Parallel Computing with R

Le Yan

HPC @ LSU
Parallel Computing: Why?

• Getting results faster
  – Running in parallel **may** speed up the time to reach solution

Example: Moving 200 boxes by 1 person vs. 10 people

• Dealing with bigger data sets
  – Running in parallel **may** allow you to use more memory than that available on a single computer

Example: Moving a grand piano by 1 person vs. 10 people
Parallel Computing: How?

• Identify a group of workers
  – For sake of simplicity we will use worker/process/thread interchangeably
• Divide the workload into chunks
• Assign one chunk or a number of chunks to each worker
• Each worker processes its own assignment in parallel with other workers
Parallel Computing: Requirements

• Hardware: modern computers are equipped with more than one CPU core and are capable of processing workloads in parallel
  – Your laptop/desktop/workstation has many cores
  – HPC clusters is composed of many nodes (servers), each of which has many cores

• Software: many software packages are aware of parallel hardware and are capable of coordinating workload processing among workers
Parallel Computing: Requirements

• Hardware: modern computers are equipped with more than one CPU core and are capable of processing workloads in parallel
  – Your laptop/desktop/workstation has many cores
  – HPC clusters is composed of many nodes (servers), each of which has many cores
• Base R is single-threaded, i.e. not parallel
  – Regardless how many cores are available, R can only use one of them
Parallel Computing: Requirements

• Hardware: modern computers are equipped with more than one CPU core and are capable of processing workloads in parallel
  – Your laptop/desktop/workstation has many cores
  – HPC clusters is composed of many nodes (servers), each of which has many cores

• Base
  – R is single-threaded, i.e. not parallel
    – Regardless how many cores are available, R can only use one of them

The goal of this training is to show how to use some R packages to achieve parallel processing
Where We Are with Base R

QB2 Cluster

Cluster = multiple nodes (servers) x multiple cores per node
What We Want to Achieve

QB2 Cluster

Cluster = multiple nodes (servers) x multiple cores per node
Parallel Computing: Caveats

• Using more workers does not always make your program run faster

Example: Moving 200 boxes by 200 people vs. 1,000 people

• **Efficiency** of parallel programs
  – Low efficiency means idle workers and vice versa
  – Defined as speedup divided by number of workers
    • 4 workers, 3x speedup, efficiency = 3/4 = 75%
    • 8 workers, 4x speedup, efficiency = 4/8 = 50%
  – Usually decrease with increasing number of workers
Is Parallel Computing for You?

Does your code run slow?

No

Don’t bother, e.g. it is perhaps not wise to spend weeks to parallelize a program that finishes in 30 seconds;

Yes

Is your code parallelizable?

No

Don’t bother, e.g. not much we can do in R if the target R function is written in C or Fortran;

Yes

Is your code parallel already?

No

Try parallelization

Yes

Some R functions utilize parallel numerical libraries – they are implicitly parallel already
Implicit Parallelization

• Some functions in R can call parallel numerical libraries
  – On LONI and LSU HPC clusters this is the multi-threaded Intel MKL library
  – Mostly linear algebraic and related functions
    • Example: linear regression, matrix decomposition, computing inverse and determinant of a matrix
R running on one node of the QB2 cluster:
20 cores total, 1 busy, 19 idle
# Matrix creation and random number generation
# are NOT implicitly parallel

# Matrix inversion is implicitly parallel
# Each node has 20 cores

A <- matrix(rnorm(10000*10000),10000,10000)

# 20 out 20 cores are busy when running
# this line

Ainv <- solve(A)

R running on one node of the QB2 cluster:
20 cores total, 20 busy, 0 idle
Know Your R Program

• Before starting writing programs, you need to be able to answer these questions
  – How do I know the program runs faster after parallelization?
  – Which part of my code slows the execution down (the most)?
• First step in parallelization: performance analysis
  – Purpose: know which part takes how long, and locate the “hotspot” first
  – Two most frequent used methods in R
    • system.time()
    • rprof() and summaryRprof()
system.time()

## Output from system.time() function
## User: time spent in user-mode
## System: time spent in kernel (I/O etc.)
## Elapsed: wall clock time

