

High Performance Computing

Martin Morgan, Seth Falcon

Fred Hutchinson Cancer Research Center

20-21 May, 2010

Motivation

Challenges

- ▶ Long calculations: bootstrap, MCMC,
- ▶ Big data: genome-wide association studies, re-sequencing,
- ▶ Long \times big: . . .

Solutions

- ▶ Avoid R programming pitfalls – *very* significant benefits.
- ▶ Parallel evaluation, especially ‘embarrassingly parallel’
- ▶ Large data management

Programming pitfalls

- Pitfalls and solutions

- Measuring performance

- Case Study: GWAS

Parallel evaluation

- Embarrassingly parallel problems

- Packages and evaluation models

- Case Study: GWAS (continued)

Large data management

- Text, binary, and streaming I/O

- Data bases and netCDF

- Case Study: GWAS (concluded)

Resources

Outline

Programming pitfalls

- Pitfalls and solutions

- Measuring performance

- Case Study: GWAS

Parallel evaluation

- Embarrassingly parallel problems

- Packages and evaluation models

- Case Study: GWAS (continued)

Large data management

- Text, binary, and streaming I/O

- Data bases and netCDF

- Case Study: GWAS (concluded)

Resources

Programming pitfalls: easy solutions

- ▶ Input only required data

```
> colClasses <-  
+   c("NULL", "integer", "numeric", "NULL")  
> df <- read.table("myfile", colClasses=colClasses)
```

- ▶ Preallocate-and-fill, not copy-and-append

```
> result <- numeric(nrow(df))  
> for (i in seq_len(nrow(df)))  
+   result[[i]] <- some_calc(df[i,])
```

- ▶ Vectorized calculations, not iteration

```
> x <- runif(100000); x2 <- x^2  
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

- ▶ Avoid unnecessary character creation operations, e.g.,
USE.NAMES=FALSE in sapply, use.names=FALSE in unlist.

Programming pitfalls: moderate solutions

- ▶ Use appropriate functions, often from specialized packages.

```
> library(limma) # microarray linear models  
> fit <- lmFit(eSet, design)
```
- ▶ Identify appropriate algorithms, e.g., `%in%` is $O(N)$, whereas naive might be $O(N^2)$

```
> x <- 1:100; s <- sample(x, 10)  
> inS <- x %in% s
```
- ▶ Use C or Fortran code. Requires knowledge of other programming languages, and how to integrate these into R

Measuring performance: timing

- ▶ Use `system.time` to measure total evaluation time
 - ▶ `gcFirst=TRUE` for 'garbage collection'
- ▶ Use `replicate` to average over invocations

```
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])
```

```
[1] 0.124 0.140 0.120 0.132 0.132
```

```
> replicate(5, system.time(rowSums(m))[[1]])
```

```
[1] 0.000 0.000 0.000 0.004 0.004
```

- ▶ Cautionary tale: <http://tinyurl.com/29bd6xv>

Measuring performance: comparison

- ▶ `identical` and `all.equal` ensure that 'optimizations' produce correct results!

```
> res1 <- apply(m, 1, sum)
```

```
> res2 <- rowSums(m)
```

```
> identical(res1, res2)
```

```
[1] TRUE
```

```
> identical(c(1, -1), c(x=1, y=-1))
```

```
[1] FALSE
```

```
> all.equal(c(1, -1), c(x=1, y=-1), check.attributes=FALSE)
```

```
[1] TRUE
```


Measuring execution time: Rprof

```
> tmpf = tempfile()
> Rprof(tmpf)
> res1 <- apply(m, 1, sum)
> Rprof(NULL); summaryRprof(tmpf)
```

\$by.self

	self.time	self.pct	total.time	total.pct
"apply"	0.16	80	0.20	100
"FUN"	0.02	10	0.02	10
"lapply"	0.02	10	0.02	10
"unlist"	0.00	0	0.02	10

\$by.total

	total.time	total.pct	self.time	self.pct
"apply"	0.20	100	0.16	80
"FUN"	0.02	10	0.02	10
"lapply"	0.02	10	0.02	10
"unlist"	0.02	10	0.00	0

Measuring memory use: `tracemem`

- ▶ Enable memory profiling

```
> ~/src/R-devel/configure --help
> ~/src/R-devel/configure --enable-memory-profiling
> make -j
```

- ▶ Copy-on-change semantics

```
> x <- 1:10; tracemem(x)
[1] "<0x1b1a8f8>"
> y <- x          # no change, so no copy
> x[1] <- 2L     # x, y now differ, so copy
tracemem[0x1b1a8f8 -> 0x1b1a8a0]:
```

Measuring memory use: `tracemem`

- ▶ Copying in R functions

