High Performance Computing with R

Drew Schmidt

February 27, 2015
## Tutorial Structure

1. **(45 Minutes) Basics:** Intro, debugging, profiling, benchmarking.
2. **(15 Minutes) Exercises**
3. **(45 Minutes) Improving R Code:** compilers, vectorization, loops, ...
4. **(30 Minutes) Exercises + Break**
5. **(45 Minutes) Interfacing to Compiled Code**
6. **(15 Minutes) Exercises**
7. **(45 Minutes) Parallelism**
Tutorial Goals

We hope to introduce you to:

1. Basic debugging.
2. Evaluating the performance of R code.
4. Why and how to interface R to C++.
5. 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. More exercises are given than you have time to complete.
2. Later exercises are more difficult than earlier ones.
3. 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.
National Institute for Computational Sciences

University of Tennessee & ORNL partnership

- NICS is an NSF HPC center established in 2007
  - Takes advantage of the strengths of UT and ORNL
- Series of computers that culminated in a 1.17 Petaflop system in Jan 2011
  - First Academic Petaflop: Kraken

Managed by UT-Battelle for the Department of Energy
Kraken Actual Usage by Discipline (Aug’12) 79.2M hours
NICS Now...

· Growing our Data Sciences

· Collaborating with industry to advance several fields

· Supply NSF cycles through Darter, Beacon, and Nautilus
### Nautilus SGI UltraViolet specs

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute processor type</td>
<td>Intel ~2.0 GHz Nehalem</td>
</tr>
<tr>
<td>Compute cores</td>
<td>1024</td>
</tr>
<tr>
<td>Compute sockets (nodes)</td>
<td>128 oct-core</td>
</tr>
<tr>
<td>Memory per core</td>
<td>4 GB</td>
</tr>
<tr>
<td>Total memory</td>
<td>4 TB (SMP)</td>
</tr>
<tr>
<td>Accelerators</td>
<td>8 NVIDIA Fermi GPUs</td>
</tr>
<tr>
<td>Peak system performance</td>
<td>10 TF</td>
</tr>
<tr>
<td>Interconnect topology</td>
<td>NUMAlink5</td>
</tr>
<tr>
<td>Parallel file system space</td>
<td>1 PB (Lustre)</td>
</tr>
<tr>
<td>Parallel file system peak performance</td>
<td>30 GB/s</td>
</tr>
</tbody>
</table>
Beacon Cray Xtreme-X Supercomputer
Peak Performance: 210.1 TFLOP/s

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>Intel Xeon E5-2670</td>
</tr>
<tr>
<td>CPUs per node</td>
<td>2 8-core, 2.6GHz</td>
</tr>
<tr>
<td>RAM per node</td>
<td>256 GB</td>
</tr>
<tr>
<td>SSD per node</td>
<td>2 x 480 GB (RAID 0)</td>
</tr>
<tr>
<td>Intel® Xeon Phi Coprocessors per node</td>
<td>4 x 5110P 60-core, 1.053GHz 8 GB GDDR5 RAM</td>
</tr>
<tr>
<td>Interconnect</td>
<td>FDR InfiniBand Fat Tree</td>
</tr>
</tbody>
</table>

Darter Cray XC30 Supercomputer
Peak Performance: 248.9 TFLOP/s

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>748</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>Intel Xeon E5-2670</td>
</tr>
<tr>
<td>CPUs per node</td>
<td>2 8-core, 2.6GHz</td>
</tr>
<tr>
<td>RAM per node</td>
<td>16 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Cray Aries Dragonfly</td>
</tr>
</tbody>
</table>

Hosted Accelerators: Intel MICs

#1 on Green500
• Extreme Science and Engineering Discovery Environment

• Follow on NSF project to TeraGrid in 2012

• Centers operate machines, and XSEDE provides seamless infrastructure for allocations, access, and training

• Researchers propose resource use through XRAS

• Supports thousands of scientists in fields such as:
  – Chemistry
  – Bioinformatics
  – Materials Science
  – Data Sciences
XSEDE Allocations

• Want to use XSEDE resources to teach a class?
  – https://portal.xsede.org/allocations-overview#types-education

• Just looking to try out a larger resource or a special resource your campus doesn’t have?
  – https://portal.xsede.org/allocations-overview#types-startup
XSEDE Allocations

• See a Campus Champion
  – https://www.xsede.org/current-champions
• Ready to scale up your research?
  – https://portal.xsede.org/allocations-overview#types-research
More “helpful” resources
xsede.org → User Services

- Resources available at each Service Provider
  - User Guides describing memory, number of CPUs, file systems, etc.
  - Storage facilities
  - Software (Comprehensive Search)

- Training: portal.xsede.org → Training
  - Course Calendar
  - On-line training
  - Certifications

- Get face-to-face help from XSEDE experts at your institution; contact your local Campus Champions.

- Extended Collaborative Support (formerly known as Advanced User Support (AUSS))
Part I

Basics
1 Introduction
- A 5 Minute Introduction to R
- R is for Lunatics
- R Resources
- Summary

2 Debugging

3 Profiling
Introduction

- A 5 Minute Introduction to R
- R is for Lunatics
- R Resources
- Summary
Introduction

A 5 Minute Introduction to R

Types

- logical ("boolean")
- integer (32-bit int)
- numeric (double)
- complex (double complex)
- character (string)
Happy Opposite Day!

```
T
# [1] TRUE
F
# [1] FALSE
T <- FALSE
F <- TRUE
T
# [1] FALSE
F
# [1] TRUE
```
Introduction

A 5 Minute Introduction to R

Package or Library?

- I wrote a library.
- I put that library into a package.
- I installed the package . . . into a library.
- I load the package with `library()` ???
*BOOM*
1 Introduction
- A 5 Minute Introduction to R
- R is for Lunatics
- R Resources
- Summary
“R is a shockingly dreadful language for an exceptionally useful data analysis environment.” — Tim Smith, from aRrgh: a newcomer’s (angry) guide to R.
But you can’t deny its popularity!

IEEE Spectrum’s 2014 Ranking of Programming Languages

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Java</td>
<td>![Network Icon] ![Mobile Icon] ![Desktop Icon]</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>![Mobile Icon] ![Desktop Icon] ![Server Icon]</td>
<td>99.3</td>
</tr>
<tr>
<td>3. C++</td>
<td>![Mobile Icon] ![Desktop Icon] ![Server Icon]</td>
<td>95.5</td>
</tr>
<tr>
<td>4. Python</td>
<td>![Network Icon] ![Desktop Icon]</td>
<td>93.4</td>
</tr>
<tr>
<td>5. C#</td>
<td>![Network Icon] ![Mobile Icon] ![Desktop Icon]</td>
<td>92.4</td>
</tr>
<tr>
<td>6. PHP</td>
<td>![Network Icon]</td>
<td>84.7</td>
</tr>
<tr>
<td>7. Javascript</td>
<td>![Network Icon] ![Mobile Icon]</td>
<td>84.4</td>
</tr>
<tr>
<td>8. Ruby</td>
<td>![Network Icon]</td>
<td>78.8</td>
</tr>
<tr>
<td>9. R</td>
<td>![Desktop Icon]</td>
<td>74.2</td>
</tr>
<tr>
<td>10. MATLAB</td>
<td>![Desktop Icon]</td>
<td>72.9</td>
</tr>
</tbody>
</table>

See:

http://spectrum.ieee.org/static/interactive-the-top-programming-languages#index
Top Data Analysis Tool

You are without doubt the worst programming language I've ever heard of.
But you HAVE heard of me!
Why use R at all?