## Usage: system.time(<code segment>)

system.time(
{
  A <- matrix(rnorm(10000*10000),10000,10000)
  Ainv <- solve(A)
})

user system elapsed
156.582  0.948  16.325

How much wall clock time it takes - perhaps the most important metric
`system.time()`

**Code**

```r
[lyan1@qb032 R]$ cat inv_st.R

print("Matrix creation:")
system.time({
  A <- matrix(rnorm(10000*10000),10000,10000)
})

print("Matrix inversion:")
system.time({
  Ainv <- solve(A)
})
```

**Output**

```r
[lyan1@qb032 R]$ Rscript inv_st.R

[1] "Matrix creation:"
user  system elapsed
7.437   0.278   7.711

[1] "Matrix inversion:"
user  system elapsed
149.092   0.768   9.417
```

Measure the execution times of different functions

Note the huge discrepancy between “user” and “elapsed” – it is an indication of implicit parallelization
rprof() and summaryRprof()
rprof() and summaryRprof()

[lyan1@qb032 R]$ Rscript inv_prof.R

$by.self

<table>
<thead>
<tr>
<th>Function</th>
<th>Self Time (sec)</th>
<th>Self Pct</th>
<th>Total Time (sec)</th>
<th>Total Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;solve.default&quot;</td>
<td>153.36</td>
<td>95.09</td>
<td>153.58</td>
<td>95.23</td>
</tr>
<tr>
<td>&quot;External&quot;</td>
<td>6.68</td>
<td>4.14</td>
<td>6.68</td>
<td>4.14</td>
</tr>
<tr>
<td>&quot;matrix&quot;</td>
<td>1.02</td>
<td>0.63</td>
<td>7.70</td>
<td>4.77</td>
</tr>
<tr>
<td>&quot;diag&quot;</td>
<td>0.22</td>
<td>0.14</td>
<td>0.22</td>
<td>0.14</td>
</tr>
</tbody>
</table>

How much time is spent in this function itself

$by.total

<table>
<thead>
<tr>
<th>Function</th>
<th>Total Time (sec)</th>
<th>Total Pct</th>
<th>Self Time (sec)</th>
<th>Self Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;solve.default&quot;</td>
<td>153.58</td>
<td>95.23</td>
<td>153.36</td>
<td>95.09</td>
</tr>
<tr>
<td>&quot;solve&quot;</td>
<td>153.58</td>
<td>95.23</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>&quot;matrix&quot;</td>
<td>7.70</td>
<td>4.77</td>
<td>1.02</td>
<td>0.63</td>
</tr>
<tr>
<td>&quot;External&quot;</td>
<td>6.68</td>
<td>4.14</td>
<td>6.68</td>
<td>4.14</td>
</tr>
<tr>
<td>&quot;rnorm&quot;</td>
<td>6.68</td>
<td>4.14</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>&quot;diag&quot;</td>
<td>0.22</td>
<td>0.14</td>
<td>0.22</td>
<td>0.14</td>
</tr>
</tbody>
</table>

How much time is spent in this function and the functions it calls
Writing Parallel R Code – parallel Package

• Introduced in R 2.14.1
• Integrated previous multicore and snow packages
• Coarse-grained parallelization
  – Suit for the chunks of computation are unrelated and do not need to communicate
• Two ways of using parallel packages
  – mc*apply function
  – for loop with %dopar%
    • Need foreach and doParallel packages
Function \texttt{mclapply}

- Parallelized version of the \texttt{lapply} function
  - Similar syntax with \texttt{mc.cores} indicates how many cores/workers to use
    \begin{verbatim}
    mclapply(X, FUN, mc.cores = <number of cores>, ...)
    \end{verbatim}
  - Return a list of the same length as \( X \), each element of which is the result of applying ‘\( \text{FUN} \)’ to the corresponding element of \( X \)
- Can use all cores on one node
  - But not on multiple nodes
# Quadratic Equation: $a x^2 + b x + c = 0$

```r
solve.quad.eq <- function(a, b, c) {
  # Return solutions
  x.delta <- sqrt(b*b - 4*a*c)
  x1 <- (-b + x.delta)/(2*a)
  x2 <- (-b - x.delta)/(2*a)

  return(c(x1, x2))
}
```

