High Performance Computing in R

Marie Auger-Méthé
University of Alberta
Dublin R, December 5th, 2012
Why HPC in R?
Why HPC in R?

• Some task cannot be done without HPC
• R can be a slow
  • R is slow compared to many other computing languages (e.g. C++)
  • for-loops are really slow in R
• R can be memory hog
  • Basic R programming involves loading all the data into a session
  • R leaks memory
CRAN Task View: High-Performance and Parallel Computing with R

• Anything related to pushing R a little further:
  • Using compiled code
  • Parallel computing (in both explicit and implicit modes)
  • Working with large objects
  • Profiling
CRAN Task View: High-Performance and Parallel Computing with R

- Anything related to pushing R a little further:
  - Using compiled code
  - Parallel computing (in both explicit and implicit modes)
  - Working with large objects
  - Profiling
Package: Parallel

- Brian Ripley, Luke Tierney and Simon Urbanek
- Builds on packages: multicore & snow
- Includes functions for random numbers
- Made for simple parallelisation with no communication between the workers
Parallel general set up

- 1) Initialize M workers (start workers, load packages on them, set seed, etc)
- 2) Send data to workers
- 3) Create M tasks and send these tasks (code) to the workers
- 4) Wait to receive the result from all workers
- 5) Shut workers
Parallel general set up

• 1) Initialize M workers (start workers, load packages on them, set seed, etc)
• 2) Send data to workers
• 3) Create M tasks and send these tasks (code) to the workers
• 4) Wait to receive the result from all workers
• 5) Shut workers
How many workers can I use

- Usually associated with number of CPUs your computer has
- But can be higher (e.g., if hyperthreading allowed) or less if restricted for multi-user policies

```r
> library(parallel)
> M <- detectCores()
> M
[1] 2
```
Setting up cluster

- Create a Socket cluster
- Some functions (e.g., mclapply) create a cluster automatically but these functions don't work on Windows.

> cl <- makeCluster(M)
Loading any required library

> clusterEvalQ(cl, library(raster))

[[1]]
[1] "raster"  "sp"     "methods"  "stats"

... 

[[2]]
[1] "raster"  "sp"     "methods"  "stats"

...
Set the seed for random numbers

- Use "L'Ecuyer-CMRG" (pseudo)-random numbers implementation for multiple stream
- Not the default random number implementation in the non-parallel set up

> clusterSetRNGStream(cl, 87)
Parallel general set up

- 1) Initialize M workers (start workers, load packages on them, set seed, etc)
- 2) Send data to workers
- 3) Create M tasks and send these tasks (code) to the workers
- 4) Wait to receive the result from all workers
- 5) Shut workers
Export data to cluster

```r
> library(raster)
> logo <- raster(system.file("external/rlogo.grd", package="raster"))
> clusterExport(cl, "logo")
> a <- 5
> b <- 1:2
> clusterExport(cl, varlist=list("a", "b"))
```
Parallel general set up

1) Initialize M workers (start workers, load packages on them, set seed, etc)
2) Send data to workers
3) Create M tasks and send these tasks (code) to the workers
4) Wait to receive the result from all workers
5) Shut workers
Create the function to send to workers

- What you want to do

```r
> res <- vector(mode="list", length=length(b))
> for (i in b){
+   res[[i]] <- -b[i]*logo
+ }
```
Create the function to send to workers

> res
[[1]]
class : RasterLayer
dimensions : 77, 101, 7777 (nrow, ncol, ncell)
...
[[2]]
class : RasterLayer
dimensions : 77, 101, 7777 (nrow, ncol, ncell)
...
Create the function to send to workers

- Create a function that contains the code that was inside your loop

```r
> for (i in b){
+   res[[i]] <- -b[i]*logo
+ }
```

- Only input of the function is the index

```r
> fun <- function(i){ -b[i]*logo }
```
Send the function to the workers

- Using a similar set up to the lapply type function

> res <- parLapply(cl, b, fun)
Parallel general set up

• 1) Initialize M workers (start workers, load packages on them, set seed, etc)
• 2) Send data to workers
• 3) Create M roughly equal sized tasks and send these tasks (code) to the workers
• 4) Wait to receive the result from all workers
• 5) Shut workers
Will return a list

```r
> res
[[1]]
class : RasterLayer
...
[[2]]
class : RasterLayer
...
```
Load balancing