- Most diverse set of statistical methods available.
- Rapid prototyping.
- CRAN (and increasingly GitHub) packages.
- *Awesome* community.
- Syntax is designed for analysis of data.
Introduction
- A 5 Minute Introduction to R
- R is for Lunatics
- R Resources
- Summary
Resources for Learning R


- R programming for those coming from other languages: [http://www.johndcook.com/R_language_for_programmers.html](http://www.johndcook.com/R_language_for_programmers.html)

Other Invaluable Resources

- **R Installation and Administration:**
  [http://cran.r-project.org/doc/manuals/R-admin.html](http://cran.r-project.org/doc/manuals/R-admin.html)

- **Task Views:** [http://cran.at.r-project.org/web/views](http://cran.at.r-project.org/web/views)

- **Writing R Extensions:**
  [http://cran.r-project.org/doc/manuals/R-exts.html](http://cran.r-project.org/doc/manuals/R-exts.html)


- The [R] stackoverflow tag.

- The #rstats hashtag on Twitter.
Introduction

- A 5 Minute Introduction to R
- R is for Lunatics
- R Resources
- Summary
Summary

- R is more data analysis package than programming language.
- But you can’t deny its popularity!
1 Introduction

2 Debugging
   - Debugging R Code
   - The R Debugger
   - Debugging Compiled Code Called by R Code
   - Summary

3 Profiling
2 Debugging

- Debugging R Code
- The R Debugger
- Debugging Compiled Code Called by R Code
- Summary
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:

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

Examining the structure of an R object:

```r
> 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()`:

```r
df <- data.frame(x=rnorm(10), y=rnorm(10))
mdl <- lm(y~x, data=df) ### That’s a "tilde" character

mdl
print(mdl)
str(mdl)
unclass(mdl)
```

Try it!
2 Debugging

- Debugging R Code
- The R Debugger
- Debugging Compiled Code Called by R Code
- Summary
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

```r
> 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)
design at #1: {
  y <- z + 1
  z <- y * 2
  z
}
Browse [2]>
design at #1: y <- z + 1
Browse [2]>
Error in f(1) : object 'z' not found
> 
```
2 Debugging

- Debugging R Code
- The R Debugger
- Debugging Compiled Code Called by R Code
- Summary
Reasonably easy to use gdb and Valgrind (from command line).

- gdb — The GNU Debugger; general purpose debugging.
- Valgrind — Memory debugger.
- For gdb, start R interactively.
- For Valgrind, need a batch script.
Debugging with gdb

Suppose we have:

- R function: `fooR()`
- Calls the C function: `fooC()`

We can debug `fooC()` via gdb by executing the following from a shell:

```
1  R -d gdb
2  b fooC
3  signal 0
4  fooR(10)
```
Debugging with Valgrind

Put the R code you wish to profile in `myscript.r` and execute the following from a shell:

```
R -d "valgrind --tool=memcheck --leak-check=full" --vanilla <
myscript.r
```
2 Debugging
- Debugging R Code
- The R Debugger
- Debugging Compiled Code Called by R Code
- Summary
R has sophisticated debugging utilities for dealing with buggy R code. (
\texttt{debug()}, \texttt{str()}, \ldots).  
Using \texttt{gdb} is awkward, but possible.  
Using \texttt{Valgrind} is straight-forward.
1 Introduction

2 Debugging

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

- Why Profile?
- Profiling R Code
- Advanced R Profiling
- Summary
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. \ldots \ 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
Why Profile?

- Because performance matters.
- Bad practices scale up!
- Your bottlenecks may surprise you.
- Because R is dumb.
- R users claim to be data people... so act like it!
Compilers often correct bad behavior...

A Really Dumb Loop

```c
int main()
{
    int x, i;
    for (i = 0; i < 10; i++)
        x = 1;
    return 0;
}
```

clang -O3 -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
```
Profiling

Why Profile?

R will not!

Dumb Loop

```r
for (i in 1:n){
  tA <- t(A)
  Y <- tA %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}
```

Better Loop

```r
tA <- t(A)
for (i in 1:n){
  Y <- tA %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}
```
Example from a Real R Package

Excerpt from Original function

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

Excerpt from Modified function

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

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.
3 Profiling

- Why Profile?
- Profiling R Code
- Advanced R Profiling
- Summary
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: \texttt{system.time()}

\texttt{system.time()} is a basic R utility for timing expressions

```r
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
```
Profiling R Code

Performance Profiling Tools: `system.time()`

Put more complicated expressions inside of brackets:

```r
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()`

```r
Rprof(filename="Rprof.out", append=FALSE, interval=0.02,
      memory.profiling=FALSE, gc.profiling=FALSE,
      line.profiling=FALSE, numfiles=100L, bufsize=10000L)
```
NO FAIR!

YOU CHANGED THE OUTCOME BY MEASURING IT
Performance Profiling Tools: `Rprof()`

```r
x <- matrix(rnorm(10000*250), nrow=10000, ncol=250)

Rprof()
invisible(prcomp(x))
Rprof(NULL)

summaryRprof()

Rprof(interval=.99)
invisible(prcomp(x))
Rprof(NULL)

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()`

```r
$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
```
Profiling

- Why Profile?
- Profiling R Code
- Advanced R Profiling
- Summary
## Other Profiling Tools

- perf, PAPI
- fpmpi, mpiP, TAU
- pbdPROF
- pbdPAPI
Profiling MPI Codes with pbdPROF

1. Rebuild pbdR packages

```
R CMD INSTALL pbdMPI_0.2-1.tar.gz \n  --configure-args="--enable-pbdPROF"
```

2. Run code

```
mpirun -np 64 Rscript my_script.R
```

3. Analyze results

```
library(pbdPROF)
prof <- read.prof("output.mpiP")
plot(prof, plot.type="messages2")
```
Profiling with **pbdPAPI**

