

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

George Ostrouchov^{†‡}, Pragnesh Patel[‡], and Drew Schmidt[‡]

June 29, 2012

Contents

1	Introduction	1
2	Token Activation	2
3	Login and Architecture of Nautilus	2
4	Overview of how to run R via a batch system	2
5	Kinds of parallelization and R limitations	2
6	Implicit Parallelism	3
6.1	pnmath	3
6.2	BLAS	4
6.3	Exercises	4
7	Explicit Parallelism	4
7.1	multicore	4
7.2	snow	5
7.3	parallel	6
7.4	Rmpi	6
7.5	foreach and do*	7
7.6	Exercises	8
8	Other important R-HPC tools	8
9	Where to Learn More?	8
10	Suggested Solutions to Exercises	9

1 Introduction

First many thanks to Amy Szczepanski for all of her useful tutorial documents. Welcome to the beginning of your journey into parallel R. Today we will cover:

- Token activation
- Login and Architecture of Nautilus
- Overview of how to run R via a batch system
- Kinds of parallelization and R limitations
- Implicit Parallelism
- Explicit Parallelism
- Other important R-HPC tools

*Copyright © 2012 George Ostrouchov, Pragnesh Patel, and Drew Schmidt. All Rights Reserved.

[†]Oak Ridge National Laboratory, Computer Science and Mathematics Division, Scientific Data Group

[‡]National Institute for Computational Sciences, Remote Analysis and Visualization Center

2 Token Activation

Switch to slide

3 Login and Architecture of Nautilus

Switch to slide

4 Overview of how to run R via a batch system

I'll likely be running most of these examples on Nautilus. Most work on HPC systems is done via scripts submitted at the command line. Furthermore, we submit these jobs through a batch environment. On Nautilus we use Moab with TORQUE. Here is an example of typical PBS (generic term for batch system) script for Nautilus.

```
1 #!/bin/bash
2 #PBS -N myjobname
3 #PBS -q analysis
4 #PBS -j oe
5 #PBS -l ncpus=8
6 #PBS -l mem=32000MB
7 #PBS -l walltime=00:10:00
8
9 module load r
10
11 cd $PBS_O_WORKDIR
12 Rscript myscript.R
```

Rjobsubmit.pbs

It asks for 8 CPUs and 32000MB (<32 GB to allow room for OS processes) RAM for 10 minutes in the analysis queue. On Nautilus, jobs in the analysis queue can pre-empt jobs in the computation queue. We would save this script into a file named *Rjobsubmit.pbs* and submit it to the queue with the command *qsub Rjobsubmit.pbs*. The system would then place your job in the queue and run it once it reaches the front of the line. Once it runs, it would save all of its output in a file named by the job name. This script is assuming that everything happens from the job submission directory (PBS_O_WORKDIR). In some of our scripts, we'll need to set environment variables to control threaded behavior. We'll typically insert those lines after *module load r*.

5 Kinds of parallelization and R limitations

There are four different kinds of parallelization:

- Bit-based parallelization: The move up the chain via 4/8/16/32/64 bit machines changes the number of steps required to run a single instruction.
- Instruction-based parallelization: Processor/program layer.
- Data-based parallelization: Decompose large data structures into independent chunks, on which you perform the same operation.
- Task-based parallelization: Perform different, independent tasks on the same data.

Two primary drivers for the increased focus on high performance computing with R can be identified: larger data sets, and increased computational requirements stemming from more sophisticated methodologies. For R, we are mostly interested in data and task parallelization.

However, R has two major barriers to achieving these goals:

- **It's single-threaded:** The R language has no explicit constructs for parallelism, such as threads or mutexes. An out-of-the-box R installation cannot take advantage of multiple CPUs.
- **It's memory-bound:** R requires that your entire dataset fit in memory (RAM). Four gigabytes of RAM will not hold eight gigabytes of data, no matter how much you smile when you ask.

