Parallel Processing with R

Amy F. Szczepański

Remote Data Analysis and Visualization Center
University of Tennessee, Knoxville

aszczepa@utk.edu

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

http://rdav.nics.tennessee.edu/

RDAV and Nautilus

Located at the National Institute for Computational Sciences (NICS) on the campus of the Oak Ridge National Laboratory

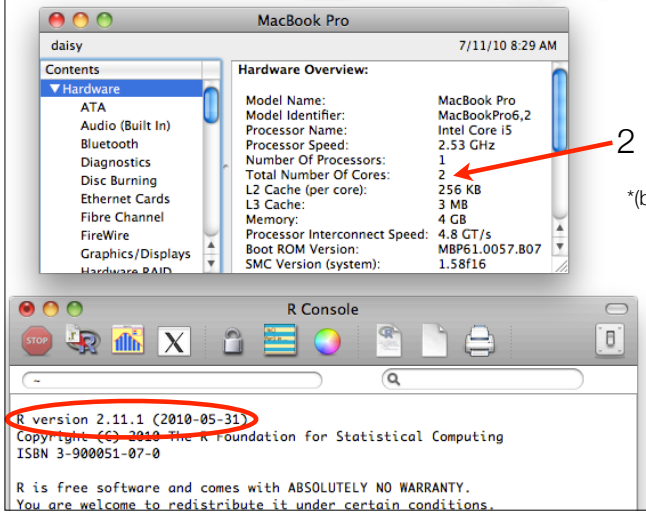
Kraken
Compute nodes
99,072 cores

NICS Network
IB and 10GigE

other NICS and TeraGrid resources

Nautilus
SGI UltraViolet SMP
1024 Intel Nehalem EX cores
16 GPUs
4 TB shared memory
Connects to 1PB filesystem

What I'm Running on my Laptop



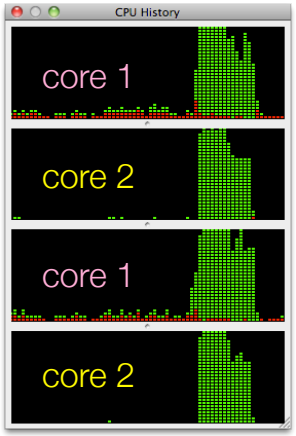
2 Cores*

*(but pretends to have four)

Some Options for R in Parallel

- use parallelized libraries for linear algebra
- `multicore` – Workstation with multiple cores
- `Rmpi` – Message Passing Interface
- `snow` – Simple Network of Workstations (won't discuss)
- `foreach` – Can use different back ends
- CRAN task view:
  High-Performance and Parallel Computing with R
  `http://cran.r-project.org/web/views/HighPerformanceComputing.html`

Parallel Linear Algebra



Matrix Multiply: $A^T A$

```
set.seed(1)
m <- 10000
n <- 5000
A <- matrix (runif(m*n),m,n)
B <- crossprod(A)
quit(save = "no")
```

Elapsed wall time: 31.291 seconds

Benchmark code adapted from Revolution Analytics

Multicore: Baseline case lapply()

```
                    user  system elapsed
                  11.821   1.169  12.990
n <- 500000

set.seed(1)
x <- runif(n)
y <- rnorm(n)
z <- rnorm(n)
mylist <- list(y~x, z~x,
y~z, x~z, z~y, x~y)
lapply(mylist, lm)
quit(save="no")
```

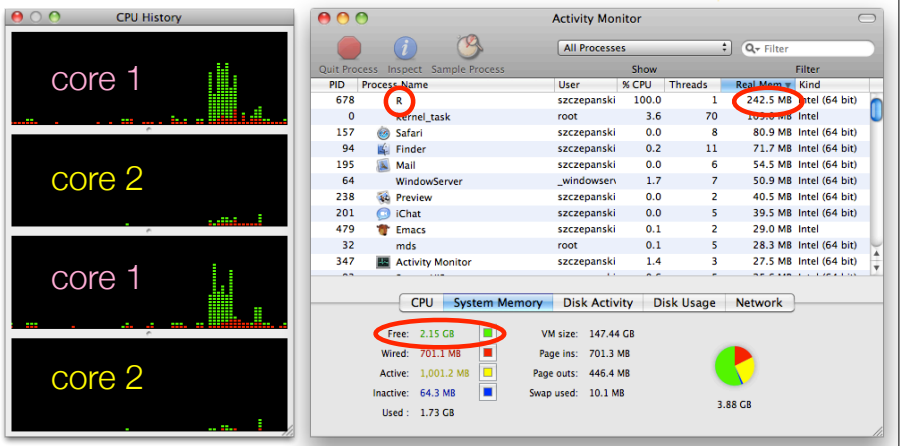

Multicore: mclapply()

```
library(multicore)    user  system elapsed
                    36.377   3.144  18.461
n <- 500000

set.seed(1)
x <- runif(n)
y <- rnorm(n)
z <- rnorm(n)
mylist <- list(y~x, z~x,
y~z, x~z, z~y, x~y)
mclapply(mylist, lm)
quit(save="no")
```