Function `solve.quad.eq`  
Input: three coefficients of a quadratic equation  
Output: solutions of the quadratic equation

Create 10 million sets of randomly generated coefficients

```r
len <- 1e7
a <- runif(len, -10, 10); b <- runif(len, -10, 10); c <- runif(len, -10, 10)
```

lapply function: call the `solve.quad.eq` function for each set of coefficients

```r
res1.s <- lapply(1:len, FUN = function(x) { solve.quad.eq(a[x], b[x], c[x])})
```

mclapply function: same arguments with one extra: `mc.cores`

```r
library(parallel)
res1.p <- mclapply(1:len,
  FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) },
  mc.cores = 4)
```
# Quadratic Equation: \( a x^2 + b x + c = 0 \)
solve.quad.eq <- function(a, b, c) {
  # Return solutions
  x.delta <- sqrt(b*b - 4*a*c)
  x1 <- (-b + x.delta)/(2*a)
  x2 <- (-b - x.delta)/(2*a)
  return(c(x1, x2))
}

## Function solve.quad.eq

**Input:** three coefficients of a quadratic equation

Create 10 million sets of randomly generated coefficients

- `lapply` function: call the `solve.quad.eq` function for each set of coefficients
- `mclapply` function: same arguments with one extra: `mc.cores`

```r
> system.time(
+ res1.s <- lapply(1:len, FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) })
+ )

user  system elapsed
358.878 0.375 359.046

> system.time(
+ res1.p <- mclapply(1:len, 
+ FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) }, 
+ mc.cores = 4)
+ )

user  system elapsed
11.098 0.342 81.581
```

#Parallel: mclapply with 4 cores
library(parallel)
res1.p <- mclapply(1:len, 
  FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) }, 
  mc.cores = 4)
# Quadratic Equation: \(a x^2 + b x + c = 0\)

```r
solve.quad.eq <- function(a, b, c)
{
  # Return solutions
  x.delta <- sqrt(b*b - 4*a*c)
  x1 <- (-b + x.delta)/(2*a)
  x2 <- (-b - x.delta)/(2*a)
  return(c(x1, x2))
}
```

### Function `solve.quad.eq`

**Input:** three coefficients of a quadratic equation

### Create 10 million sets of randomly generated coefficients

```r
len <- 1e7
a <- runif(len, -10, 10); b <- runif(len, -10, 10); c <- runif(len, -10, 10)
```

### Serial: `lapply`

```r
res1.s <- lapply(1:len, FUN = function(x) { solve.quad.eq(a[x], b[x], c[x])})
```

### Parallel: `mclapply` with 4 cores

```r
library(parallel)
res1.p <- mclapply(1:len,
  FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) },
  mc.cores = 4)
```

```r
> system.time(
  + res1.s <- lapply(1:len, FUN = function(x) { solve.quad.eq(a[x], b[x], c[x])}))
  user  system elapsed
358.878   0.375  359.046
> system.time(  
  + res1.p <- mclapply(1:len,
      FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) },
      mc.cores = 4)
  )
  user  system elapsed
11.098   0.342  81.581
```

It’s always a good idea to check the efficiency of a parallel program:

- Speedup = 359.046/81.581 = 4.40
- Efficiency = 4.40/4 = 110% (!)

```
#Parallel: mclapply with 4 cores
library(parallel)
res1.p <- mclapply(1:len,
  FUN = function(x) { solve.quad.eq(a[x], b[x], c[x]) },
  mc.cores = 4)
```
%dopar%

- **From** `doParallel` **package**
  - On top of packages `parallel`, `foreach`, `iterator`
- **Purpose:** parallelize a `for` loop
- **Can run on multiple nodes**
%dopar%

• Steps
  – Create a cluster of workers (makeCluster)
  – Register the cluster (registerDoParallel)
  – Process the for loop in parallel (foreach ...
    %dopar%)
  – Stop the cluster (stopCluster)
# Parallel version with `%dopar%`