```
> l <- list(a=1:10, b=1:10); tracemem(l$a)
[1] "<0x1131ce0>"
> df0 <- as.data.frame(l)
tracemem[0x1131ce0 -> 0x1131bd8]: eval as.data.frame.list a
tracemem[0x1131bd8 -> 0x1131a20]: data.frame eval eval as.c
tracemem[0x1131a20 -> 0x11318c0]: as.data.frame.integer as.
> df1 <- data.frame(a=l$a, b=l$b)
tracemem[0x1131ce0 -> 0x11332c0]: data.frame
tracemem[0x11332c0 -> 0x1133160]: as.data.frame.integer as.
> identical(df0, df1)
[1] TRUE
```

Case study: GWAS

- ▶ Subset of genome-wide association study data

```
> fname1 <- system.file("extdata", "gwas_2.rda",  
+                        package="AdvancedR")  
> load(fname1)  
> gwas[1:2, 1:8]
```

	CaseControl	Sex	Age	X1	X2	X3	X4	X5
id_1	Case	M	40	AA	AB	AA	AB	AA
id_2	Case	F	33	AA	AA	AA	AA	AA

GWAS and glm

- ▶ Interested in fitting generalized linear model to each SNP

```
> snp0 <- function(i, gwas) {  
+   snp <- gwas[[i+3L]]  
+   glm(CaseControl ~ Age + Sex + snp,  
+       family=binomial, data=gwas)$coef  
+ }  
> system.time(sapply(1:10, snp0, gwas))  
  
   user  system elapsed  
4.905   0.036   4.946
```

GWAS case study: further directions

glm can be optimized for SNPs

- ▶ Build the design matrix for $\text{CaseControl} \sim \text{Age} + \text{Sex}$ once, rather than once per SNP
- ▶ Use the estimate without the SNP as a starting point
- ▶ `snpMatrix` fits GLMs very efficiently

Outcome

- ▶ ~ 1000 SNPs per second

Important lessons

- ▶ Careful optimization can often greatly reduce evaluation time
- ▶ Others may likely have done the work for you!

Outline

Programming pitfalls

- Pitfalls and solutions

- Measuring performance

- Case Study: GWAS

Parallel evaluation

- Embarrassingly parallel problems

- Packages and evaluation models

- Case Study: GWAS (continued)

Large data management

- Text, binary, and streaming I/O

- Data bases and netCDF

- Case Study: GWAS (concluded)

Resources

'Embarrassingly parallel' problems

Problems that are:

- ▶ Easily divisible into different, more-or-less identical, independent *tasks*
- ▶ Tasks distributed across distinct computational *nodes*.
- ▶ Examples: bootstrap; MCMC; row- or column-wise matrix operations; 'batch' processing of multiple files, ...

What to expect: ideal performance

- ▶ Execution time inversely proportional to number of available nodes: $10\times$ speed-up requires 10 nodes, $100\times$ speedup requires 100 nodes
- ▶ Communication (data transfer between nodes) is expensive
- ▶ 'Coarse-grained' tasks work best

Packages and other solutions

Package	Hardware	Challenges
multicore	Computer	Not Windows (doSMP soon)
Rmpi	Cluster	Additional job management software (e.g., slurm)
snow	Cluster	Light-weight; convenient if MPI not available
BLAS, pnmath	Computer	Customize R build; benefits math routines only

Parallel interfaces

- ▶ Package-specific, e.g., `mpi.parLapply`
- ▶ `foreach`, `iterators`, `doMC`, ...: common interface; fault tolerance; alternative programming model

General guidelines for parallel computing

- ▶ Maximize computation per job
- ▶ Distribute data implicitly, e.g., using shared file systems
- ▶ Nodes transform large data to small summary
 - ▶ E.g.: ShortRead quality assessment.
- ▶ Construct self-contained functions that avoid global variables.
- ▶ Random numbers need special care!

multicore

- ▶ Shared memory, i.e., one computer with several cores