There are a lot of different ways to solve performance issues. The solution mostly depends on the specific problem. Well known techniques to improve R code include: Vectorized computation, Parallel apply functions, Other language interfaces (C, C++, FORTRAN etc...).

6 Implicit Parallelism

6.1 pnmath

Some speedups are free—our installation of R on Nautilus is available with the Intel MKL library which includes multi-threaded version of many math functions, such as matrix arithmetic, gaussian elimination, cholesky decomposition, etc.

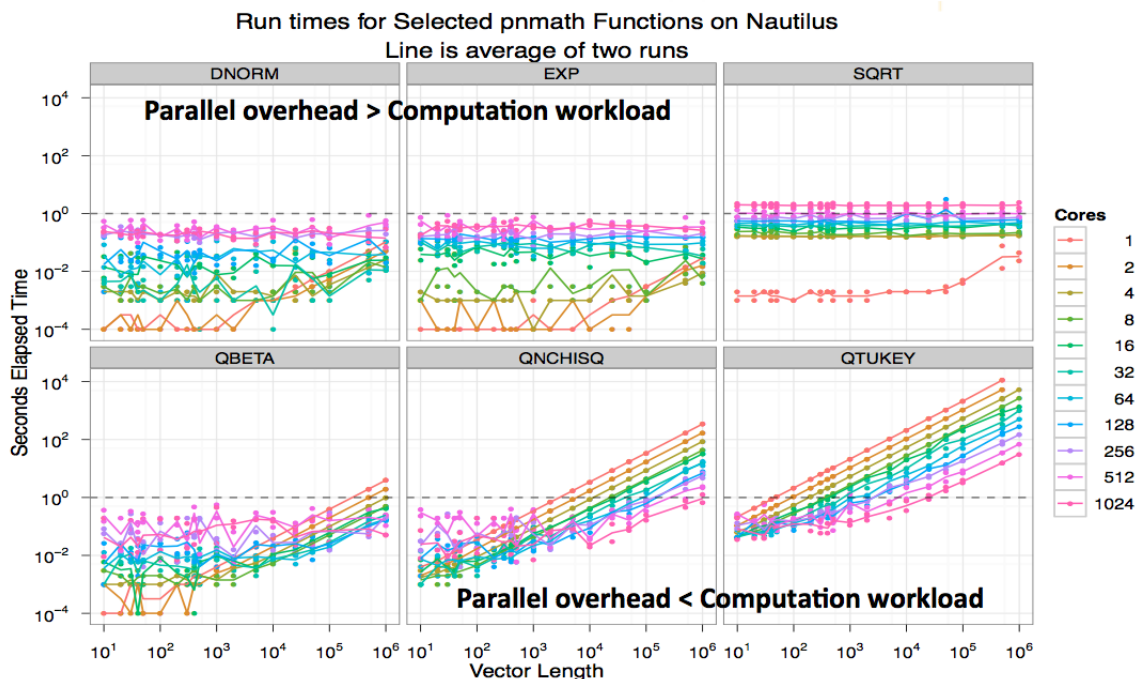
Today multi-core systems are a standard for every workstation, and the number of processors per chip is growing. There is a compelling need for integration of R code into multi-core environments. The *pnmath* package is a first step in this direction. The *pnmath* package takes many of the standard math functions in R and replaces them with multi-threaded versions using OpenMP. It uses the OpenMP parallel processing directives for implicit parallelism. Loading the package automatically replaces the built-in math functions by the parallel versions. At load time, a calibration is carried out to determine the parallel overhead. *pnmath* implements parallelized versions of most of the non-RNG routines in the math library. Once the package is loaded, the user employs familiar R functions while *pnmath* runs parallel versions behind the scenes.

When using *pnmath*, we can set number of threads in our R script with *setNumPnmathThreads(N)* R function or we can set the number of threads with an environment variable in PBS script with *OMP_NUM_THREADS=N*. The default version of this package was written to run on, at most eight cores; running on more than eight cores requires some tweaks to the *pnmath* package.