# Step 1: Create a cluster of 4 workers
```r
c1 <- makeCluster(4)
```

# Step 2: Register the cluster
```r
registerDoParallel(cl)
```

# Step 3: Process the loop
```r
ls <- foreach(icount(iters)) %dopar% {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}
```

# Step 4: Stop the cluster
```r
stopCluster(cl)
```

---

# Workload:
Create 1,000 random samples, each with 1,000,000 observations from a standard normal distribution, then take a summary for each sample.

```r
iters <- 1000

# Sequential version
for (i in 1:iters) {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}
```
%dopar%: On A Single Node

---

# Workload:
# Create 1,000 random samples, each with 1,000,000 observations
# from a standard normal distribution, and take a summary for each sample.

iters <- 1000

# Sequential version
for (i in 1:iters) {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}

system.time(
  for (i in 1:iters) {
    to.ls <- rnorm(1e6)
    to.ls <- summary(to.ls)
  }
)
user  system elapsed
60.249  3.499  63.739

# Parallel version with %dopar%

# Step 1: Create a cluster of 4 workers
cl <- makeCluster(4)

# Step 2: Register the cluster
registerDoParallel(cl)

# Step 3: Process the loop with %dopar%
ls <- foreach(icount(iters)) %dopar% {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}

# Step 4: Stop the cluster
stopCluster(cl)

system.time({
  cl <- makeCluster(4)
  registerDoParallel(cl)
  ls <- foreach(icount(iters)) %dopar% {
    to.ls <- rnorm(1e6)
    to.ls <- summary(to.ls)
  }
  stopCluster(cl)
})
user  system elapsed
0.232  0.032  17.738
%dopar%: One Single Node

# Workload:
# Create 1,000 random samples, each with 1,000,000 observations from a standard normal distribution, then take a summary for each sample.

iters <- 1000

# Sequential version
for (i in 1:iters) {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}

user  system elapsed
60.249   3.499  63.739

# Parallel version with %dopar%
cl <- makeCluster(4)
registerDoParallel(cl)
ls <- foreach(icount(iters)) %dopar% {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}
stopCluster(cl)

user  system elapsed
0.232   0.032  17.738

Speedup = 63.739/17.738 = 3.59
Efficiency = 3.59/4 = 90%
makeCluster()

• We specify how many workers to use
• On the same node:
  
  ```r
  cl <- makeCluster(<number of workers>)
  ```

• On multiple nodes:

  – Example: create 4 workers, 2 on \texttt{qb101} and 2 on \texttt{qb102}
  
  ```r
  cl <- makeCluster(c("qb101","qb101","qb102","qb102"))
  ```
# Read all host names
hosts <- as.vector(unique(read.table(Sys.getenv("PBS_NODEFILE"), stringsAsFactors=F))[,1])

# Count number of hosts
nh <- length(hosts)

# Use 4 workers
nc <- 4

# Make a cluster on multiple nodes
cl <- makeCluster(rep(hosts, each = nc/nh))
registerDoParallel(cl)

ls<-foreach(icount(iters)) %dopar% {
  to.ls<-rnorm(1e6)
  to.ls<-summary(to.ls)
}

stopCluster(cl)

Get the host names of the nodes

Same steps for the rest of the code:
- Make a cluster
- Register the cluster
- Process loop with %dopar%
- Stop the cluster
Running Parallel R Codes

• Now we have a R code that can run in parallel
• So the next question is:
  – How do we know how many workers we should we run it with?
    • The more the better (faster)?
Running Parallel R Codes

• Now we have a R code that can run in parallel

• So the next question is:
  – How do we know how many workers we should we run it with?
    • The more the better (faster)?