- Bindings for Performance Application Programming Interface (PAPI)
- Gathers detailed hardware counter data.
- High and low level interfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description of Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>system.flips()</code></td>
<td>Time, floating point instructions, and Mflips</td>
</tr>
<tr>
<td><code>system.flops()</code></td>
<td>Time, floating point operations, and Mflops</td>
</tr>
<tr>
<td><code>system.cache()</code></td>
<td>Cache misses, hits, accesses, and reads</td>
</tr>
<tr>
<td><code>system.epc()</code></td>
<td>Events per cycle</td>
</tr>
<tr>
<td><code>system.idle()</code></td>
<td>Idle cycles</td>
</tr>
<tr>
<td><code>system.cpuormem()</code></td>
<td>CPU or RAM bound*</td>
</tr>
<tr>
<td><code>system.utilization()</code></td>
<td>CPU utilization*</td>
</tr>
</tbody>
</table>
Profiling with \texttt{pbdPAPI}

```r
x <- system.cache(rnorm(1e5), type="miss")
x

# L1 Cache Misses: 15186
# L2 Cache Misses: 3550
# L3 Cache Misses: 1241

plot(x)
```
Profiling with pbdPAPI

```r
y <- system.cache(rnorm(5e5), type="miss")
plot(x, y)
```
pbdPAPI

To learn more about pbdPAPI, see:

- Guide to the pbdPAPI Package
- Advanced R Profiling with pbdPAPI
- Cache Rules Everything Around Me
3 Profiling

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

Summary
Summary

- *Profile, profile, profile.*
- Use `system.time()` to get a general sense of a method.
- Use `Rprof()` for more detailed profiling.
- Other tools exist for more hardcore applications (e.g., `pbdPAPI` and `pbdPROF`).
Exercises
Part II

Improving R Performance
Benchmarking
  • Benchmarking
  • Summary

Free Improvements

Writing Better R Code
4 Benchmarking

- Benchmarking
- Summary
There’s a lot that goes on when executing an R function.

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.
rbenchmark is a simple package that easily benchmarks different functions:

```r
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:

```r
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

I generally prefer `rbenchmark`, but the built-in plots for `microbenchmark` are nice:

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

Summary
Don’t just time 1 evaluation to compare 2 methods.
You could write the stuff yourself easily enough...
But `rbenchmark` and `microbenchmark` already exist and work very well.
4 Benchmarking

5 Free Improvements
- Building R with a Different Compiler
- The Bytecode Compiler
- Choice of BLAS Library
- Summary

6 Writing Better R Code
Free Improvements

- Building R with a Different Compiler
- The Bytecode Compiler
- Choice of BLAS Library
- Summary
**Better Compiler**

- GNU (gcc/gfortran) and clang/gfortran are free and will compile anything, but don’t produce the fastest binaries.
- Don’t even bother with MSVC.
- Intel icc is very fast on intel hardware.

Better compiler  $\implies$ Faster R
### Compiling R with icc and ifort

- Faster, but not painless.
- Requires Intel Composer suite license ($$$).
- Improvements are most visible on Intel hardware.
- See [Intel’s help pages](https://www.intel.com) for details.
Free Improvements

- Building R with a Different Compiler
- The Bytecode Compiler
- Choice of BLAS Library
- Summary
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

```r
library(compiler)

test <- function (x) x+1

test <- cmpfun(test)

disassemble(test)
```

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

# < bytecode: 0x38c86c8 >

# 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

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

From The Shell

```bash
export R_COMPILE_PKGS=1
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?

```r
f <- function(n) for (i in 1:n) 2*(3+4)

library(compiler)
f_comp <- cmpfun(f)

library(rbenchmark)
n <- 100000
benchmark(f(n), f_comp(n), columns=c("test", "replications", "elapsed", "relative"), order="relative")
```

# test replications elapsed relative
# 2 f_comp(n) 100 2.604 1.000
# 1 f(n) 100 2.845 1.093
The Compiler: How much does it help *really*?

```r
# Free Improvements

```g <- function(n)
{
x <- matrix(runif(n*n), nrow=n, ncol=n)
  min(colSums(x))
}

library(compiler)
g_comp <- cmpfun(g)

library(rbenchmark)

n <- 1000
benchmark(g(n), g_comp(n), columns=c("test", "replications", "elapsed", "relative"),
  order="relative")
```

# test replications elapsed relative
# 2 g_comp(n) 100 6.854 1.000
# 1 g(n) 100 6.860 1.001

---

nimbios.org/tutorials/TT_RforHPC  
Drew Schmidt  
High Performance Computing with R  
59/172
Free Improvements

- Building R with a Different Compiler
- The Bytecode Compiler
- Choice of BLAS Library
- Summary
The BLAS

- Basic Linear Algebra Subprograms.
- Basic numeric matrix operations.
- Used to compute matrix factorizations (LAPACK).
- Used in linear algebra and many statistical operations.
- Different implementations available.
- Several multithreaded BLAS libraries exist.
Comparison of Different BLAS Implementations for Matrix–Matrix Multiplication and SVD

- `x%*%x` on 2000x2000 matrix (~31 MiB)
- `x%*%x` on 4000x4000 matrix (~122 MiB)
- `svd(x)` on 1000x1000 matrix (~8 MiB)
- `svd(x)` on 2000x2000 matrix (~31 MiB)

**R Code:**

```r
set.seed(1234)
m <- 2000
n <- 2000
x <- matrix(rnorm(m*n), m, n)
object.size(x)
library(rbenchmark)
benchmark(x%*%x)
benchmark(svd(x))
```
Using Parallel BLAS

- See the *R Installation and Administration* manual for info.
- **Warning:** doesn’t always play nice with the `parallel` package!
5 Free Improvements

- Building R with a Different Compiler
- The Bytecode Compiler
- Choice of BLAS Library

Summary
Summary

- Compiling R itself with a different compiler can improve performance, but is non-trivial.
- The compiler package offers small, but free speedup.
- The (bytecode) compiler works best on loops.
4 Benchmarking

5 Free Improvements

6 Writing Better R Code
   - Loops
   - Ply Functions
   - Vectorization
   - Loops, Plys, and Vectorization
   - Summary
Writing Better R Code

- Loops
- Ply Functions
- Vectorization
- Loops, Plys, and Vectorization
- Summary
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!
Loops 1

```r
square_loop_noinit <- function(n){
  x <- c()
  for (i in 1:n){
    x <- c(x, i^2)
  }
  x
}

square_loop_withinit <- function(n){
  x <- integer(n)
  for (i in 1:n){
    x[i] <- i^2
  }
  x
}
```

Loops 2

```r
library(rbenchmark)
n <- 1000

benchmark(square_loop_noinit(n), square_loop_withinit(n))
# test replications elapsed relative
# 1 square_loop_noinit(n) 100 0.257 2.596
# 2 square_loop_withinit(n) 100 0.099 1.000
```
Writing Better R Code

- Loops
- Ply Functions
- Vectorization
- Loops, Plys, and Vectorization
- Summary
“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()`.
Writing Better R Code

Ply Functions

Ply Examples: apply()

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

x
# [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,] 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...

```r
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="")))))),)
```
6 Writing Better R Code