```
> system.time(lapply(1:10, snp0, gwas))
  user  system elapsed
1.672   0.016   1.687
> library(multicore)
> system.time(mclapply(1:10, snp0, gwas))
  user  system elapsed
1.864   0.348   1.119
```

multicore: under the hood

- ▶ Operating system `fork`: new process, initially identical to current, OS-level copy-on-change.
- ▶ `parallel`: spawns new process, returns process id, starts expression evaluation.
- ▶ `collect`: queries process id to retrieve result, terminates process.
- ▶ `mclapply`: orchestrates `parallel` / `collect`

foreach

- ▶ `foreach`: establishes a for-like iterator
- ▶ `%dopar%`: infix binary function; left-hand-side: `foreach`; right-hand-side: expression for evaluation
- ▶ Variety of parallel back-ends, e.g., `doMC` for multicore; register with `registerDoMC`

```
> library(foreach)
> if ("windows" != .Platform$OS.type) {
+   library(doMC); registerDoMC()
+   res <- foreach(i=1:10) %dopar% snp0(i, gwas)
+ }
```

iterators and foreach

iterators package

- ▶ `iter`: create an iterator on an object
- ▶ `nextElem`: return the next element of the object
- ▶ Built-in (e.g, `iapply`, `isplit`) and customizable

```
> snp1 <- function(snp, gwas) {  
+   glm(CaseControl ~ Age + Sex + snp,  
+       family=binomial, data=gwas)$coef  
+ }  
> snps <- gwas[,11:20]  
> res <- foreach(it=iter(snps, "column")) %dopar%  
+   snp1(it, gwas)
```

Rmpi on a cluster

- ▶ 'Message passing' interface (MPI)

Players

- ▶ `slurm`: allocate resources, e.g., `salloc -N 4` allocates 4 nodes for computation
- ▶ `mpi`: e.g., `mpirun -n 1` starts a program on one node
- ▶ R and the Rmpi package

Interactive Rmpi: manager / worker

```
hyrax1:~> salloc -N 4 mpirun -n 1 R --interactive --quiet
salloc: Granted job allocation 239631
> library(Rmpi)
> mpi.spawn.Rslaves()
[...SNIP...]
> mpi.parSapply(1:10,
                function(i) c(i=i, rank=mpi.comm.rank()))
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
i      1   2   3   4   5   6   7   8   9   10
rank   1   1   1   2   2   3   3   4   4   4
> mpi.quit()
salloc: Relinquishing job allocation 239631
```


Manage / worker

- ▶ 'Manager' script that spawns workers, tells workers what to do, collates results
- ▶ Submit as 'batch' job on a single R node
- ▶ View example script with `readScript("spawn.R")`

```
hyrax1:~> salloc -N 4 mpirun -n 1 \  
R CMD BATCH /path/to/spawn.R
```

Single instruction, multiple data (SIMD)

- ▶ Single script, evaluated on each node, `readScript("simd.R")`.
- ▶ Script specializes data for specific node
- ▶ After evaluation, script specializes so that one node collates results from others

```
hyrax1:~> salloc -N 4 mpirun -n 4 \  
  R CMD BATCH --slave /path/to/simd.R
```

Case study: GWAS (continued)

- ▶ Readily parallelized – glm for each SNP fit independently
- ▶ Divide SNPs into equal sized groups, one group per node
- ▶ SIMD evaluation model
- ▶ Need to manage data – appropriate SNPs and metadata to each node

Important lessons

- ▶ Parallel evaluation for real problems can be difficult
- ▶ Parallelization after optimization
- ▶ Optimize / parallelize only after confirming that no one else has already done the work!

Outline

Programming pitfalls

- Pitfalls and solutions

- Measuring performance

- Case Study: GWAS

Parallel evaluation

- Embarrassingly parallel problems

- Packages and evaluation models

- Case Study: GWAS (continued)

Large data management

- Text, binary, and streaming I/O

- Data bases and netCDF

- Case Study: GWAS (concluded)

Resources

Large data management

Putting appropriate data in memory

- ▶ An R analysis can make multiple copies of each data set
- ▶ Limits performance (I/O, but also calculations)
- ▶ Wastes system resources (e.g., decreasing the number of parallel tasks that can be executed)

Solutions

- ▶ Text versus R binary files
- ▶ 'Stream' processing
- ▶ Data base use
- ▶ High-performance numeric storage

Text versus R binary files

- ▶ Text is slower than compressed binary
- ▶ Compressed binary is slower than binary

```
> ftmp <- tempfile()
> write.csv(gwas, ftmp)
> system.time(read.csv(ftmp, row.names=1))[[3]]

[1] 10.265

> save(gwas, file=ftmp)
> replicate(5, system.time(load(ftmp, new.env()))[[3]])

[1] 0.903 0.899 0.902 0.902 0.901

> save(gwas, file=ftmp, compress=FALSE)
> replicate(5, system.time(load(ftmp, new.env()))[[3]])

