HPC With R: The Basics

Drew Schmidt

November 12, 2016

Tutorial Goals

We hope to introduce you to:

1. Basic debugging.
2. Evaluating the performance of R code.
3. Some R best practices to help with performance.
4. Basics of parallelism in R.

Exercises

Each section has a complement of exercises to give hands-on reinforcement of ideas introduced in the lecture.

1. Later exercises are more difficult than earlier ones.
2. Some exercises require use of things not explicitly shown in lecture; look through the documentation mentioned in the slides to find the information you need.

Part I

Basics

Resources for Learning R

- *The Art of R Programming* by Norm Matloff: http://nostarch.com/artofr.htm
- *An Introduction to R* by Venables, Smith, and the R Core Team: http://cran.r-project.org/doc/manuals/R-intro.pdf
- *The R Inferno* by Patrick Burns: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Mathesaurus: http://mathesaurus.sourceforge.net/
- R programming for those coming from other languages: http://www.johndcook.com/R_language_for_programmers.html
- *aRrgh: a newcomer's (angry) guide to R*, by Tim Smith and Kevin Ushey: http://tim-smith.us/arrgh/

Other Invaluable Resources

- *R Installation and Administration*: http://cran.r-project.org/doc/manuals/R-admin.html
- *Task Views*: http://cran.at.r-project.org/web/views
- *Writing R Extensions*: http://cran.r-project.org/doc/manuals/R-exts.html
- Mailing list archives: http://tolstoy.newcastle.edu.au/R/
- The [R] stackoverflow tag.
- The #rstats hastag on Twitter.

Debugging R Code

- Very broad topic ...
- We'll hit the highlights.
- For more examples, see:
  cran.r-project.org/doc/manuals/R-exts.html#Debugging

Object Inspection Tools

- `print()`
- `str()`
- `unclass()`

Object Inspection Tools: `print()`

Basic printing:

```
> x <- matrix(1:10, nrow=2)
> print(x)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Object Inspection Tools: `str()`

Examining the **str**ucture of an R object:

```
> x <- matrix(1:10, nrow=2)
> str(x)
 int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
```

Object Inspection Tools: `unclass()`

Exposing all data with `unclass()`:

```
1 df <- data.frame(x=rnorm(10), y=rnorm(10))
2 mdl <- lm(y~x, data=df) ### That's a "tilde" character
3
4 mdl
5 print(mdl)
6
7 str(mdl)
8
9 unclass(mdl)
```

Try it!

The R Debugger

- `debug()`
- `debugonce()`
- `undebug()`

Using The R Debugger

1. Declare function to be debugged: `debug(foo)`
2. Call function: `foo(arg1, arg2, ...)`

   next: `Enter` or `n` followed by `Enter`.
   break: Halt execution and exit debugging: `Q`.
   exit: Continue execution and exit debugging: `c`.
3. Call `undebug()` to stop debugging

Using the Debugger

Example Debugger Interaction

```
> f <- function(x){y <- z+1;z <- y*2;z}
> f(1)
Error in f(1) : object 'z' not found
> debug(f)
> f(1)
debugging in: f(1)
debug at #1: {
    y <- z + 1
    z <- y * 2
    z
}
Browse[2]>
debug at #1: y <- z + 1
Browse[2]>
Error in f(1) : object 'z' not found
>
```

Debugging Compiled Code

- Reasonably easy to use gdb and Valgrind.
- See "Writing R Extensions" manual.

Performance and Accuracy

Sometimes $\pi = 3.14$ is (a) infinitely faster than the "correct" answer and (b) the difference between the "correct" and the "wrong" answer is meaningless. ... The thing is, some specious value of "correctness" is often irrelevant because it doesn't matter. While performance almost always matters. And I absolutely detest the fact that people so often dismiss performance concerns so readily.

— Linus Torvalds, August 8, 2008

Compilers often correct bad behavior...

A Really Dumb Loop

```
1 int main(){
2     int x, i;
3     for (i=0; i<10; i++)
4         x = 1;
5     return 0;
6 }
```

clang -O3 -S example.c

```
main:
        .cfi_startproc
# BB#0:
        xorl    %eax, %eax
        ret
```

clang -S example.c

```
main:
        .cfi_startproc
# BB#0:
        movl    $0, -4(%rsp)
        movl    $0, -12(%rsp)
.LBB0_1:
        cmpl    $10, -12(%rsp)
        jge     .LBB0_4
# BB#2:
        movl    $1, -8(%rsp)
# BB#3:
        movl    -12(%rsp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rsp)
        jmp     .LBB0_1
.LBB0_4:
        movl    $0, %eax
        ret
```

R will not!

Dumb Loop