- Loops
- Ply Functions
- Vectorization
- Loops, Plys, and Vectorization
- Summary
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.

```r
ply <- function(x) lapply(rep(1, 1000), rnorm)
vec <- function(x) rnorm(1000)

library(rbenchmark)
benchmark(ply(x), vec(x))

# test replications elapsed relative
# 1 ply(x) 100 0.348 38.667
# 2 vec(x) 100 0.009 1.000
```
Vectorization Best Practices

- Vectorize if at all possible.
- Note that this consumes potentially a lot of memory!
6 Writing Better R Code

- Loops
- Ply Functions
- Vectorization
- Loops, Plys, and Vectorization
- Summary
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

```r
# Writing Better R Code

Loops, Plys, and Vectorization

```squares

```r
# 1
square_sapply <- function(n) sapply(1:n, function(i) i^2)

# 2
square_vec <- function(n) (1:n)*(1:n)

# Library (rbenchmark)

library(rbenchmark)

n <- 100000

benchmark(square_loop_noinit(n), square_loop_withinit(n),
          square_sapply(n), square_vec(n))

# test replications elapsed relative
# 1 square_loop_noinit(n) 100 17.296 2470.857
# 2 square_loop_withinit(n) 100 0.933 133.286
# 3 square_sapply(n) 100 1.218 174.000
# 4 square_vec(n) 100 0.007 1.000

```

nimbios.org/tutorials/TT_RforHPC

Drew Schmidt

High Performance Computing with R
Writing Better R Code

- Loops
- Ply Functions
- Vectorization
- Loops, Plys, and Vectorization
- Summary
Summary

- Pre-allocate your data in loops.
- Vectorize when you can.
- Try a ply function when you can’t.
Exercises
Part III

Interfacing to Compiled Code
Introduction to Rcpp
- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help

Using Rcpp

The Typical Monte Carlo Simulation for Estimating π

Computing the Cosine Similarity Matrix
Introduction to Rcpp

- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help
What Language is R Written In?

- R is mostly a C program
- R extensions are mostly R programs

**Percent of Core R Lines of Code**

- R
- C
- Fortran

**Percent Contribution of Language to Contrib**

- R
- C
- C++
- Fortran
- Perl
Foreign Language Interfaces

- C/C++: `.Call()`, `.C()` (deprecated)
- Fortran: `.Call()`, `.Fortran()` (deprecated)
- Java: rJava package
- Python: rPython package
- ...

For the remainder, we will focus on C++ via Rcpp.
Introduction to Rcpp

- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help
### What Rcpp is

- R interface to compiled code.
- Package ecosystem (Rcpp, RcppArmadillo, RcppEigen, ...).
- Utilities to make writing C++ more convenient for R users.
- **A tool which requires C++ knowledge to effectively utilize.**
- GPL licensed (like R).
What Rcpp **is not**

- Magic.
- Automatic R-to-C++ converter.
- A way around having to learn C++.
- A tool to make existing R functionality faster (unless you rewrite it!).
- As easy to use as R.
Advantages of Rcpp

- Compiled code is fast.
- Easy to install.
- Easy to use (comparatively).
- Better documented than alternatives.
- Large, friendly, helpful community.
Rcpp Package Dependencies
## Disadvantages

- It’s C++ (there be dragons).
- Difficult to debug/profile.
- Rcpp designed to only work with R.
Introduction to Rcpp

- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help
Documentation

- The numerous Rcpp vignettes
  [http://cran.r-project.org/web/packages/Rcpp/index.html](http://cran.r-project.org/web/packages/Rcpp/index.html)
  (start with Introduction, quickref, and FAQ).

- **High Performance Functions with Rcpp**, Hadley Wickham:
  [http://adv-r.had.co.nz/Rcpp.html](http://adv-r.had.co.nz/Rcpp.html)

Where to Get Help

- The documentation.
- The [rcpp] tag on stackoverflow.
- Rcpp-devel list: http://lists.r-forge.r-project.org/mailman/listinfo/rcpp-devel
Advice

<table>
<thead>
<tr>
<th>New to C++?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Get a good book on just C++.</td>
</tr>
<tr>
<td>• Be patient. C++ is really hard.</td>
</tr>
<tr>
<td>• Learn the art of reading template explosions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Know R?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Never use . in object names.</td>
</tr>
<tr>
<td>• Lines end with ;.</td>
</tr>
<tr>
<td>• Returns of functions must be explicitly named.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Know C++?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No voids.</td>
</tr>
<tr>
<td>• If data is modified, do it in a copy.</td>
</tr>
<tr>
<td>• R functions are not thread safe!!!</td>
</tr>
</tbody>
</table>
Introduction to Rcpp

Using Rcpp
- C vs Rcpp
- Using Rcpp with R

The Typical Monte Carlo Simulation for Estimating $\pi$

Computing the Cosine Similarity Matrix
Using Rcpp

- C vs Rcpp
- Using Rcpp with R
C/C++ API’s and Extensions for R

- The native C interface.
- Rcpp
  - RcppArmadillo
  - RcppCNPy
  - RcppEigen
- Rcpp11, Rcpp14, ...
- RcppGSL
- RcppRedis
- ...

nimbios.org/tutorials/TT_RforHPC  Drew Schmidt  High Performance Computing with R  91/172
To see the difference, let’s construct:

```r
list(a=1L, b=2.0)
```

using the native C interface and with Rcpp.
The C Interface

```c
#include <R.h>
#include <Rinternals.h>

SEXP examplefun(){
  SEXP ret, retnames, a, b;
  PROTECT(a = allocVector(INTSXP, 1));
  PROTECT(b = allocVector(REALSXP, 1));

  INTEGER(a)[0] = 1;
  REAL(b)[0] = 2.0;

  PROTECT(ret = allocVector(VECSXP, 2));
  SET_VECTOR_ELT(ret, 0, a);
  SET_VECTOR_ELT(ret, 1, b);

  PROTECT(retnames = allocVector(STRSXP, 2));
  SET_STRING_ELT(retnames, 0, mkChar("a"));
  SET_STRING_ELT(retnames, 1, mkChar("b"));
  setAttrib(ret, R_NamesSymbol, retnames);

  UNPROTECT(4);
  return ret;
}
```
```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::List examplefun()
{
    Rcpp::IntegerVector a(1);
    Rcpp::NumericVector b(1);

    a[0] = 1;
    b[0] = 2.0;

    Rcpp::List ret =
        Rcpp::List::create(Rcpp::Named("a") = a,
                           Rcpp::Named("b") = b);

    return ret;
}
```
I can’t in good conscience describe C++ as *good for beginners*. Rcpp is cleaner. Like C++? You’ll *love* Rcpp.
Using Rcpp

- C vs Rcpp
- Using Rcpp with R
Rcpp

What about compiling, linking, loading, wrapping, etc?
Building with Rcpp

We will be using `sourceCpp()` to build our examples:

1. Create C++ function as string in R.
2. Use `sourceCpp` to generate wrapper.
3. Call your function in R.
sourceCpp(): Create C++ Function

code <- '   
#include <Rcpp.h>

// [[Rcpp::export]]
int plustwo(int n)
{
  return n+2;
}
,'
sourceCpp(): Use sourceCpp

```
1 library(Rcpp)
2 sourceCpp(code=code)
```
sourceCpp(): Call Your Function in R

```r
plustwo(1)
# [1] 3
```
Introduction to Rcpp

Using Rcpp

The Typical Monte Carlo Simulation for Estimating $\pi$
- Background and Outline
- Implementation
- Summary

Computing the Cosine Similarity Matrix
The Typical Monte Carlo Simulation for Estimating $\pi$

- Background and Outline
- Implementation
- Summary
Example 1: Monte Carlo Simulation to Estimate $\pi$

Sample $N$ uniform observations $(x_i, y_i)$ in the unit square $[0, 1] \times [0, 1]$. Then

$$\pi \approx 4 \left( \frac{\# \text{ Inside Circle}}{\# \text{ Total}} \right) = 4 \left( \frac{\# \text{ Blue}}{\# \text{ Blue} + \# \text{ Red}} \right)$$
The Typical Monte Carlo Simulation for Estimating $\pi$

Outline

1. Implement in R using loops.
2. Implement in R using vectorization.
3. Implement in C++ with Rcpp.
4. Benchmark.
5. Examine other performance considerations.
The Typical Monte Carlo Simulation for Estimating $\pi$

- Background and Outline
- Implementation
- Summary
Example 1: Monte Carlo Simulation Code

R Code (loops)

```r
mcsim_r <- function (n)
{
  r <- 0L

  for (i in 1:n){
    u <- runif(1)
    v <- runif(1)

    if (u^2 + v^2 <= 1)
      r <- r + 1
  }

  return( 4*r/n )
}
```
Example 1: Monte Carlo Simulation Code

R Code (vectorized)

```r
mcsim_r_vec <- function(n) {
  x <- matrix(runif(n * 2), ncol=2)
  r <- sum(rowSums(x^2) <= 1)
  return(4*r/n)
}
```
The Typical Monte Carlo Simulation for Estimating $\pi$

Implementation

Example 1: Monte Carlo Simulation Code

Rcpp Code

```r
code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
double mcsim_rcpp(const int n) {
    int i, r = 0;
    double u, v;

    for (i=0; i<n; i++){
        u = R::runif(0, 1);
        v = R::runif(0, 1);

        if (u*u + v*v <= 1)
            r++;
    }

    return (double) 4.*r/n;
}
"

library(Rcpp)
sourceCpp(code=code)
```

Example 1: Monte Carlo Simulation Code

Benchmarking the Methods

```r
library(rbenchmark)

n <- 100000L

benchmark(R.loop = mcsim_r(n),
          R.vec = mcsim_r_vec(n),
          C = mcsim_c(n),
          Rcpp = mcsim_rcpp(n),
          columns=c("test", "replications", "elapsed", "relative"))
```

<table>
<thead>
<tr>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rcpp</td>
<td>100</td>
<td>0.309</td>
<td>1.000</td>
</tr>
<tr>
<td>R.loop</td>
<td>100</td>
<td>65.543</td>
<td>212.113</td>
</tr>
<tr>
<td>R.vec</td>
<td>100</td>
<td>1.989</td>
<td>6.437</td>
</tr>
</tbody>
</table>
```
Example 1: Monte Carlo Simulation Code

Benchmarking the Methods

```r
library(rbenchmark)

n <- 1000000L

benchmark(R.vec = mcsim_r_vec(n),
          Rcpp = mcsim_rcpp(n),
          columns=c("test", "replications", "elapsed",
                   "relative"))
```

<table>
<thead>
<tr>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rcpp</td>
<td>100</td>
<td>30.825</td>
<td>1.000</td>
</tr>
<tr>
<td>R.vec</td>
<td>100</td>
<td>135.075</td>
<td>4.382</td>
</tr>
</tbody>
</table>
The Typical Monte Carlo Simulation for Estimating \( \pi \)

### Implementation

What About the Compiler?

#### Benchmarking the Methods

```r
library(rbenchmark)
library(compiler)

mcsim_r <- cmpfun(mcsim_r)
mcsim_r_vec <- cmpfun(mcsim_r_vec)
mcsim_rcpp <- cmpfun(mcsim_rcpp)

n <- 100000L

benchmark(R.loop = mcsim_r(n),
          R.vec = mcsim_r_vec(n),
          Rcpp = mcsim_rcpp(n),
          columns=c("test", "replications", "elapsed", "relative"))
```

<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Rcpp</td>
<td>100</td>
<td>0.311</td>
<td>1.000</td>
</tr>
<tr>
<td>1</td>
<td>R.loop</td>
<td>100</td>
<td>55.125</td>
<td>177.251</td>
</tr>
<tr>
<td>2</td>
<td>R.vec</td>
<td>100</td>
<td>1.107</td>
<td>3.559</td>
</tr>
</tbody>
</table>
```

nimbios.org/tutorials/TT_RforHPC

Drew Schmidt

High Performance Computing with R

108/172
## Memory Usage in Bytes (roughly)

### Loops:

\[ 4(n + 3) + 8 \cdot 3 \]

- **Integers**: \(4n\)
- **Doubles**: \(8 \cdot 3\)

### Vectorized:

\[ 4n + 8(2 + 2n) \]

- **Integers**: \(4n\)
- **Doubles**: \(8(2 + 2n)\)

### Rcpp:

\[ 4 \cdot 3 + 8 \cdot 3 \]

- **Integers**: \(4 \cdot 3\)
- **Doubles**: \(8 \cdot 3\)
The Typical Monte Carlo Simulation for Estimating \( \pi \)

- Background and Outline
- Implementation
- Summary
### Summary

For $n = 100,000$ iterations and 100 replicates:

<table>
<thead>
<tr>
<th></th>
<th>Loops</th>
<th>Vectorized</th>
<th>Rcpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg Runtime (seconds)</td>
<td>0.65543</td>
<td>0.01999</td>
<td>0.00309</td>
</tr>
<tr>
<td>Avg Compiled Runtime (seconds)</td>
<td>0.55125</td>
<td>0.1107</td>
<td>0.00311</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>1.526 MiB</td>
<td>13.733 MiB</td>
<td>36 bytes</td>
</tr>
</tbody>
</table>

Processor: Core i5 Sandy Bridge
R Version: 3.1.2
C++ Compiler: clang++ 3.5.0
CXX Flags: -03 -fpic
Some Thoughts

- Bad R often looks like good C/C++.
- The bytecode compiler helps, but not much.
- R’s memory footprint is terrible.
Computing the Cosine Similarity Matrix

