

R

Introduction

The R is a language and environment for statistical computing and graphics. R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible.

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

Another convenience is the ease with which the C code or third party libraries may be integrated within R.

Extensive support for parallel computing is available within R.

Read more on <http://www.r-project.org/>, <http://cran.r-project.org/doc/manuals/r-release/R-lang.html>

Modules

**The R version 3.0.1 is available on Anselm, along with GUI interface Rstudio

Application	Version	module
R	R 3.0.1	R
Rstudio	Rstudio 0.97	Rstudio

```
$ module load R
```

Execution

The R on Anselm is linked to highly optimized MKL mathematical library. This provides threaded parallelization to many R kernels, notably the linear algebra subroutines. The R runs these heavy calculation kernels without any penalty. By default, the R would parallelize to 16 threads. You may control the threads by setting the OMP_NUM_THREADS environment variable.

Interactive execution

To run R interactively, using Rstudio GUI, log in with ssh -X parameter for X11 forwarding. Run rstudio:

```
$ module load Rstudio
$ rstudio
```

Batch execution

To run R in batch mode, write an R script, then write a bash jobscript and execute via the qsub command. By default, R will use 16 threads when running MKL kernels.

Example jobscript:

```
#!/bin/bash

# change to local scratch directory
cd /lscratch/$PBS_JOBID || exit

# copy input file to scratch
cp $PBS_O_WORKDIR/rscript.R .

# load R module
module load R

# execute the calculation
R CMD BATCH rscript.R routput.out

# copy output file to home
cp routput.out $PBS_O_WORKDIR/.

#exit
exit
```

This script may be submitted directly to the PBS workload manager via the qsub command. The inputs are in rscript.R file, outputs in routput.out file. See the single node jobscript example in the Job execution section.

Parallel R

Parallel execution of R may be achieved in many ways. One approach is the implied parallelization due to linked libraries or specially enabled functions, as described above. In the following sections, we focus on explicit parallelization, where parallel constructs are directly stated within the R script.

Package parallel

The package parallel provides support for parallel computation, including by forking (taken from package multicore), by sockets (taken from package snow) and random-number generation.

The package is activated this way:

```
$ R
> library(parallel)
```

More information and examples may be obtained directly by reading the documentation available in R

```
> ?parallel
> library(help = "parallel")
> vignette("parallel")
```

Download the package parallel vignette.

The forking is the most simple to use. Forking family of functions provide parallelized, drop in replacement for the serial apply() family of functions.

Forking via package parallel provides functionality similar to OpenMP construct `#omp parallel for`

Only cores of single node can be utilized this way!

Forking example:

```
library(parallel)

#integrand function
f <- function(i,h) {
  x <- h*(i-0.5)
  return (4/(1 + x*x))
}

#initialize
size <- detectCores()

while (TRUE)
{
  #read number of intervals
  cat("Enter the number of intervals: (0 quits) ")
  fp<-file("stdin"); n<-scan(fp,nmax=1); close(fp)

  if(n<=0) break

  #run the calculation
```

```

n <- max(n,size)
h <- 1.0/n

i <- seq(1,n);
pi3 <- h*sum(simplify2array(mclapply(i,f,h,mc.cores=size)));

#print results
cat(sprintf("Value of PI %16.14f, diff= %16.14fn",pi3,pi3-pi))
}

```

The above example is the classic parallel example for calculating the number π . Note the **detectCores()** and **mclapply()** functions. Execute the example as:

```
$ R --slave --no-save --no-restore -f pi3p.R
```

Every evaluation of the integrand function runs in parallel on different process.

Package Rmpi

package Rmpi provides an interface (wrapper) to MPI APIs.

It also provides interactive R slave environment. On Anselm, Rmpi provides interface to the OpenMPI.

Read more on Rmpi at <http://cran.r-project.org/web/packages/Rmpi/>, reference manual is available at <http://cran.r-project.org/web/packages/Rmpi/Rmpi.pdf>

When using package Rmpi, both openmpi and R modules must be loaded

```
$ module load openmpi
$ module load R
```

Rmpi may be used in three basic ways. The static approach is identical to executing any other MPI program. In addition, there is Rslaves dynamic MPI approach and the mpi.apply approach. In the following section, we will use the number π integration example, to illustrate all these concepts.

static Rmpi

Static Rmpi programs are executed via mpiexec, as any other MPI programs. Number of processes is static - given at the launch time.

Static Rmpi example:

```

library(Rmpi)

#integrand function
f <- function(i,h) {

```

```

x <- h*(i-0.5)
return (4/(1 + x*x))
}

#initialize
invisible(mpi.comm.dup(0,1))
rank <- mpi.comm.rank()
size <- mpi.comm.size()
n<-0

while (TRUE)
{
  #read number of intervals
  if (rank==0) {
    cat("Enter the number of intervals: (0 quits) ")
    fp<-file("stdin"); n<-scan(fp,nmax=1); close(fp)
  }

  #broadcast the intervals
  n <- mpi.bcast(as.integer(n),type=1)

  if(n<=0) break

  #run the calculation
  n <- max(n,size)
  h <- 1.0/n

  i <- seq(rank+1,n,size);
  mypi <- h*sum(sapply(i,f,h));

  pi3 <- mpi.reduce(mypi)

  #print results
  if (rank==0) cat(sprintf("Value of PI %16.14f, diff= %16.14fn",pi3,pi3-pi))
}

mpi.quit()

```

The above is the static MPI example for calculating the number π . Note the **library(Rmpi)** and **mpi.comm.dup()** function calls. Execute the example as:

```
$ mpiexec R --slave --no-save --no-restore -f pi3.R
```

dynamic Rmpi

Dynamic Rmpi programs are executed by calling the R directly. openmpi module must be still loaded. The R slave processes will be spawned by a function call within the Rmpi program.