```
1 for (i in 1:n){
2   tA <- t(A)
3   Y <- tA %*% Q
4   Q <- qr.Q(qr(Y))
5   Y <- A %*% Q
6   Q <- qr.Q(qr(Y))
7 }
8
9 Q
```

Better Loop

```
1  tA <- t(A)
2
3 for (i in 1:n){
4   Y <- tA %*% Q
5   Q <- qr.Q(qr(Y))
6   Y <- A %*% Q
7   Q <- qr.Q(qr(Y))
8 }
9
10 Q
```

Example from a Real R Package

Exerpt from Original function

```
1 while(i<=N){
2   for(j in 1:i){
3     d.k <- as.matrix(x)[l==j,l==j]
4       ...
```

Exerpt from Modified function

```
1 x.mat <- as.matrix(x)
2
3 while(i<=N){
4   for(j in 1:i){
5     d.k <- x.mat[l==j,l==j]
6       ...
```

By changing just 1 line of code, performance of the main method improved by **over 350%**!

Some Thoughts

- R is slow.
- Bad programmers are slower.
- R can't fix bad programming.

Timings

Getting simple timings as a basic measure of performance is easy, and valuable.

- `system.time()` — timing blocks of code.
- `Rprof()` — timing execution of R functions.
- `Rprofmem()` — reporting memory allocation in R .
- `tracemem()` — detect when a copy of an R object is created.

Performance Profiling Tools: system.time()

system.time() is a basic R utility for timing expressions

```
x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)

system.time(t(x) %*% x)
#    user  system elapsed
#   2.187   0.032   2.324

system.time(crossprod(x))
#    user  system elapsed
#   1.009   0.003   1.019

system.time(cov(x))
#    user  system elapsed
#   6.264   0.026   6.338
```

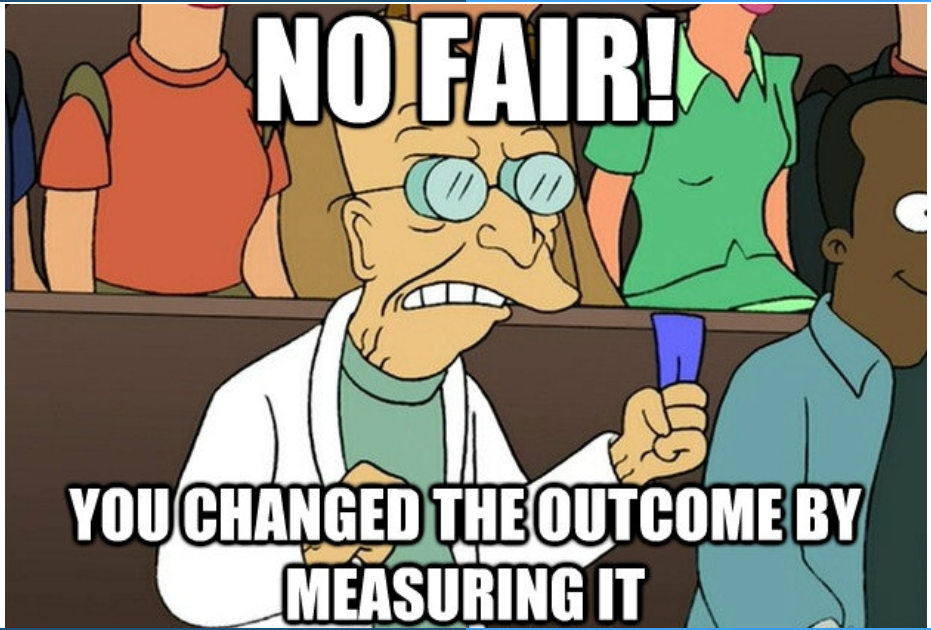
Performance Profiling Tools: system.time()

Put more complicated expressions inside of brackets:

```
x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)

system.time({
  y <- x+1
  z <- y*2
})
#    user  system elapsed
#   0.057   0.032   0.089
```

Performance Profiling Tools: `Rprof()`

```
1 Rprof(filename="Rprof.out", append=FALSE, interval=0.02,
2   memory.profiling=FALSE, gc.profiling=FALSE,
3   line.profiling=FALSE, numfiles=100L, bufsize=10000L)
```


NO FAIR!
YOU CHANGED THE OUTCOME BY MEASURING IT

Performance Profiling Tools: Rprof()

```
1  x <- matrix(rnorm(10000*250), nrow=10000, ncol=250)
2
3  Rprof()
4  invisible(prcomp(x))
5  Rprof(NULL)
6
7  summaryRprof()
8
9  Rprof(interval=.99)
10 invisible(prcomp(x))
11 Rprof(NULL)
12
13 summaryRprof()
```

Performance Profiling Tools: `Rprof()`