Introduction to Rcpp

Using Rcpp

The Typical Monte Carlo Simulation for Estimating $\pi$

Computing the Cosine Similarity Matrix
- Background and Outline
- Implementation
- Benchmarks
- Summary
Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary
Cosine Similarity

Recall from vector calculus that for vectors $x$ and $y$

$$
cos(x, y) = \|x\| \|y\| \cos(\theta(x, y))
$$

We define

$$
cosim(x, y) := cos(\theta(x, y)) = \frac{x \cdot y}{\|x\| \|y\|}
$$
The cosine similarity matrix of a given (possibly non-square) matrix is the matrix of all pairwise similarities of the columns, i.e., given

\[ X_{n,p} = [x_1, \ldots, x_p] \]

We take

\[ \text{cosim}(X)_{ij} = \text{cosim}(x_i, x_j) \]
Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary
Computing the Cosine Similarity Matrix

Original implementation

From CRAN’s lsa package version 0.73 (in R/lsa.R)

```r
cosine <- function (x, y = NULL) {
  if (is.matrix(x) && is.null(y)) {
    co = array(0, c(ncol(x), ncol(x)))
    f = colnames(x)
    dimnames(co) = list(f, f)
    for (i in 2:ncol(x)) {
      for (j in 1:(i - 1)) {
        co[i, j] = cosine(x[, i], x[, j])
      }
    }
    co = co + t(co)
    diag(co) = 1
    return(as.matrix(co))
  }
  else if (is.vector(x) && is.vector(y))
    return(crossprod(x, y)/sqrt(crossprod(x) * crossprod(y)))
  else
    stop("argument mismatch.")
}
```
Computing the Cosine Similarity Matrix

Implementation

R Improvements 1

cosine_loop <- function(x){
  cp <- crossprod(x)
  dg <- diag(cp)

  co <- matrix(0.0, length(dg), length(dg))

  for (j in 2L:length(dg)){
    for (i in 1L:(j-1L)){
      co[i, j] <- cp[i, j] / sqrt(dg[i] * dg[j])
    }
  }

  co <- co + t(co)
  diag(co) <- 1.0

  return( co )
}
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericMatrix fill_loop(Rcpp::NumericMatrix cp,
   Rcpp::NumericVector dg){
const unsigned int n = cp.nrow();
Rcpp::NumericMatrix co(n, n);

  // Fill lower triangle and diagonal
  for (int j=0; j<n; j++){
    for (int i=0; i<=j; i++){  
      if (i == j)
        co(j, j) = 1.0;
      else
        co(i, j) = cp(i, j) / std::sqrt(dg[i] * dg[j]);
    }
  }
"

Computing the Cosine Similarity Matrix

Implementation

Rcpp 2

```
// Copy lower triangle to upper
for (int j=0; j<n; j++){
    for (int i=j+1; i<n; i++)
        co(i, j) = co(j, i);
}

return co;
```

```
sourceCpp(code=code)
```

```
cosine_Rcpp <- function(x){
  cp <- crossprod(x)
  dg <- diag(cp)
  co <- fill_loop(cp, dg)
  return( co )
}
```

Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary
library(rbenchmark)

reps <- 10

for (i in 1:10) {
  n <- i*100
  x <- matrix(rnorm(n*n), n, n)

  benchmark(cosine(x), cosine_loop(x), cosine_Rcpp(x),
            replications=reps, columns=c("test",
                                     "relative"))
}
## Relative Performance

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>cosine()</th>
<th>cosine_loop()</th>
<th>cosine_Rcpp()</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>340</td>
<td>44.5</td>
<td>1</td>
</tr>
<tr>
<td>200x200</td>
<td>535.167</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>300x300</td>
<td>441.632</td>
<td>42.895</td>
<td>1</td>
</tr>
<tr>
<td>400x400</td>
<td>495.176</td>
<td>42.412</td>
<td>1</td>
</tr>
<tr>
<td>500x500</td>
<td>519.877</td>
<td>41.456</td>
<td>1</td>
</tr>
<tr>
<td>600x600</td>
<td>512.264</td>
<td>36.758</td>
<td>1</td>
</tr>
<tr>
<td>700x700</td>
<td>392.114</td>
<td>25.486</td>
<td>1</td>
</tr>
<tr>
<td>800x800</td>
<td>474.341</td>
<td>28.498</td>
<td>1</td>
</tr>
<tr>
<td>900x900</td>
<td>523.841</td>
<td>29.367</td>
<td>1</td>
</tr>
<tr>
<td>1000x1000</td>
<td>459.322</td>
<td>23.995</td>
<td>1</td>
</tr>
</tbody>
</table>
### Relative Performance with Bytecode Compilation

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>cosine()</th>
<th>cosine_loop()</th>
<th>cosine_Rcpp()</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>300</td>
<td>25.5</td>
<td>1</td>
</tr>
<tr>
<td>200x200</td>
<td>360.25</td>
<td>25.125</td>
<td>1</td>
</tr>
<tr>
<td>300x300</td>
<td>454.059</td>
<td>29.941</td>
<td>1</td>
</tr>
<tr>
<td>400x400</td>
<td>252.885</td>
<td>14.705</td>
<td>1</td>
</tr>
<tr>
<td>500x500</td>
<td>315.518</td>
<td>17.671</td>
<td>1</td>
</tr>
<tr>
<td>600x600</td>
<td>323.662</td>
<td>15.398</td>
<td>1</td>
</tr>
<tr>
<td>700x700</td>
<td>430.507</td>
<td>18.169</td>
<td>1</td>
</tr>
<tr>
<td>800x800</td>
<td>385.504</td>
<td>15.043</td>
<td>1</td>
</tr>
<tr>
<td>900x900</td>
<td>469.728</td>
<td>16.709</td>
<td>1</td>
</tr>
<tr>
<td>1000x1000</td>
<td>505.706</td>
<td>16.625</td>
<td>1</td>
</tr>
</tbody>
</table>
Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary
Summary

- Bad R often looks like good C/C++.
- Compiled code can be much faster than R code.
- Vectorized code better than loops, but worse than more tailored compiled code.
Exercises
Part IV

Parallelism
An Overview of Parallelism
- Terminology: Parallelism
- Guidelines
- Summary

Shared Memory Parallelism in R

Distributed Memory Parallelism with R

The pbdR Project

Distributed Matrices
An Overview of Parallelism

- Terminology: Parallelism
- Guidelines
- Summary
Parallelism

Serial Programming

Parallel Programming

- Problem
- CPU

- Problem
- CPU 1
  - Sub Problem 1
- CPU 2
  - Sub Problem 2
An Overview of Parallelism

Parallelism

Serial Programming

Parallel Programming

Serial Programming:
- **make_lunch**
  - Get resources
  - Work
  - Work
  - Return

Parallel Programming:
- **mpirun -np 2 make_lunch_par**
  - **make_lunch_par**
    - Get resources
    - Work
    - Work
  - combine
  - Return
Parallel Programming Vocabulary: Difficulty in Parallelism

1. *Implicit parallelism*: Parallel details hidden from user
   Example: Using multi-threaded BLAS

2. *Explicit parallelism*: Some assembly required...
   Example: Using the `mclapply()` from the `parallel` package

3. *Embarrassingly Parallel* or *loosely coupled*: Obvious how to make parallel; lots of independence in computations.
   Example: Fit two independent models in parallel.