The answer is: scaling test (trial and error)
cl <- makePSOCKcluster(rep(hosts, each = clusterSize[i]/nh))
registerDoParallel(cl)

t <- system.time(
  ls <- foreach(icount(iters)) %dopar% {
    to.ls <- rnorm(1e6)
    to.ls <- summary(to.ls)
  }
)

stopCluster(cl)

Nothing is returned, so chunks of the workload are independent of each other
We call this relationship between the number of workers and run time the “scaling behavior”.

```r
ls <- foreach(icount(iters)) %dopar% {
  to.ls<-rnorm(1e6)
  to.ls<-summary(to.ls)
}
```

On one node (20 cores total)

On two nodes (40 cores total)

Ideal behavior
ls <- foreach(icount(iters)) %dopar% {
  to.ls <- rnorm(1e6)
  to.ls <- summary(to.ls)
}

On one node (20 cores total)

On two nodes (40 cores total)

Ideal behavior

Observations:
- Deviation from ideal behavior should be expected;
- Number of workers can differ from number of cores, but it doesn’t make sense to have more workers than the number of cores;
- More is not necessarily faster, and could be slower;
res2.p <- foreach(i=1:core, .combine='rbind') %dopar% 
{
    # local data for results
    res <- matrix(0, nrow=chunk.size, ncol=2)
    for(x in ((i-1)*chunk.size+1):(i*chunk.size)) {
        res[x - (i-1)*chunk.size,] <- solve.quad.eq(a[x], b[x], c[x])
    }
    # return local results
    res
} 

The results from each chunk are aggregated into “res2.p”, so chunks of the workload are independent.
On one node (20 cores total)
On two nodes (40 cores total)

Ideal behavior
On one node (20 cores total)

On two nodes (40 cores total)

Observations:
- If there is data dependency, performance deteriorates faster (compared to cases where there is none);
- Performance deteriorates faster when some workers on one node and some on the other;

Ideal behavior
What we have learned about parallel (R) codes

• With increasing number of workers, efficiency decreases, and eventually adding more workers slows it down

• Best scaling behaviors are typically found with codes with no data dependency (we call it “embarrassingly parallel”)

• With this in mind, when developing our codes, we should reduce data dependency as much as possible
How Many Workers to Use

If there is no constraint, minimize the wall clock time

Sometimes our goal should be to maximize efficiency
Summary:
Steps of Developing Parallel (R) Codes

• Step 1: Analyze performance
  – Find “hot spots” – parallelizable code segments that slow down the execution the most
• Step 2: Parallelize code segments
• Step 3: Run scaling tests
  – How do efficiency and run time change with increasing number of workers?
  – What are the optimal number of workers?
• Step 4: Is the code fast enough?
  – If yes, stop developing (for now) and move on to production runs;
  – If no, go back to step 1 and start another iteration.
Memory Management

• Replica of data objects could be created for every worker
  – Memory usage would increase with the number of workers
• R does not necessarily clean them up even if you close the cluster
  – Need to monitor memory footprint closely
  – The `Rprof` function is capable of memory profiling as well
res2.p <- foreach(i=1:core, .combine='rbind') %dopar% {
  # local data for results
  res <- matrix(0, nrow=chunk.size, ncol=2)
  for(x in ((i-1)*chunk.size+1):(i*chunk.size)) {
    res[x - (i-1)*chunk.size,] <- solve.quad.eq(a[x], b[x], c[x])
  }
  # return local results
  res
}

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S %CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>87483</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>539m</td>
<td>314m</td>
<td>6692</td>
<td>R 100.0</td>
<td>0.5</td>
<td>0:02.05</td>
<td>R</td>
</tr>
<tr>
<td>87492</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>539m</td>
<td>314m</td>
<td>6692</td>
<td>R 100.0</td>
<td>0.5</td>
<td>0:02.05</td>
<td>R</td>
</tr>
<tr>
<td>87465</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>539m</td>
<td>314m</td>
<td>6692</td>
<td>R 99.4</td>
<td>0.5</td>
<td>0:02.04</td>
<td>R</td>
</tr>
<tr>
<td>87474</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>539m</td>
<td>314m</td>
<td>6692</td>
<td>R 99.4</td>
<td>0.5</td>
<td>0:02.04</td>
<td>R</td>
</tr>
</tbody>
</table>