```
$by.self
                self.time self.pct total.time total.pct
"La.svd"             0.68    69.39       0.72     73.47
"%*%"                0.12    12.24       0.12     12.24
"aperm.default"      0.04     4.08       0.04      4.08
"array"              0.04     4.08       0.04      4.08
"matrix"             0.04     4.08       0.04      4.08
"sweep"              0.02     2.04       0.10     10.20
### output truncated by presenter

$by.total
                 total.time total.pct self.time self.pct
"prcomp"               0.98    100.00       0.00     0.00
"prcomp.default"       0.98    100.00       0.00     0.00
"svd"                  0.76     77.55       0.00     0.00
"La.svd"               0.72     73.47       0.68    69.39
### output truncated by presenter

$sample.interval
[1] 0.02

$sampling.time
[1] 0.98
```

Performance Profiling Tools: `Rprof()`

```
$by.self
[1] self.time  self.pct   total.time total.pct
<0 rows> (or 0-length row.names)

$by.total
[1] total.time total.pct  self.time  self.pct
<0 rows> (or 0-length row.names)

$sample.interval
[1] 0.99

$sampling.time
[1] 0
```

3 Profiling
- Why Profile?
- Profiling R Code
- Advanced R Profiling

Other Profiling Tools

- perf, PAPI
- fpmpi, mpiP, TAU
- pbdPROF
- pbdPAPI

See forthcoming paper *Analyzing Analytics: Advanced Performance Analysis Tools for R* for more details.

Benchmarking

- There's *a lot* that goes on when executing an R funciton.
- Symbol lookup, creating the abstract syntax tree, creating promises for arguments, argument checking, creating environments, ...
- Executing a second time can have dramatically different performance over the first execution.
- Benchmarking several methods fairly requires some care.

Benchmarking tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

```
x <- matrix(rnorm(10000*500), nrow=10000, ncol=500)

f <- function(x) t(x) %*% x
g <- function(x) crossprod(x)

library(rbenchmark)
benchmark(f(x), g(x), columns=c("test", "replications", "elapsed", "relative"))

#   test replications elapsed relative
# 1 f(x)          100  13.679    3.588
# 2 g(x)          100   3.812    1.000
```

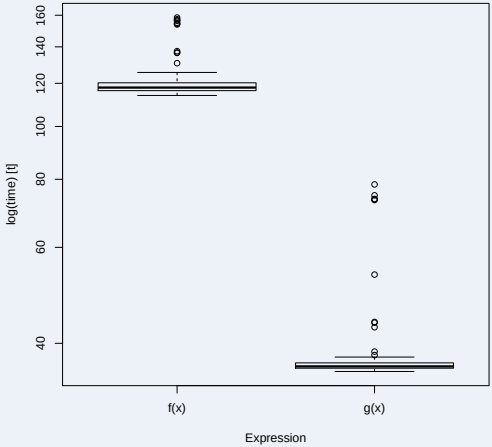
Benchmarking tools: microbenchmark

microbenchmark is a separate package with a slightly different philosophy:

```
x <- matrix(rnorm(10000*500), nrow=10000, ncol=500)

f <- function(x) t(x) %*% x
g <- function(x) crossprod(x)

library(microbenchmark)
microbenchmark(f(x), g(x), unit="s")

# Unit: seconds
#   expr        min         lq       mean     median         uq        max neval
#   f(x) 0.11418617 0.11647517 0.12258556 0.11754302 0.12058145 0.17292507   100
#   g(x) 0.03542552 0.03613772 0.03884497 0.03668231 0.03740173 0.07478309   100
```

Benchmarking tools: microbenchmark

```
1 bench <- microbenchmark(f(x), g(x), unit="s")
2 boxplot(bench)
```



Part II

Improving R Performance

5 Free Improvements
- Packages
- The Bytecode Compiler
- Choice of BLAS Library

6 Writing Better R Code

Packages

- Many high-quality "application" packages exist.
- Data manipulation: dplyr, data.table
- Modeling/math: Many! Try the CRAN taskviews.
- Parallelism: Discussed separately.

The Compiler Package

- Released in 2011 (Tierney)
- Bytecode: sort of like machine code for interpreters...
- Improves R code speed by 2-5% generally.
- Does best on loops.

Bytecode Compilation

- Non-core packages not (bytecode) compiled by default.
- "Base" and "recommended" (core) packages are.
- Downsides:
  - (slightly) larger install size
  - (much!) longer install process
  - doesn't fix bad code
- Upsides: slightly faster.

Compiling a Function