   Example: Speed up model fitting for one model.
An Overview of Parallelism

Terminology: Parallelism

- **Wallclock Time**: Time of the clock on the wall from start to finish
- **Speedup**: Unitless measure of improvement; more is better.

\[
S_{n_1,n_2} = \frac{\text{Time for } n_1 \text{ cores}}{\text{Time for } n_2 \text{ cores}}
\]

- \(n_1\) is often taken to be 1
- In this case, comparing parallel algorithm to serial algorithm
An Overview of Parallelism

Terminology: Parallelism

Speedup

Good Speedup

Bad Speedup

Cores

Speedup

Application

Optimal

nimbios.org/tutorials/TT_RforHPC  Drew Schmidt  High Performance Computing with R
## Shared and Distributed Memory Machines

<table>
<thead>
<tr>
<th><strong>Shared Memory</strong></th>
<th><strong>Distributed</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct access to read/change memory (one node)</td>
<td>No direct access to read/change memory (many nodes); requires communication</td>
</tr>
<tr>
<td>Examples: laptop, GPU, MIC</td>
<td>Examples: cluster, server, supercomputer</td>
</tr>
</tbody>
</table>

**Shared Memory**
- Memory
  - CPU
  - CPU
  - CPU
  - CPU

**Distributed**
- Memory
  - CPU
  - CPU
  - CPU
  - CPU

Network
An Overview of Parallelism

Shared and Distributed Memory Machines

Shared Memory Machines

Thousands of cores

Nautilus, University of Tennessee
1024 cores
4 TB RAM

Distributed Memory Machines

Hundreds of thousands of cores

Titan, Oak Ridge National Lab
299,008 cores
584 TB RAM
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](http://cran.r-project.org/web/views/HighPerformanceComputing.html)
An Overview of Parallelism

Terminology: Parallelism

Parallel Programming Packages for R

Distributed Memory

Interconnection Network

PROC + cache

PROC + cache

PROC + cache

PROC + cache

Mem

Mem

Mem

Mem

Shared Memory

CORE + cache

CORE + cache

CORE + cache

CORE + cache

Network

Memory

Focus on who owns what data and what communication is needed

Focus on which tasks can be parallel

Co-Processor

Same Task on Blocks of data

Local Memory

GPU: Graphical Processing Unit

MIC: Many Integrated Core

Focus on who owns what data and what communication is needed

Co-Processor

Same Task on Blocks of data

Local Memory

GPU: Graphical Processing Unit

MIC: Many Integrated Core
An Overview of Parallelism

- Terminology: Parallelism
- Guidelines
- Summary
Independence

- Parallelism requires *independence*.
- Separate evaluations of R functions is embarrassingly parallel.

![Diagram of Cores/Nodes]

- Fit Model 1
- Fit Model 2
- Fit Model 3
- Fit Model 4

Choose Best

Best
Portability

Many parallel R packages break on Windows

A fatal exception 0E has occurred at 0028:C0011E36 in UXD UMM(01) + 00010E36. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue...
RNG’s in Parallel

- Be careful!
- Aided by `rlecuyer`, `rsprng`, and `doRNG` packages.
Parallel Programming: In Theory
Parallel Programming: In Practice
An Overview of Parallelism

- Terminology: Parallelism
- Guidelines
- Summary
Many kinds of parallelism available to R.

Better/parallel BLAS is free speedup for linear algebra, but takes some work.
11 An Overview of Parallelism

12 Shared Memory Parallelism in R
   • The parallel Package
   • The foreach Package

13 Distributed Memory Parallelism with R

14 The pbdR Project

15 Distributed Matrices
Shared Memory Parallelism in R

- The parallel Package
- The foreach Package
The parallel Package

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

\[
parallel = \text{snow} + \text{multicore}
\]
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

```r
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)
```

```r
x <- lapply(1:10, sqrt)
library(parallel)
x.mc <- mclapply(1:10, sqrt)
all.equal(x.mc, x)
# [1] TRUE
```
The parallel Package: multicore

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

simplify2array(mclapply(1:2, function(i) Sys.getpid(), mc.cores=4))
# [1] 27457 2745
```
The parallel Package: snow

- Uses sockets.
  + Works on all platforms.
  - More fiddley than mclapply().
  - Not as efficient as forks.
### Set up the worker processes

```r
cl <- makeCluster(detectCores())
cl

# socket cluster with 4 nodes on host localhost

parSapply(cl, 1:5, sqrt)

stopCluster(cl)
```
## The parallel Package: Summary

### All
- `detectCores()`
- `splitIndices()`

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

### snow
- `makeCluster()`
- `stopCluster()`
- `parLapply()`
- `parSapply()`
- and others...
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 syntax.
  - Efficiency issues if you aren't careful!
Efficiency Issues

Coin Flipping with 24 Cores

### Bad performance
```r
foreach (i = 1:len)
  %dopar% tinyfun(i)
```

### Expected performance
```r
foreach (i = 1:ncores)
  %dopar% {
    out <- numeric(len/ncores)
    for (j in 1:(len/ncores))
      out[i] <- tinyfun(j)
    out
  }
```
The foreach Package: General Procedure

- Load **foreach** and your backend package.
- Register your backend.
- Call **foreach**
Using foreach: serial

```r
library(foreach)

### Example 1
foreach(i = 1:3) %do% sqrt(i)

### Example 2
n <- 50
reps <- 100
x <- foreach(i = 1:reps) %do% {
  sum(rnorm(n, mean = i)) / (n * reps)
}
```

Drew Schmidt
High Performance Computing with R
Using foreach: Parallel

```r
library(foreach)
library(<mybackend>)

register <MyBackend>()

### Example 1
foreach(i=1:3) %dopar% sqrt(i)

### Example 2
n <- 50
tests <- 100

x <- foreach(i=1:tests) %dopar% {
  sum(rnorm(n, mean=i)) / (n*tests)
}
```

nimbios.org/tutorials/TT_RforHPC
Drew Schmidt
High Performance Computing with R
foreach backends

multicore

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

snow

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

- Make sure to register your backend.
- Different backends may have different performance.
- Use `%dopar%` for parallel foreach.
- `%do%` and `%dopar%` must appear on the same line as the `foreach()` call.
11 An Overview of Parallelism

12 Shared Memory Parallelism in R

13 Distributed Memory Parallelism with R
   - Distributed Memory Parallelism
   - Rmpi
   - pbdMPI vs Rmpi
   - Summary

14 The pbdR Project

15 Distributed Matrices
Distributed Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary
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.
Hasty Explanation of MPI

- MPI = Message Passing Interface
- Recall: Distributed machines can’t directly manipulate memory of other nodes.
- Can *indirectly* manipulate them, however...
- Distinct nodes collaborate by passing messages over network.
Distributed Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary
Distributed Memory Parallelism with R

Rmpi

**Rmpi Hello World**