With 4 workers:
Memory = 314*4 = 1256 MB
res2.p <- foreach(i=1:core, .combine='rbind') %dopar% {
    # local data for results
    res <- matrix(0, nrow=chunk.size, ncol=2)
    for(x in ((i-1)*chunk.size+1):(i*chunk.size)) {
        res[x - (i-1)*chunk.size,] <- solve.quad.eq(a[x], b[x], c[x])
    }
    # return local results
    res
}

With 20 workers:
Memory = 276*20 = 5520 MB
```
res2.p <- foreach(i=1:core, .combine='rbind') %dopar% {
  # local data for results
  res <- matrix(0, nrow=chunk.size, ncol=2)
  for(x in ((i-1)*chunk.size+1):(i*chunk.size)) {
    res[x - (i-1)*chunk.size,] <- solve.quad.eq(a[x], b[x], c[x])
  }
  # return local results
  res
}
```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>87514</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.8</td>
<td>0.4</td>
<td>0:03.63</td>
<td>R</td>
</tr>
<tr>
<td>87523</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.8</td>
<td>0.4</td>
<td>0:03.63</td>
<td>R</td>
</tr>
<tr>
<td>87676</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.8</td>
<td>0.4</td>
<td>0:03.61</td>
<td>R</td>
</tr>
<tr>
<td>87505</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.5</td>
<td>0.4</td>
<td>0:03.64</td>
<td>R</td>
</tr>
<tr>
<td>87532</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.5</td>
<td>0.4</td>
<td>0:03.63</td>
<td>R</td>
</tr>
<tr>
<td>87577</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.5</td>
<td>0.4</td>
<td>0:03.63</td>
<td>R</td>
</tr>
<tr>
<td>87613</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.2</td>
<td>0.4</td>
<td>0:03.61</td>
<td>R</td>
</tr>
<tr>
<td>87640</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.2</td>
<td>0.4</td>
<td>0:03.61</td>
<td>R</td>
</tr>
<tr>
<td>87649</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.2</td>
<td>0.4</td>
<td>0:03.61</td>
<td>R</td>
</tr>
<tr>
<td>87667</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>99.2</td>
<td>0.4</td>
<td>0:03.61</td>
<td>R</td>
</tr>
<tr>
<td>87586</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.8</td>
<td>0.4</td>
<td>0:03.59</td>
<td>R</td>
</tr>
<tr>
<td>87631</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.8</td>
<td>0.4</td>
<td>0:03.60</td>
<td>R</td>
</tr>
<tr>
<td>87658</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.8</td>
<td>0.4</td>
<td>0:03.60</td>
<td>R</td>
</tr>
<tr>
<td>87550</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.5</td>
<td>0.4</td>
<td>0:03.60</td>
<td>R</td>
</tr>
<tr>
<td>87622</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.5</td>
<td>0.4</td>
<td>0:03.60</td>
<td>R</td>
</tr>
<tr>
<td>87568</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>98.5</td>
<td>0.4</td>
<td>0:03.60</td>
<td>R</td>
</tr>
<tr>
<td>87604</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>97.5</td>
<td>0.4</td>
<td>0:03.56</td>
<td>R</td>
</tr>
<tr>
<td>87559</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>96.2</td>
<td>0.4</td>
<td>0:03.52</td>
<td>R</td>
</tr>
<tr>
<td>87595</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>91.5</td>
<td>0.4</td>
<td>0:03.36</td>
<td>R</td>
</tr>
<tr>
<td>87541</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>87.9</td>
<td>0.4</td>
<td>0:03.27</td>
<td>R</td>
</tr>
<tr>
<td>87541</td>
<td>lyan1</td>
<td>20</td>
<td>0</td>
<td>501m</td>
<td>276m</td>
<td>6692</td>
<td>R</td>
<td>86.9</td>
<td>0.4</td>
<td>0:03.22</td>
<td>R</td>
</tr>
</tbody>
</table>

The memory footprint doesn’t increase linearly with the number of workers, but quite close, so we need to monitor it closely when changing the number of workers.