```
test <- function(x) x+1
test
# function(x) x+1

library(compiler)

test <- cmpfun(test)
test
# function(x) x+1
# <bytecode: 0x38c86c8>

disassemble(test)
# list(.Code, list(7L, GETFUN.OP, 1L, MAKEPROM.OP, 2L, PUSHCONSTARG.OP,
#     3L, CALL.OP, 0L, RETURN.OP), list(x + 1, `+`, list(.Code,
#     list(7L, GETVAR.OP, 0L, RETURN.OP), list(x)), 1))
```

Compiling Packages

From R

```
1 install.packages("my_package", type="source", INSTALL_opts="--byte-compile")
```

From The Shell

```
1 export R_COMPILE_PKGS=1
2 R CMD INSTALL my_package.tar.gz
```

Or add the line: `ByteCompile: yes` to the package's `DESCRIPTION` file.

The Compiler: How much does it help *really*?

```
1  f <- function(n) for (i in 1:n) 2*(3+4)
2
3
4  library(compiler)
5  f_comp <- cmpfun(f)
6
7
8  library(rbenchmark)
9
10 n <- 100000
11 benchmark(f(n), f_comp(n), columns=c("test", "replications", "elapsed",
       "relative"),
12 order="relative")
13 #         test replications elapsed relative
14 # 2 f_comp(n)          100   2.604    1.000
15 # 1       f(n)          100   2.845    1.093
```

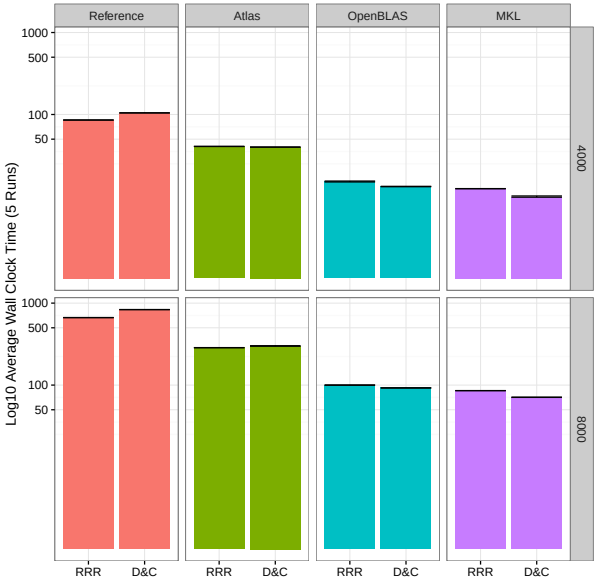
The Compiler: How much does it help *really*?

```
1  g <- function(n){
2    x <- matrix(runif(n*n), nrow=n, ncol=n)
3    min(colSums(x))
4  }
5
6  library(compiler)
7  g_comp <- cmpfun(g)
8
9
10 library(rbenchmark)
11
12 n <- 1000
13 benchmark(g(n), g_comp(n), columns=c("test", "replications", "elapsed",
       "relative"),
14 order="relative")
15 #         test replications elapsed relative
16 # 2 g_comp(n)          100   6.854    1.000
17 # 1       g(n)          100   6.860    1.001
```

5 Free Improvements
- Packages
- The Bytecode Compiler
- Choice of BLAS Library

The BLAS

- Basic Linear Algebra Subprograms.
- Basic numeric matrix operations.
- Used in linear algebra and many statistical operations.
- Different implementations available.
- Several multithreaded BLAS libraries exist.



Comparing Symmetric Eigenvalue Performance http://bit.ly/2f49Sop

Loops

- `for`
- `while`
- No `goto`'s or `do while`'s.
- They're *really* slow.

Loops: Best Practices

- *Profile, profile, profile.*
- Mostly try to avoid.
- Evaluate practicality of rewrite (plys, vectorization, compiled code)
- Always preallocate storage; don't grow it dynamically.

"Ply" Functions

- R has functions that apply other functions to data.
- In a nutshell: loop sugar.
- Typical *ply's:
  - `apply()`: apply function over matrix "margin(s)".
  - `lapply()`: apply function over list/vector.
  - `mapply()`: apply function over multiple lists/vectors.
  - `sapply()`: same as `lapply()`, but (possibly) nicer output.
  - Plus some other mostly irrelevant ones.
- Also `Map()` and `Reduce()`.

Ply Examples: apply()

```
x <- matrix(1:10, 2)

x
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10

apply(X=x, MARGIN=1, FUN=sum)
# [1] 25 30

apply(X=x, MARGIN=2, FUN=sum)
# [1]  3  7 11 15 19

apply(X=x, MARGIN=1:2, FUN=sum)
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10
```

Ply Examples: lapply() and sapply()

```
lapply(1:4, sqrt)
# [[1]]
# [1] 1
#
# [[2]]
# [1] 1.414214
#
# [[3]]
# [1] 1.732051
#
# [[4]]
# [1] 2

sapply(1:4, sqrt)
# [1] 1.000000 1.414214 1.732051 2.000000
```

Transforming Loops Into Ply's

```
vec <- numeric(n)
for (i in 1:n){
  vec[i] <- my_function(i)
}
```

Becomes:

```
sapply(1:n, my_function)
```

Ply's: Best Practices

- Most ply's are just shorthand/higher expressions of loops.
- Generally not much faster (if at all), especially with the compiler.
- Thinking in terms of `lapply()` can be useful however...

Ply's: Best Practices

- With ply's and lambdas, can do some fiendishly crafty things.
- But don't go crazy...