[1] 0.643 0.672 0.644 0.653 0.649

> unlink(ftmp)
```

'Stream' processing

- ▶ Read in a chunk, process, read in next chunk
- ▶ Use 'connections' to keep file open between chunks
- ▶ Good for very large data sets (if necessary)
- ▶ A few packages, e.g., biglm, exploit this model
- ▶ See `readScript("fapply.R")`

Data bases

SQL

- ▶ Represent data in a SQL data base
- ▶ Best for *relational* (structured) data of moderate (e.g., millions of rows) size
- ▶ Not the best solution for, e.g., array-like numerical data

Use

- ▶ DBI package provides abstract interface
- ▶ RSQLite (built-in to R), RMySQL, RPostgreSQL, ... provide implementations

Example: RSQLite set-up

```
> db0 <- tempfile()
> library(RSQLite)
> drv <- dbDriver("SQLite")
> conn <- dbConnect(drv, dbname=db0)
```


GWAS metadata

Create

```
> gwasPhenotypes <- gwas[,1:3]
> dbWriteTable(conn, "gwasPhenotypes", gwasPhenotypes)
```

```
[1] TRUE
```

Retrieve

```
> q <- dbSendQuery(conn, "SELECT * FROM gwasPhenotypes")
> fetch(q, n = 2) # first 2; n = -1 for all
```

	row_names	CaseControl	Sex	Age
1	id_1	Case	M	40
2	id_2	Case	F	33

```
> invisible(dbClearResult(q)) # close out query
```

Clean-up

```
> invisible(dbDisconnect(conn))
```

NetCDF and the ncdf package

NetCDF and ncdf

- ▶ Network Common Data Form: array-oriented scientific data
- ▶ ncdf: R package for NetCDF access
 - ▶ Warning: character arrays very inefficient in ncdf
- ▶ ncdf4: recent; NetCDF 4 format; not yet available for Windows

Data and library

```
> ngwas <- local({  
+   x0 <- lapply(gwas[, -(1:3)], as.integer)  
+   matrix(unlist(x0, use.names=FALSE), ncol=length(x0))  
+ })  
> ncdf0 <- tempfile()  
> library(ncdf)
```

ncdf, continued

- ▶ Define dimensions and variable

```
> sampd <- dim.def.ncdf("Sample", "id", seq_len(nrow(ngwas)))
> snpd <- dim.def.ncdf("SNP", "id", seq_len(ncol(ngwas)))
> snpv <- var.def.ncdf("Genotype",
+                       units="1: AA, 2: AB; 3: BB",
+                       dim=list(sampd, snpd),
+                       missval=-1L, prec="integer")
```

- ▶ Create file

```
> nc <- create.ncdf(ncdf0, snpv)
> put.var.ncdf(nc, snpv, ngwas)
> invisible(close(nc))
```

ncdf, continued

- ▶ Very favorable file I/O performance

```
> nc <- open.ncdf(ncdf0)
> system.time({
+   nc_gwas <- get.var.ncdf(nc, "Genotype")
+ })[[1]]

[1] 0.164
```

- ▶ Easy to obtain data slices

```
> g <- get.var.ncdf(nc, "Genotype", start=c(30, 100),
+   count=c(10, 20)) # samples 30:40, snps 100:120
> g <- get.var.ncdf(nc, "Genotype", start=c(1,1000),
+   count=c(-1, 100)) # all samples, snps 1000:1100
> invisible(close(nc))
```

Case study: GWAS (concluded)

Overall solution

- ▶ Optimize `glm` for SNPs
- ▶ Store SNP data as netCDF, metadata as SQL
- ▶ Use SIMD model to parallelize calculations

Outcome

- ▶ Initially: < 10 SNPs per second
- ▶ Optimized: ~ 1000 SNP per second
- ▶ 100 node cluster: $\sim 100,000$ SNP per second

Outline

Programming pitfalls

- Pitfalls and solutions

- Measuring performance

- Case Study: GWAS

Parallel evaluation

- Embarrassingly parallel problems

- Packages and evaluation models

- Case Study: GWAS (continued)

Large data management

- Text, binary, and streaming I/O

- Data bases and netCDF

- Case Study: GWAS (concluded)

Resources

Resources

- ▶ News group:
`https://stat.ethz.ch/mailman/listinfo/r-sig-hpc`
- ▶ CRAN Task View: `http://cran.fhcrc.org/web/views/HighPerformanceComputing.html`
- ▶ Key packages: multicore, Rmpi, snow, foreach (and friends); RSQLite, ncdf