Dynamic Rmpi example:

```
#integrand function
f <- function(i,h) {
  x <- h*(i-0.5)
  return (4/(1 + x*x))
}

#the worker function
workerpi <- function()
{
  #initialize
  rank <- mpi.comm.rank()
  size <- mpi.comm.size()
  n<-0

  while (TRUE)
  {
    #read number of intervals
    if (rank==0) {
      cat("Enter the number of intervals: (0 quits) ")
      fp<-file("stdin"); n<-scan(fp,nmax=1); close(fp)
    }

    #broadcast the intervals
    n <- mpi.bcast(as.integer(n),type=1)

    if(n<=0) break

    #run the calculation
    n <- max(n,size)
    h <- 1.0/n

    i <- seq(rank+1,n,size);
    mypi <- h*sum(sapply(i,f,h));

    pi3 <- mpi.reduce(mypi)

    #print results
    if (rank==0) cat(sprintf("Value of PI %16.14f, diff= %16.14fn",pi3,pi3-pi))
```

```

}
}

#main
library(Rmpi)

cat("Enter the number of slaves: ")
fp<-file("stdin"); ns<-scan(fp,nmax=1); close(fp)

mpi.spawn.Rslaves(nslaves=ns)
mpi.bcast.Robj2slave(f)
mpi.bcast.Robj2slave(workerpi)

mpi.bcast.cmd(workerpi())
workerpi()

mpi.quit()

```

The above example is the dynamic MPI example for calculating the number . Both master and slave processes carry out the calculation. Note the `mpi.spawn.Rslaves()`, `mpi.bcast.Robj2slave()`** and the `mpi.bcast.cmd()`** function calls. Execute the example as:

```
$ R --slave --no-save --no-restore -f pi3Rslaves.R
```

mpi.apply Rmpi

`mpi.apply` is a specific way of executing Dynamic Rmpi programs.

`mpi.apply()` family of functions provide MPI parallelized, drop in replacement for the serial `apply()` family of functions.

Execution is identical to other dynamic Rmpi programs.

`mpi.apply` Rmpi example:

```

#integrand function
f <- function(i,h) {
  x <- h*(i-0.5)
  return (4/(1 + x*x))
}

#the worker function
workerpi <- function(rank,size,n)
{
  #run the calculation
  n <- max(n,size)
  h <- 1.0/n

```

```

    i <- seq(rank,n,size);
    mypi <- h*sum(sapply(i,f,h));

    return(mypi)
}

#main
library(Rmpi)

cat("Enter the number of slaves: ")
fp<-file("stdin"); ns<-scan(fp,nmax=1); close(fp)

mpi.spawn.Rslaves(nslaves=ns)
mpi.bcast.Robj2slave(f)
mpi.bcast.Robj2slave(workerpi)

while (TRUE)
{
    #read number of intervals
    cat("Enter the number of intervals: (0 quits) ")
    fp<-file("stdin"); n<-scan(fp,nmax=1); close(fp)
    if(n<=0) break

    #run workerpi
    i=seq(1,2*ns)
    pi3=sum(mpi.parSapply(i,workerpi,2*ns,n))

    #print results
    cat(sprintf("Value of PI %16.14f, diff= %16.14fn",pi3,pi3-pi))
}

mpi.quit()

```

The above is the mpi.apply MPI example for calculating the number π . Only the slave processes carry out the calculation. Note the mpi.parSapply(), ** function call. The package parallel exampleabove{.anchor may be trivially adapted (for much better performance) to this structure using the mclapply() in place of mpi.parSapply().

Execute the example as:

```
$ R --slave --no-save --no-restore -f pi3parSapply.R
```


Combining parallel and Rmpi

Currently, the two packages can not be combined for hybrid calculations.

Parallel execution

The R parallel jobs are executed via the PBS queue system exactly as any other parallel jobs. User must create an appropriate jobscript and submit via the **qsub**

Example jobscript for static Rmpi parallel R execution, running 1 process per core:

```
#!/bin/bash
#PBS -q qprod
#PBS -N Rjob
#PBS -l select=100:ncpus=16:mpiprocs=16:ompthreads=1

# change to scratch directory
SCRDIR=/scratch/$USER/myjob
cd $SCRDIR || exit

# copy input file to scratch
cp $PBS_O_WORKDIR/rscript.R .

# load R and openmpi module
module load R
module load openmpi

# execute the calculation
mpiexec -bycore -bind-to-core R --slave --no-save --no-restore -f rscript.R

# copy output file to home
cp routput.out $PBS_O_WORKDIR/.

#exit
exit
```

For more information about jobscripts and MPI execution refer to the Job submission and general MPI sections.