```
1 cat(sapply(letters, function(a) sapply(letters, function(b) sapply(letters,
      function(c) sapply(letters, function(d) paste(a, b, c, d, letters, "\n",
      sep=""))))))
```

Vectorization

- `x+y`
- `x[, 1] <- 0`
- `rnorm(1000)`

Vectorization

- Same in R as in other high-level languages (Matlab, Python, ...).
- Idea: use pre-existing compiled kernels to avoid interpreter overhead.
- Much faster than loops and plys.

```
1 ply <- function(x) lapply(rep(1, 1000), rnorm)
2 vec <- function(x) rnorm(1000)
3
4 library(rbenchmark)
5 benchmark(ply(x), vec(x))
6 #     test replications elapsed relative
7 # 1 ply(x)          100   0.348   38.667
8 # 2 vec(x)          100   0.009    1.000
```

Putting It All Together

- Loops are slow.
- `apply()`, `Reduce()` are just `for` loops.
- `Map()`, `lapply()`, `sapply()`, `mapply()` (and most other core ones) are *not* `for` loops.
- *Ply functions are not vectorized.*
- Vectorization is fastest, but often needs lots of memory.

Squares

Let's compute the square of the numbers 1–100000, using

- `for` loop without preallocation
- `for` loop with preallocation
- `sapply()`
- vectorization

Squares

```
1 square_sapply <- function(n) sapply(1:n, function(i) i^2)
2
3 square_vec <- function(n) (1:n)*(1:n)
```