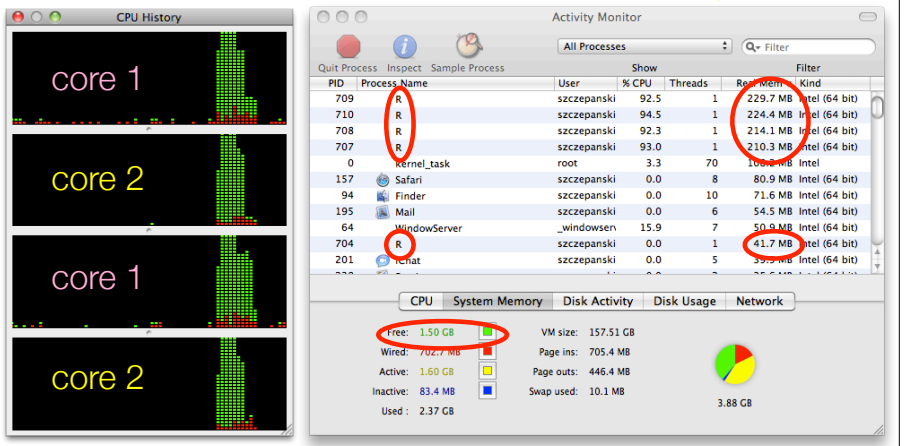

Multicore: parallel() / collect()

```
library(multicore)
                    parallel(lm(y~x))
                    parallel(lm(z~x))
n <- 500000         parallel(lm(y~z))
                    parallel(lm(x~z))
                    parallel(lm(z~y))
set.seed(1)         parallel(lm(x~y))
x <- runif(n)
y <- rnorm(n)
z <- rnorm(n)       myanswer <- collect()
                    quit(save="no")


  user  system elapsed
13.784   1.129  19.862
```

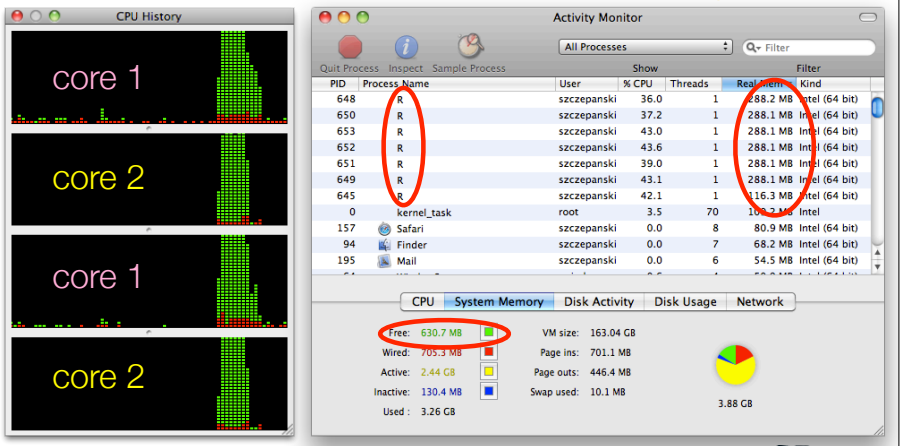
Multicore: Baseline case lapply()



Multicore: mclapply()

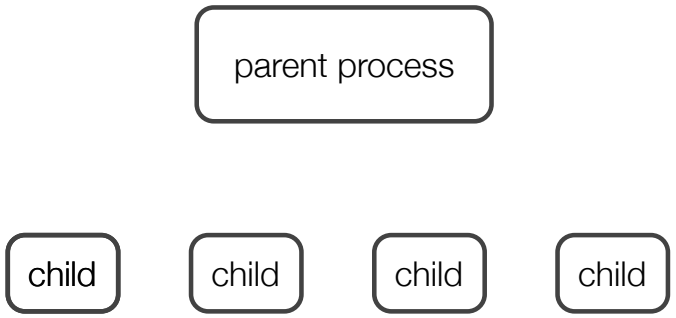


Multicore: parallel()/collect()