Below is a quick example that you could run in an interactive R session to try this package. It creates two vectors of random numbers (uniform distribution) and applies various math functions to them. Some functions get more of a speed-up than others (depends on parallel overhead and computation workload).

```
1 v1 <- runif(1000)
2 v2 <- runif(100000000)
3 # Using default nmath library
4 print("Serial nmath version timing:")
5 system.time(exp(v2))
6 system.time(sqrt(v2))
7 system.time(qtukey(v1,2,3))
8
9 # Using pnmath library
10 print("Parallel nmath timing:")
11 library("pnmath") #uses OpenMP
12 setNumPnmathThreads(4)
13 print(sprintf("Number of threads: %d", getNumPnmathThreads()))
14 system.time(exp(v2))
15 system.time(sqrt(v2))
16 system.time(qtukey(v1,2,3))
17 # To run this program on four cores on Nautilus:
18 # For pnmath
19 #-----
20 # export OMP_NUM_THREADS=4(in PBS script)
21 # OR
22 # setNumPnmathThreads(4)
```

pnmath_example.R



6.2 BLAS

Doing a lot of matrix math? You can build R against a multithreaded Basic Linear Algebra Subprogram (BLAS). R uses unoptimized routines to do linear algebra if not linked with external BLAS. As I mentioned before, our R is configured with the Intel MKL.

You can compile and link R with different BLAS libraries:

- *Netlib-BLAS*
- *GotoBLAS2*
- *Intel MKL*
- *AMD ACML*

Here is an example R script that demonstrates R-MKL.

```
1 # Here is code that creates a matrix x filled with random numbers
   from a normal distribution and calculates x(t) x three ways:
2
3 # (1) Element by element, using the rule that we use "by hand"
   with %o% for the dot product.
4 # (2) Using the matrix multiplication operation %o% to multiply
   x(t) by x.
5 # (3) Using the statistics function crossprod(x).
6
7 its = 2500
8 dim = 1750
9 x <- matrix(rnorm(its*dim),its, dim)
10
11 # Single threaded breakup calculation
12 print("single threaded- breakup calculation")
13 system.time({
14     result1 <- matrix(0,dim,dim)
15     for (i in 1:its)
16         result1 <- result1 + x[i,] %o% x[i,]
17 })
18
19 print("multithreaded- BLAS matrix mult with 8 threads using %*% ")
20 system.time({result2 <- t(x) %*% x})
21
22 print("multithreaded- BLAS matrix mult with 8 threads using
   crossprod")
23 system.time({result3 <- crossprod(x)})
24
25 # To verify your result from all three calculations.
26 print(all.equal(result1, result2, result3))
27
28
29 #To run this program on eight cores on Nautilus, you need to set
   following variables in PBS script:
30 #export MKL_NUM_THREADS=8
31 #export MKL_DYNAMICS=FALSE
```

mkl.example_multithreaded.R

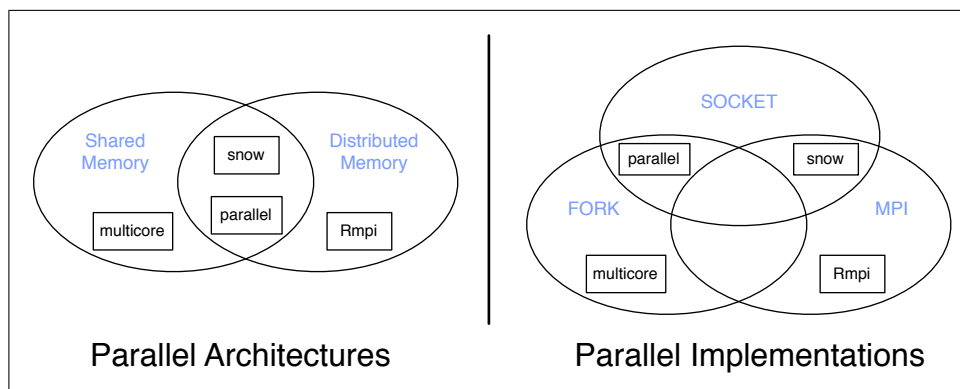
6.3 Exercises

1. Write a program to call *dnorm*, *qbeta* and *ptukey* functions from *pnmath* and measure performance with 2, 4 and 8 threads. You can try with different sizes of vectors.
2. Create a program to use multithreaded Cholesky Factorization, Singular Value Decomposition and Principal Components Analysis and compare performance using 4 and 8 threads (Tips: *?chol*, *?svd*, *?prcomp*).