```
1 library(rbenchmark)
2 n <- 100000
3
4 benchmark(square_loop_noinit(n), square_loop_withinit(n), square_sapply(n),
      square_vec(n))
5 #                       test replications elapsed relative
6 # 1   square_loop_noinit(n)          100  17.296 2470.857
7 # 2 square_loop_withinit(n)          100   0.933  133.286
8 # 3        square_sapply(n)          100   1.218  174.000
9 # 4           square_vec(n)          100   0.007    1.000
```

Part III

Parallelism

Parallel Programming Packages for R

Shared Memory

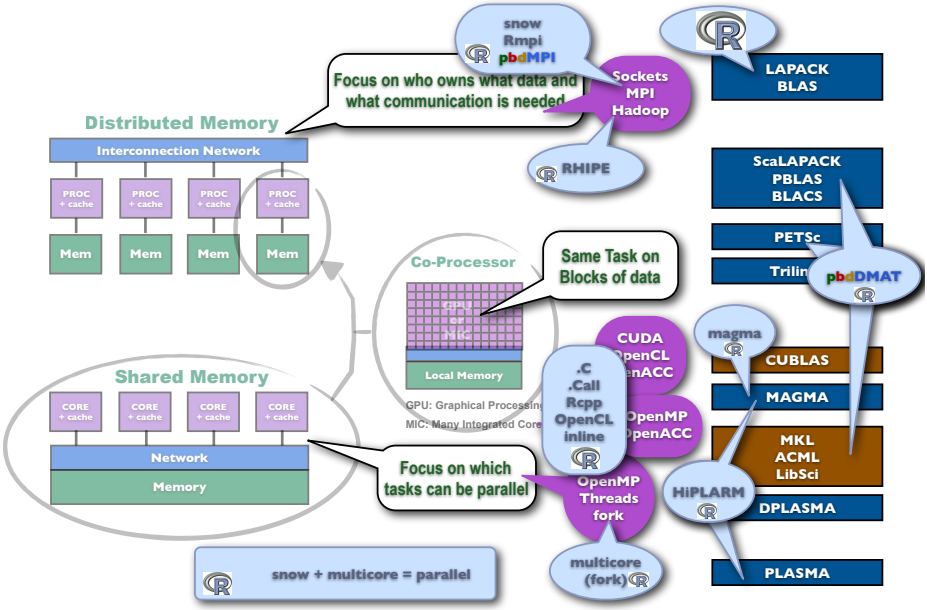
Examples: **parallel**, **snow**, **foreach**, **gputools**, **HiPLARM**

Distributed

Examples: **pbdR**, **Rmpi**, **RHadoop**, **RHIPE**

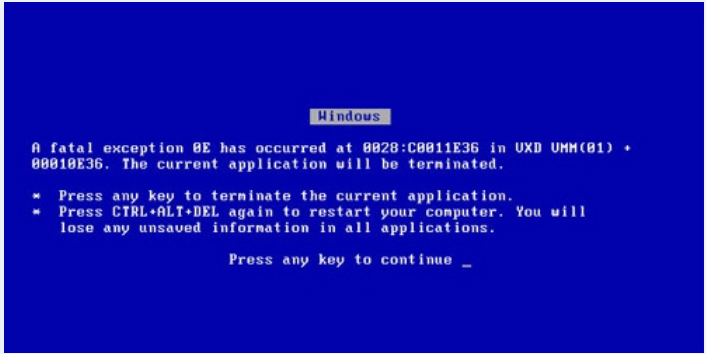
CRAN HPC Task View

For more examples, see:
http://cran.r-project.org/web/views/HighPerformanceComputing.html


snow
Rmpi
pbdMPI
Sockets
MPI
Hadoop
RHIPE
Focus on who owns what data and what communication is needed
Distributed Memory
Interconnection Network
PROC + cache
Mem
LAPACK
BLAS
ScaLAPACK
PBLAS
BLACS
PETSc
Trilin
pbdDMAT
Co-Processor
Same Task on Blocks of data
Local Memory
GPU: Graphical Processin
MIC: Many Integrated Core
CUDA
OpenCL
nACC
.C
.Call
Rcpp
OpenCL
inline
OpenMP
penACC
magma
CUBLAS
MAGMA
MKL
ACML
LibSci
HiPLARM
DPLASMA
PLASMA
Shared Memory
CORE + cache
Network
Memory
Focus on which tasks can be parallel
OpenMP
Threads
fork
multicore
(fork)
snow + multicore = parallel

Portability

Many parallel R packages break on Windows



RNG's in Parallel

- Be careful!
- Aided by **rlecuyer**, **rsprng**, and **doRNG** packages.

Parallel Programming: In Theory

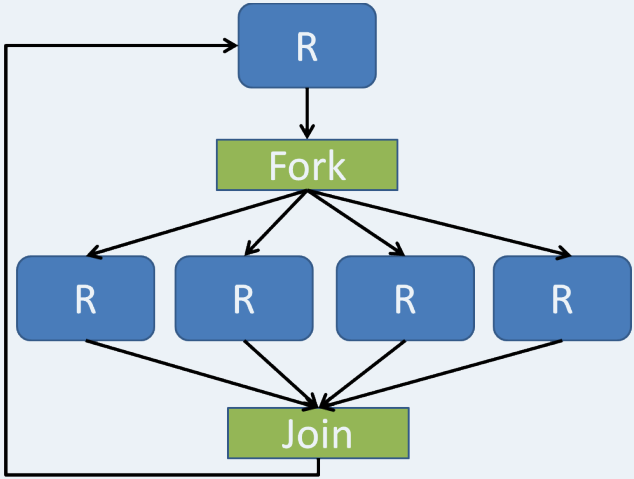
Parallel Programming: In Practice

The parallel Package

- Comes with R $\geq$ 2.14.0
- Has 2 disjoint interfaces.

$$\textbf{parallel} = \textbf{snow} + \textbf{multicore}$$

The parallel Package: multicore



Operates on fork/join paradigm.

The parallel Package: multicore

+ Data copied to child on write (handled by OS)
+ Very efficient.
- No Windows support.
- Not as efficient as threads.

The parallel Package: multicore

```
mclapply(X, FUN, ...,
  mc.preschedule=TRUE, mc.set.seed=TRUE,
  mc.silent=FALSE, mc.cores=getOption("mc.cores", 2L),
  mc.cleanup=TRUE, mc.allow.recursive=TRUE)
```

```
x <- lapply(1:10, sqrt)

library(parallel)
x.mc <- mclapply(1:10, sqrt)

all.equal(x.mc, x)
# [1] TRUE
```

The parallel Package: multicore

```
1 simplify2array(mclapply(1:10, function(i) Sys.getpid(), mc.cores=4))
2 # [1] 27452 27453 27454 27455 27452 27453 27454 27455 27452 27453
3
4 simplify2array(mclapply(1:2, function(i) Sys.getpid(), mc.cores=4))
5 # [1] 27457 2745
```

The parallel Package: snow

? Uses sockets.

\+ Works on all platforms.

\- More fiddley than `mclapply()`.

\- Not as efficient as forks.

The parallel Package: snow