With 20 workers:
Memory = 276*20 = 5520 MB
R with GPU

• GPU stands for **Graphic Processing Unit**
  – Originally designed to process graphic data
  – Can tremendously accelerate certain types of computation as well, e.g. matrix multiplications
  – All nodes on LONI QB2 cluster are equipped with two GPU’s

• Package **gpuR** brings the processing power of GPU to R
Example: Matrix Multiplication on GPU

```r
[lyan1@gb032 R]$ cat matmul_gpu.R
# Load necessary library
library(gpuR)

ORDER <- 8192

# On CPU
A <- matrix(rnorm(ORDER^2), nrow=ORDER)
B <- matrix(rnorm(ORDER^2), nrow=ORDER)
cetime <- system.time(C <- A %*% B)
print(paste("On CPU:",ctime["elapsed"],"seconds"))

# On GPU
vclA <- vclMatrix(rnorm(ORDER^2), nrow=ORDER, ncol=ORDER)
vclB <- vclMatrix(rnorm(ORDER^2), nrow=ORDER, ncol=ORDER)
gtime <- system.time(vclC <- vclA %*% vclB)
print(paste("On GPU:",gtime["elapsed"],"seconds"))
print(paste("The speedup is",ctime["elapsed"]/gtime["elapsed"]))
```

On CPU: Create matrix A and B, then multiply them

On GPU: Create matrix A and B (with a different function than on CPU), then multiply them
Example: Matrix Multiplication on GPU

```
Rscript matmul_gpu.R
Loading required package: methods
Number of platforms: 1
- platform: NVIDIA Corporation: OpenCL 1.2 CUDA 8.0.0
  - gpu index: 0
    - Tesla K20Xm
  - gpu index: 1
    - Tesla K20Xm
checked all devices
completed initialization
gpuR 1.2.1
Attaching package: 'gpuR'
The following objects are masked from 'package:base':
  colnames, svd

[1] "On CPU: 4.295 seconds"
[1] "On GPU: 0.0309999999999988 seconds"
[1] "The speedup is 138.54838709678"
```

Wow! Huge speedup! Especially so given the CPU results are obtained with 20 cores (implicitly parallel).
The Other Side of The Coin

```
# Load necessary library
library(gpuR)

ORDER <- 8192

# On CPU
ctime <- system.time({A <- matrix(rnorm(ORDER^2), nrow=ORDER)
                      B <- matrix(rnorm(ORDER^2), nrow=ORDER)
                      C <- A %*% B})
print(paste("On CPU:",ctime["elapsed"],"seconds"))

# On GPU
gtime <- system.time({
                      vclA <- vclMatrix(rnorm(ORDER^2), nrow=ORDER, ncol=ORDER)
                      vclB <- vclMatrix(rnorm(ORDER^2), nrow=ORDER, ncol=ORDER)
                      vclC <- vclA %*% vclB
                      })
print(paste("On GPU:",gtime["elapsed"],"seconds"))
print(paste("The speedup is",ctime["elapsed"]/gtime["elapsed"]))
```

Same code with only one difference:
Now we are measuring the run time including matrix creation.
The Other Side of The Coin

```r
[1yan1@qb072 R]$ Rscript matmul_gpu_overall.R
Loading required package: methods
Number of platforms: 1
- platform: NVIDIA Corporation: OpenCL 1.2 CUDA 8.0.0
  - gpu index: 0
    - Tesla K20Xm
  - gpu index: 1
    - Tesla K20Xm
checked all devices
completed initialization
gpuR 1.2.1
Attaching package: 'gpuR'
The following objects are masked:
colnames, svd

[1] "On GPU: 13.897 seconds"
[1] "The speedup is 1.028514859322"
```

This time, not impressive at all: Matrix creation is not much work, but moving data to/from GPU takes a lot of time. And again, wall clock time is what matters at the end of day.
Deep Learning in R

• Since 2012, Deep Neural Network (DNN) has gained great popularity in applications such as
  – Image and pattern recognition
  – Natural language processing

• There are a few R packages that support DNN
  – MXNet (multiple nodes with GPU support)
  – H2o (multiple nodes)
  – Darch
  – Deepnet
  – Rpud
References

• ParallelR (www.parallelr.com)
  – Code: https://github.com/PatricZhao/ParallelR

• R Documentation for packages mentioned in this tutorial
Thank you!