- More processes than workers
- Processes of unequal time
Load balancing

> Sys.sleep(1)
> sleepT <- 1:10
> sleepT
[1] 1 2 3 4 5 6 7 8 9 10
> sum(sleepT)
[1] 55
> system.time(for(i in sleepT){Sys.sleep(i)})

  user  system elapsed
0.00   0.00   55.52
Load balancing

```r
> sum(6:10)
[1] 40
> library(snow)
> tm1 <- snow.time(parLapply(cl, sleepT, Sys.sleep))
> tm1
          elapsed    send   receive  node 1  node 2
1      40.44      0.00    0.03    15.17    40.42
```
Load balancing

```r
> tm2 <- snow.time(clusterApplyLB(cl, sleepT, Sys.sleep))
> tm2

 elapsed  send  receive  node 1  node 2
   30.18   0.00   0.01   25.14   30.18
```
Load balancing

> plot(tm1)
> plot(tm2)
Parallel general set up

1) Initialize M workers (start workers, load packages on them, set seed, etc)
2) Send data to workers
3) Create M tasks and send these tasks (code) to the workers
4) Wait to receive the result from all workers
5) Shut workers
Stop cluster

> stopCluster(cl)

- If processes crash (there is an error) you might not be able to stop your cluster this way
- You might have to stop it manually
  - Windows: through Task Manager
  - Mac: Activity Monitor
Why use parallel?

- Much faster!
  - Linear increase in speed with number of cores

```r
> sleepT <- rep(10,2)
> system.time(for(i in sleepT){Sys.sleep(i)})

user  system elapsed
0.00   0.00   20.12

> system.time(parLapply(cl,sleepT,Sys.sleep))

user  system elapsed
0.00   0.00   10.14
```
Why not use parallel?

- Some problems are not parallelizable
- Use all your CPU
- Debugging is hard
- Random numbers can be complicated to work with
Reproducibility of random seed

```r
> fun <- function(i){ rnorm(100) }
> cl <- makeCluster(2)
> clusterSetRNGStream(cl, 87)
> res1 <- parLapply(cl,1:2, fun)
> res2 <- parLapply(cl,1:2, fun)
> identical(res1, res2)
[1] FALSE
```
Reproducibility of random seed

> clusterSetRNGStream(cl, 87)
> res3 <- parLapply(cl,1:2,fun)
> identical(res1,res3)

[1] TRUE
Reproducibility of random seed

> stopCluster(cl)
> cl <- makeCluster(1)
> clusterSetRNGStream(cl, 87)
> res4 <- parLapply(cl, 1:2, fun)
> identical(res1, res4)
[1] FALSE
> stopCluster(cl)
Reproducibility of random seed

> library(doRNG)
> library(doParallel)
Reproducibility of random seed

> cl <- makeCluster(2); registerDoParallel(cl)
> set.seed(87)
> res5 <- foreach(i=1:2) %dorng% { fun(i) }
> stopCluster(cl)
> cl <- makeCluster(1); registerDoParallel(cl)
> set.seed(87)
> res6 <- foreach(i=1:2) %dorng% { fun(i) }
> identical(res5, res6)

[1] TRUE
Want to know more?
Package: bigmemory

- Work with matrices
- Does not load in R
- Write back on them, so you can theoretically share information between workers

Error: cannot allocate vector of size 3.7 Gb
Big.matrix vs matrix

```r
> library(bigmemory)
> x <- matrix(0.1, nrow=100000, ncol=10)
> xbm <- big.matrix(nrow=100000, ncol=10)
> xbm[] <- 0.1

> object.size(x)
8000112 bytes

> object.size(xbm)
372 bytes
```
### Big.matrix vs matrix

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

> x[1:2, ]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

> xbm[1:2, ]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Big.matrix vs matrix

> xbm

An object of class "big.matrix"
Slot "address":
<pointer: 0x06b03d68>
THANKS!
Big.matrix vs matrix

> head(x)

[1,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
...

> head(xbm)

[1,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
...