```
1 ### Set up the worker processes
2 cl <- makeCluster(detectCores())
3 cl
4 # socket cluster with 4 nodes on host   localhost
5
6 parSapply(cl, 1:5, sqrt)
7
8 stopCluster(cl)
```

The parallel Package: Summary

All

- `detectCores()`
- `splitIndices()`

multicore

- `mclapply()`
- `mcmapply()`
- `mcparallel()`
- `mccollect()`
- and others...

snow

- `makeCluster()`
- `stopCluster()`
- `parLapply()`
- `parSapply()`
- and others...

8 Shared Memory Parallelism in R
- The parallel Package
- The foreach Package

The foreach Package

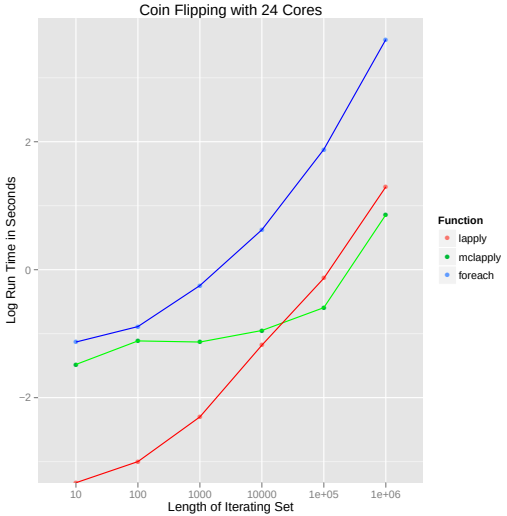
- On Cran (Revolution Analytics).
- Main package is **foreach**, which is a single interface for a number of "backend" packages.
- Backends: **doMC**, **doMPI**, **doParallel**, **doRedis**, **doRNG**, **doSNOW**.

The foreach Package: The Idea

Unify the disparate interfaces.

The foreach Package

+ Works on all platforms (if backend does).
+ Can even work serial with minor notational change.
+ Write the code once, use whichever backend you prefer.
- Really bizarre, non-R-ish synatx.
- Efficiency issues if you aren't careful!




```
1  ### Bad performance
2  foreach(i=1:len) %dopar%
       tinyfun(i)
3
4  ### Expected performance
5  foreach(i=1:ncores) %dopar% {
6    out <- numeric(len/ncores)
7    for (j in 1:(len/ncores))
8      out[i] <- tinyfun(j)
9    out
10 }
```

The foreach Package: General Procedure

- Load **foreach** and your backend package.
- Register your backend.
- Call `foreach`

Using foreach: serial

```
1  library(foreach)
2
3  ### Example 1
4  foreach(i=1:3) %do% sqrt(i)
5
6  ### Example 2
7  n <- 50
8  reps <- 100
9
10 x <- foreach(i=1:reps) %do% {
11   sum(rnorm(n, mean=i)) / (n*reps)
12 }
```

Using foreach: Parallel

```
1  library(foreach)
2  library(<mybackend>)
3
4  register<MyBackend>()
5
6  ### Example 1
7  foreach(i=1:3) %dopar% sqrt(i)
8
9  ### Example 2
10 n <- 50
11 reps <- 100
12
13 x <- foreach(i=1:reps) %dopar% {
14   sum(rnorm(n, mean=i)) / (n*reps)
15 }
```

foreach backends

multicore

```
1 library(doParallel)
2 registerDoParallel(cores=ncores)
3 foreach(i=1:2) %dopar% Sys.getpid()
```

snow

```
1 library(doParallel)
2 cl <- makeCluster(ncores)
3 registerDoParallel(cl=cl)
4
5 foreach(i=1:2) %dopar% Sys.getpid()
6 stopCluster(cl)
```

Why Distribute?

- Nodes only hold so much ram.
- Commodity hardware: $\approx 32 - 64$ gib.
- With a few exceptions (**ff**, **bigmemory**), R does computations in memory.
- If your problem doesn't fit in the memory of one node...

Packages for Distributed Memory Parallelism in R

- **Rmpi**, and **snow** via **Rmpi**.
- **RHIPE** and **RHadoop** ecosystem.
- **pbdR** ecosystem.

9 Distributed Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi

Rmpi Hello World

```
1  mpi.spawn.Rslaves(nslaves=2)
2  #         2 slaves are spawned successfully. 0 failed.
3  # master (rank 0, comm 1) of size 3 is running on: wootabega
4  # slave1 (rank 1, comm 1) of size 3 is running on: wootabega
5  # slave2 (rank 2, comm 1) of size 3 is running on: wootabega
6
7  mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
8  # $slave1
9  # [1] "I am 1 of 3"
10 #
11 # $slave2
12 # [1] "I am 2 of 3"
13
14 mpi.exit()
```

Using Rmpi from snow