7 Explicit Parallelism

7.1 multicore

This package provides a way of running parallel computations in R on machines with multiple cores or CPUs (e.g. Nautilus). *multicore* is a popular parallel programming package for use on multiprocessor and multicore computers. It is written by Simon Urbanek. It relies on the system's *fork* for parallelism. Because *fork()* is a Posix system call, *multicore* can't really



be used on Windows machines. *Fork()* can also cause problems for functions that use resources that were allocated or initialized exclusively for the master, or parent process. This is particularly a problem with graphics functions, so it isn't generally recommended to use multicore with an R GUI. Nevertheless, multicore works perfectly for most R functions on Posix systems, such as Linux and Mac OS X, and its use of *fork()* makes it very efficient and convenient.

Solves: Single-threaded.

Pros: Simple and efficient; easy to install; no configuration needed.

Cons: Can only use one machine; doesn't support Windows;

Pivotal functions:

- *mclapply* - parallelized version of *lapply*
- *pvec* - parallelization of vectorized functions
- *parallel* and *collect* - functions to evaluate R expressions in parallel and collect the results.

```
1 library(multicore)
2 options(cores = 2) # Specify number of workers
3 mclapply(1:10, function(i) rnorm(10), mc.cores = 2) # the workers
   will produce different values
```

multicore_example.R

7.2 snow

snow(Simple Network Of Workstations): Good for use on traditional clusters, especially if MPI is available. It supports MPI(Message Passing Interface), PVM(Parallel Virtual Machine), nws(NetWorkSpaces), and sockets for communication, and is quite portable, running on Linux, Mac OS X, and Windows.

snow provides support for easily executing R functions in parallel. Most of the parallel execution functions in *snow* are variations of the standard *lapply()* function, making it fairly easy to learn. To implement these parallel operations, *snow* uses a master/worker architecture, where the master sends tasks to the workers, and the workers execute the tasks and return the results to the master.

snow can be used with socket connections, MPI, PVM, or NetWorkSpaces. The socket transport doesn't require any additional packages, and is the most portable. MPI is supported via the *Rmpi* package, PVM via *rpvm*, and NetWorkSpaces via *nws*.

snow doesn't provide mechanisms for dealing with large data, such as distributing data files to the workers. The input arguments must fit into memory when calling a snow function, and all of the task results are kept in memory on the master until they are returned to the caller in a list.

Solves: Single-threaded, memory-bound.

Pros: Mature, popular package; leverages MPI's speed without its complexity.

Cons: Can be difficult to configure.

Pivotal functions:

- *makeCluster* - Initialize slave R processes
- *stopCluster* - Stop cluster
- *parApply* - the parallel version of the R function *apply*.
- *parLapply* - the parallel version of the R function *lapply*.
- ...

```

1 library(snow)
2 #Create a list of two sequences of numbers
3 params <- list(alpha = 1:10, beta = exp(-3:3))
4
5 cl <- makeCluster(4, type="SOCK") #create a cluster of four workers
   on the local machine using the socket transport.
6 #cl <- makeCluster(4, type="MPI") ##create a cluster of four
   workers using MPI.
7
8 #Calculate quantiles for each sequence - parSapply returns a matrix
9 res <- parSapply(cl, params, quantile)
10 res

```

snow_simple_example.R

7.3 parallel

A merger of *multicore* and *snow* (but excluding MPI, PVM and NWS clusters) that comes built into R as of R 2.14.0 and greater.

