of bigmemory.

    bigtabulate: supports `table()`, `split()` and `tapply()` like functionality for large matrices.

    foreach + bigmemory: a winning combination for massive data SMP programming.

27. Here is an example that uses a very, very large matrix. This example illustrates how to work with a matrix with more than $2^{31} - 1$ elements.

```
# big.matrix: no 2^31-1 object size limitation.

library(bigmemory)
R <- 3e9 # 3 billion rows
C <- 2 # 2 columns

print("48 GB total size:")
R*C*8 # 48 GB total size
```

```
date()

x <- filebacked.big.matrix(R, C, type='double',backingfile='huge-data.bin',
descriptorfile='huge-data.desc')
## Generates huge-data.bin and huge-data.desc files.
## Now we can use huge-data.desc file in any R session.

x[1,] <- rnorm(C)
x[nrow(x),] <- runif(C)
summary(x[1,])

summary(x[nrow(x),])

date()
```

Note: This example *will* leave a 48 GB file on your hard drive!

28. Here is another series of examples that work with a large data set. This uses the airline data set. You can download the data from `http://stat-computing.org/dataexpo/2009/the-data.html`. This example is based on the year 2008 data, which comes in the file **2008.csv.bz2** and can be extracted with `bzip2 -d *.bz2`, giving you the 659 MB spreadsheet.

    This script will create a binary file-backing for this matrix, which we name **airline.bin** and a descriptor file that we name **airline.desc**. These files allow R to access the large data set more quickly.

```
library(bigmemory)
library(biganalytics)

x <- read.big.matrix("2008.csv", type="integer", header=TRUE,
  backingfile="airline.bin", descriptorfile="airline.desc", extraCols="Age")

summary(x)
```

29. This next script will let us use this backing file to access the data quickly in later R sessions, without having to take several minutes to read the spreadsheet.

```
library(bigmemory)
library(biganalytics)

xdesc<- dget("airline.desc") ## we do not need to read all data from csv again.
x<-attach.big.matrix(xdesc)

system.time(numplanes<-colmax(x,"TailNum", na.rm=TRUE))
system.time(numplanes<-colmax(x,"TailNum", na.rm=TRUE))

system.time(colmin(x, 1))
system.time(a <- x[,1])
system.time(a <- x[,2])

colnames(x)
tail(x, 1)
```

30. A great reference for R in an HPC environment is:
    `http://dirk.eddelbuettel.com/papers/useR2009hpcTutorial.pdf`