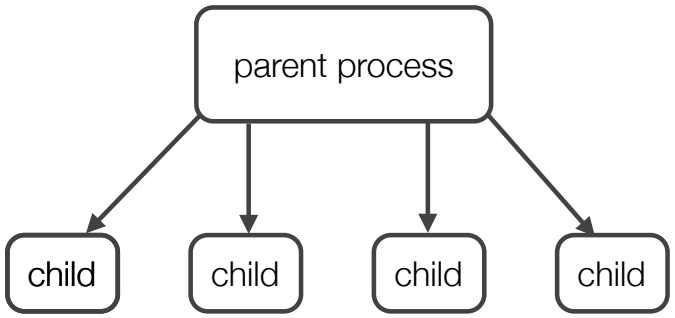
http://math.acadiau.ca/ACMMaC/Rmpi/index.html

Rmpi `library(Rmpi)`



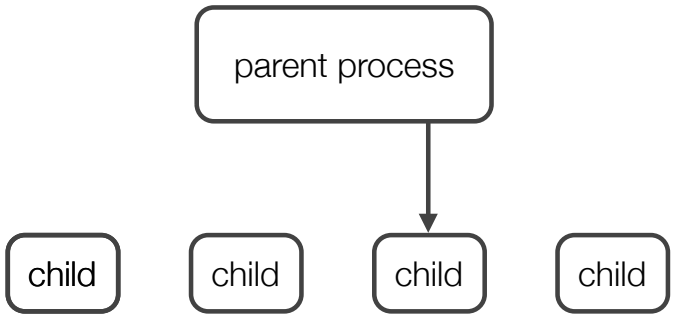
`mpi.spawn.Rslaves([nslaves=#])`

Rmpi



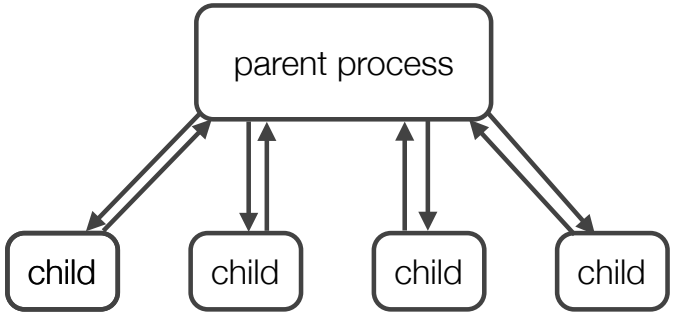

```
mpi.bcast.Robj2slave(object)
mpi.bcast.cmd("R code")
```


Rmpi




```
mpi.send.Robj(object,destination,tag)
object <- mpi.recv.Robj(mpi.any.source(),
mpi.any.tag())
```


Rmpi




```
results <- mpi.remote.exec("R code")
```


Rmpi

- Do basic set up
- Create child processes so that total number of processes (parent + children) is ≤ number of cores
- Parent maintains list of tasks not yet done — careful with memory management!
- As children finish tasks, they send results to parent and ask parent for a new task
- This continues until there are no more tasks left, and the children shut down
- Parent combines results from children

foreach

Partnered with the `iterators` package

non-parallel version:

```
library(foreach)
result <- foreach(index = vector,
          .combine = "function") %do%
          {
            R code
          }
```

foreach

For parallel processing, must register a back end

```
library(foreach)
library(doMC)
registerDoMC()
result <- foreach(index = vector,
          .combine = "function") %dopar%
          {
            R code
          }
```

foreach

For parallel processing, must register a back end

doMC	multicore
doSMP*	like multicore but for Windows
doSNOW	snow
doMPI	Rmpi

*Part of REvolution R

`library` the package then register it.

foreach — Example

Bootstrap on Iris data

back end	elapsed time
sequential	74.969
doMC	32.969
doMPI	208.961

http://cran.r-project.org/web/packages/doMC/vignettes/gettingstartedMC.pdf

Performance depends on specifics of your data and calculation

Summary

package	pros	cons
linear algebra	•Easy if installed •Parallel code is optimized	•Can interfere with manual attempts at parallelization
multicore	•Very easy to use	•Each process needs memory •Not available on all platforms
Rmpi	•Can control everything •Scales from desktop to cluster	•Code can be more complicated and more difficult to write/debug
foreach	•Compatible with several back-ends •Scales from desktop to cluster	•Inherits cons from chosen back-end