Solves: Single-threaded, memory-bound.

Pros: No installation necessary; has great support for parallel random number generation.

Cons: Can only use one machine on Windows; can be difficult to configure on multiple Linux machines.

Pivotal functions:

- combination of *multicore* and *snow* pivotal functions

```

1 ## Parallel K-means example
2 library(parallel)
3 library(MASS)
4 RNGkind("L'Ecuyer-CMRG")
5 mc.cores <- detectCores()
6 results <- mclapply(
7   rep(25, 4),
8   function(nstart) kmeans(Boston, 4, nstart=nstart),
9   mc.cores=mc.cores)
10
11 i <- sapply(results, function(result) result$tot.withinss)
12 result <- results[[which.min(i)]]

```

parallel_example_kmeans.R

7.4 Rmpi

Rmpi provides an interface between R and the Message Passing Interface (MPI), a standard for parallel computing. It allows us to use MPI directly from R.

```

1 # Load the R MPI package if it is not already loaded
2 if (!is.loaded("mpi_initialize")) {
3   library("Rmpi")
4 }
5
6 # Spawn as many slaves as possible
7 mpi.spawn.Rslaves(nslaves=8)
8
9 # In case R exits unexpectedly, have it automatically clean up
10 # resources taken up by Rmpi (slaves, memory, etc...)
11 .Last <- function(){
12   if (is.loaded("mpi_initialize")){
13     if (mpi.comm.size(1) > 0) {
14       print("Please use mpi.close.Rslaves() to close slaves.")
15       mpi.close.Rslaves()
16     }
17     print("Please use mpi.quit() to quit R")
18     .Call("mpi_finalize")
19   }
20 }
21

```

```

22 # Tell all slaves to return a message identifying themselves
23 mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
24
25 # Tell all slaves to close down, and exit the program
26 mpi.close.Rslaves()
27 mpi.quit()

```

Rmpi_simple_example.R

R's parallel methods concentrate on bringing parallelism to interactive computation. This includes Rmpi and its methods for spawning workers. It is because of R's strong emphasis on interactive data analysis. On the other hand, much of the parallel computing community works with SPMD(single process, multiple data; or single program, multiple data) type of batch parallelism. We are currently working on developing some R packages with tools that target SPMD batch programming in R. Some of them will be released by this Fall.

7.5 foreach and do*

Foreach is an idiom that allows for iterating over elements in a collection, without the use of an explicit loop counter. The *foreach()* function executes an arbitrary R expression across an input. *foreach()*'s strength is that it can execute in parallel with the help of a supplied parallel backend so that you can write one version of the code, and execute it with Rmpi, snow, etc.

Parallel computation depends upon a parallel backend that must be registered before performing the computation. The parallel backends available will be system-specific, but include *doParallel*, *doMC*, *doMPI*, and *doSNOW*.

- *doMC* uses the *multicore* package.
- *doSNOW* uses the *snow* package.
- *doParallel* uses the *parallel* package.
- *doMPI* uses the *Rmpi* package.

The *foreach* and *%do%/%dopar%* operators provide a looping construct that can be viewed as a hybrid of the standard *for* loop and *lapply* function. It looks similar to the *for* loop, and it evaluates an expression, rather than a function (as in *lapply*), but it's purpose is to return a value (a list, by default), rather than to cause side-effects. This facilitates parallelization, but looks more natural to people that prefer *for* loops to *lapply*.

```

1 library(foreach)
2
3 # Simple example
4 foreach (j=1:4) %do% { j }
5
6 foreach (j=1:4,.combine=c) %do% { j }
7
8 foreach (j=1:4,.combine='+') %do% { j }