```r
mpi.spawn.Rslaves(nslaves=2)
#
# 2 slaves are spawned successfully. 0 failed.
#
# master (rank 0, comm 1) of size 3 is running on: wootabega
#
# slave1 (rank 1, comm 1) of size 3 is running on: wootabega
#
# slave2 (rank 2, comm 1) of size 3 is running on: wootabega
#

mpi.remote.exec(paste("I
am",mpi.comm.rank(),"of",mpi.comm.size()))
#
# $slave1
#
# [1] "I am 1 of 3"
#
#
# $slave2
#
# [1] "I am 2 of 3"

mpi.exit()
```
Using Rmpi from snow

```r
library(snow)
library(Rmpi)

cl <- makeCluster(2, type = "MPI")
clusterCall(cl, function() Sys.getpid())
clusterCall(cl, runif, 2)
stopCluster(cl)
mpi.quit()
```
Rmpi Resources

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

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary
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 Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary
Summary

- Distributed parallelism is necessary when computations no longer fit in ram.
- Several options available; most go beyond the scope of this talk.
An Overview of Parallelism

Shared Memory Parallelism in R

Distributed Memory Parallelism with R

The pbdR Project

Distributed Matrices
# Recall: Parallel R Packages

## Shared Memory
1. **foreach**
2. **parallel**
3. **snow**
4. **multicore**

## Distributed
1. **Rmpi**
2. **RHadoop, RHadoop**
3. **pbdR**

(and others...)

---

Programming with Big Data in R (pbdR)

Striving for *Productivity, Portability, Performance*

- Free³ R packages.
- Bridging high-performance compiled code with high-productivity of R
- Scalable, big data analytics.
- Offers implicit and explicit parallelism.
- Methods have syntax *identical* to R.

³MPL, BSD, and GPL licensed
pbdR Packages

pbdR

<table>
<thead>
<tr>
<th>pbdDEMO</th>
<th>pbdDMAT</th>
<th>pbdBASE</th>
<th>pbdSLAP</th>
<th>pbdMPI</th>
<th>pbdPROF</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>pbdR</th>
<th>ScaLAPACK</th>
<th>NetCDF4</th>
<th>mpi</th>
<th>LAPACK</th>
<th>PBLAS</th>
<th>BLACS</th>
<th>HDF5</th>
<th>MPI</th>
</tr>
</thead>
</table>

nimbios.org/tutorials/TT_RforHPC  Drew Schmidt  High Performance Computing with R  165/172
Distributed Matrices and Statistics with **pbdDMAT**

**Least Squares Benchmark**

```r
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”.
11 An Overview of Parallelism

12 Shared Memory Parallelism in R

13 Distributed Memory Parallelism with R

14 The pbdR Project

15 Distributed Matrices
ddmatrix: 2-dimensional Block-Cyclic with 6 Processors

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

Processor grid = \[
\begin{bmatrix}
  0 & 1 & 2 \\
  3 & 4 & 5
\end{bmatrix} = \begin{bmatrix}
  (0,0) & (0,1) & (0,2) \\
  (1,0) & (1,1) & (1,2)
\end{bmatrix}
\]
Understanding \texttt{ddmatrix}: Local View

\[
\begin{bmatrix}
  x_{11} & x_{12} & x_{17} & x_{18} \\
  x_{21} & x_{22} & x_{27} & x_{28} \\
  x_{51} & x_{52} & x_{57} & x_{58} \\
  x_{61} & x_{62} & x_{67} & x_{68} \\
  x_{91} & x_{92} & x_{97} & x_{98}
\end{bmatrix}_{5 \times 4}
\]

\[
\begin{bmatrix}
  x_{13} & x_{14} & x_{19} \\
  x_{23} & x_{24} & x_{29} \\
  x_{53} & x_{54} & x_{59} \\
  x_{63} & x_{64} & x_{69} \\
  x_{93} & x_{94} & x_{99}
\end{bmatrix}_{5 \times 3}
\]

\[
\begin{bmatrix}
  x_{15} & x_{16} \\
  x_{25} & x_{26} \\
  x_{55} & x_{56} \\
  x_{65} & x_{66} \\
  x_{95} & x_{96}
\end{bmatrix}_{5 \times 2}
\]

\[
\begin{bmatrix}
  x_{31} & x_{32} & x_{37} & x_{38} \\
  x_{41} & x_{42} & x_{47} & x_{48} \\
  x_{71} & x_{72} & x_{77} & x_{78} \\
  x_{81} & x_{82} & x_{87} & x_{88}
\end{bmatrix}_{4 \times 4}
\]

\[
\begin{bmatrix}
  x_{33} & x_{34} & x_{39} \\
  x_{43} & x_{44} & x_{49} \\
  x_{73} & x_{74} & x_{79} \\
  x_{83} & x_{84} & x_{89}
\end{bmatrix}_{4 \times 3}
\]

\[
\begin{bmatrix}
  x_{35} & x_{36} \\
  x_{45} & x_{46} \\
  x_{75} & x_{76} \\
  x_{85} & x_{86}
\end{bmatrix}_{4 \times 2}
\]

Processor grid =

\[
\begin{bmatrix}
  0 & 1 & 2 \\
  3 & 4 & 5
\end{bmatrix} = \begin{bmatrix}
  (0,0) & (0,1) & (0,2) \\
  (1,0) & (1,1) & (1,2)
\end{bmatrix}
\]
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

```
cov(x)
```

Parallel Code

```
cov(x)
```
ddmatrix Syntax

```r
1  cov.x <- cov(x)
2  pca <- prcomp(x)
3  x <- x[, -1]
4  col.sd <- apply(x, MARGIN=2, FUN=sd)
```
Part V

Wrapup
Wrapup
### Performance-Centered Development Model

1. Just get it working.
2. Profile vigorously.
3. Weigh your options.
   - Improve R code? (`lapply()`, vectorization, a package, \ldots)
   - Incorporate C/C++?
   - Go parallel?
   - Some combination of these\ldots
4. Don’t forget the free stuff (BLAS, bytecode compiler, \ldots).
5. Repeat 2 — 4 until performance is acceptable.
Thanks so much for attending!

Questions?

Followup session: Friday, March 6 from 1:00pm-3:00pm Eastern Time

Please go to www.xsede.org and create account if you don’t have one already.

Register for training at: https://portal.xsede.org/course-calendar/-/training-user/class/375/session/618

Password is: hpcR.