```
1 library(snow)
2 library(Rmpi)
3
4 cl <- makeCluster(2, type = "MPI")
5 clusterCall(cl, function() Sys.getpid())
6 clusterCall(cl, runif, 2)
7 stopCluster(cl)
8 mpi.quit()
```

Rmpi Resources

- **Rmpi** tutorial: http://math.acadiau.ca/ACMMaC/Rmpi/
- **Rmpi** manual: http://cran.r-project.org/web/packages/Rmpi/Rmpi.pdf

pbdMPI vs Rmpi

- **Rmpi** is interactive; **pbdMPI** is exclusively batch.
- **pbdMPI** is easier to install.
- **pbdMPI** has a simpler interface.
- **pbdMPI** integrates with other pbdR packages.

Example Syntax

Rmpi

```
# int
mpi.allreduce(x, type=1)
# double
mpi.allreduce(x, type=2)
```

pbdMPI

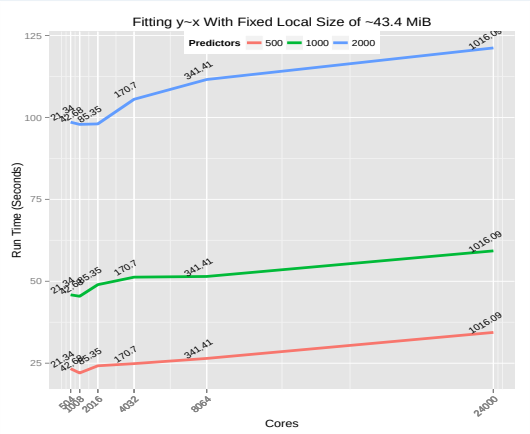
```
allreduce(x)
```

Types in R

```
> typeof(1)
[1] "double"
> typeof(2)
[1] "double"
> typeof(1:2)
[1] "integer"
```

Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark




```
x <- ddmatrix("rnorm", nrow=m, ncol=n)
y <- ddmatrix("rnorm", nrow=m, ncol=1)
mdl <- lm.fit(x=x, y=y)
```

pbdR Scripts

- They're just R scripts.
- Can't run interactively (with more than 1 rank).
- We can use **pbdinline** to get "pretend interactivity".

ddmatrix: 2-dimensional Block-Cyclic with 6 Processors

$$x = \left[\begin{array}{ccc|cc|ccc} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \\ \hline x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \\ \hline x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ \hline x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{9\times 9}$$

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

Understanding `ddmatrix`: Local View

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{27} & x_{28} \\ \hline x_{51} & x_{52} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{67} & x_{68} \\ \hline x_{91} & x_{92} & x_{97} & x_{98} \end{array}\right]_{5\times4} \left[\begin{array}{cc|c} x_{13} & x_{14} & x_{19} \\ x_{23} & x_{24} & x_{29} \\ \hline x_{53} & x_{54} & x_{59} \\ x_{63} & x_{64} & x_{69} \\ \hline x_{93} & x_{94} & x_{99} \end{array}\right]_{5\times3} \left[\begin{array}{cc} x_{15} & x_{16} \\ x_{25} & x_{26} \\ \hline x_{55} & x_{56} \\ x_{65} & x_{66} \\ \hline x_{95} & x_{96} \end{array}\right]_{5\times2}$$

$$\left[\begin{array}{cc|cc} x_{31} & x_{32} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{47} & x_{48} \\ \hline x_{71} & x_{72} & x_{77} & x_{78} \\ x_{81} & x_{82} & x_{87} & x_{88} \end{array}\right]_{4\times4} \left[\begin{array}{cc|c} x_{33} & x_{34} & x_{39} \\ x_{43} & x_{44} & x_{49} \\ \hline x_{73} & x_{74} & x_{79} \\ x_{83} & x_{84} & x_{89} \end{array}\right]_{4\times3} \left[\begin{array}{cc} x_{35} & x_{36} \\ x_{45} & x_{46} \\ \hline x_{75} & x_{76} \\ x_{85} & x_{86} \end{array}\right]_{4\times2}$$

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

Methods for class ddmatrix

pbdDMAT has over 100 methods with *identical* syntax to R:

- `[`, `rbind()`, `cbind()`, ...
- `lm.fit()`, `prcomp()`, `cov()`, ...
- `%*%`, `solve()`, `svd()`, `norm()`, ...
- `median()`, `mean()`, `rowSums()`, ...

Serial Code

```
1 cov(x)
```

Parallel Code

```
1 cov(x)
```

Part IV

Wrapup

~Thanks!~

Questions?


Email: wrathematics@gmail.com


GitHub: https://github.com/wrathematics


Web: http://wrathematics.info


Twitter: @wrathematics