```

foreach_example.R

```

1 library(foreach)
2 library(doMC)
3 registerDoMC(2) #register the parallel backend with the foreach
                  package and set the number of cores to use for parallel
                  execution.
4 getDoParWorkers() #returns the number of execution workers.
5 getDoParRegistered() #returns TRUE if a doPar backend has been
                       registered, otherwise FALSE.
6 getDoParName() #returns the name of the currently registered doPar
                 backend.
7 getDoParVersion() #returns the version of the currently registered
                    doPar backend.
8
9 # Simple example
10 foreach (j=1:4) %dopar% { j }
11
12 foreach (j=1:4,.combine=c) %dopar% { j }
13
14 foreach (j=1:4,.combine='+') %dopar% { j }

```

doMC_example.R

The *doParallel* package provides a parallel backend for the `foreach %dopar%` function using the *parallel* package of R 2.14.0 and later. *parallel* package is useful for parallel execution of R code on machines with multiple cores or processors or multiple computers. It is essentially a blend of the *snow* and *multicore* packages. By default, the *doParallel* package uses *snow*-like functionality on Windows systems and *multicore*-like functionality on Unix-like systems.

```
1 library(doSNOW)
2 library(foreach)
3
4 registerDoSNOW(makeCluster(2, type = "SOCK"))
5 getDoParWorkers() #returns the number of execution workers.
6 getDoParRegistered() #returns TRUE if a doPar backend has been
   registered, otherwise FALSE.
7 getDoParName() #returns the name of the currently registered doPar
   backend.
8 getDoParVersion() #returns the version of the currently registered
   doPar backend.
9
10 foreach(i=1:16) %dopar%{
11     test <- rnorm(100000)
12     loess.smooth(test, test)
13 }
```

doSNOW_example.R

7.6 Exercises

1. Write a program to perform summation of vector size 10000000 using *foreach* and *doMC* and compare performance of serial vs parallel implementation (using 8 cores).

8 Other important R-HPC tools

Switch to slide

9 Where to Learn More?

- HPC and Parallel Computing with R: <http://cran.r-project.org/web/views/HighPerformanceComputing.html>
- *pnmath*: <http://homepage.stat.uiowa.edu/luke/R/experimental/>
- Multithreaded BLAS: <http://blog.revolutionanalytics.com/2010/06/performance-benefits-of-multithreaded-r.html>
- *multicore*: <http://cran.r-project.org/web/packages/multicore/multicore.pdf>
- *snow*: <http://cran.r-project.org/web/packages/snow/snow.pdf>
- *parallel*: <http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf>
- *Rmpi*: <http://math.acadiau.ca/ACMMaC/Rmpi/>
- *foreach*: <http://cran.r-project.org/web/packages/foreach/foreach.pdf>
- *doMC*: <http://cran.r-project.org/web/packages/doMC/doMC.pdf>
- *doSNOW*: <http://cran.r-project.org/web/packages/doSNOW/doSNOW.pdf>
- *doParallel*: <http://cran.r-project.org/web/packages/doParallel/doParallel.pdf>
- *doMPI*: <http://cran.r-project.org/web/packages/doMPI/vignettes/doMPI.pdf>
- Rprof: <http://stat.ethz.ch/R-manual/R-patched/library/utils/html/Rprof.html>
- gputools: <http://cran.r-project.org/web/packages/gputools/index.html>
- R magma: <http://cran.r-project.org/web/packages/magma/index.html>
- R and hadoopism: <https://github.com/RevolutionAnalytics/RHadoop/wiki/rmr>
- RHIPE: <http://www.datadr.org/>
- seque: <http://code.google.com/p/seque/>

- Byte Code Compiler: <http://stat.ethz.ch/R-manual/R-devel/library/compiler/html/compile.html>
- Rcpp: <http://dirk.eddelbuettel.com/code/rcpp.html>
- inline: <http://cran.r-project.org/web/packages/inline//index.html>

10 Suggested Solutions to Exercises

Download from <http://www.nics.tennessee.edu/computing-resources/nautilus/software?&software=r>
OR

Copy */lustre/medusa/pragnesh/R_parallel_session3.zip* file to your lustre directory(*/lustre/medusa/\$USER*)