Acknowledgements (for yesterday’s ‘New Bioconductor Developments’) 

Hervé Pagès  *Biostrings*, sequence infrastructure.

Marc Carlson  Annotations, *GenomicFeatures*, sequence infrastructure.

Nishant Gopalakrishnan  New packages, flow cytometry, *graph*, sequence infrastructure.

Chao-Jen Wong  Flow cytometry, microarrays.

Dan Tenenbaum  Web site, build system.

Valerie Obenchain  Sequence infrastructure and analysis.
Efficient R Programming

Martin Morgan

Fred Hutchinson Cancer Research Center
mtmorgan@fhcrc.org

17-18 November, 2010
Efficient R Programming

- Common programming pitfalls and their solutions
- R tools for measuring performance
- Parallelization to increase throughput.
- Managing data.
Programming pitfalls: easy solutions

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

- Preallocate-and-fill, not copy-and-append
  ```r
  > result <- numeric(nrow(df))
  > for (i in seq_len(nrow(df)))
  +    result[[i]] <- some_calc(df[i,])
  ```

- Vectorized calculations, not iteration
  ```r
  > x <- runif(100000); x2 <- x^2
  > m <- matrix(x2, nrow=1000); y <- rowSums(m)
  ```

- Avoid unnecessary character-based operations, e.g.,
  `USE.NAMES=FALSE` in `sapply`, `use.names=FALSE` in `unlist`. 
Programming pitfalls: moderate solutions

- Use appropriate functions, often from specialized packages.
  
  ```r
  > library(limma) # microarray linear models
  > fit <- lmFit(eSet, design)
  ```

- Identify appropriate algorithms. Polynomial:
  ```r
  > x <- 1:100; s <- sample(x, 10)
  > inS1 <- logical(length(x))
  > for (i in x) {
  +   for (j in s)
  +     if (i == j) inS1[j] <- TRUE
  + }
  > inS2 <- x %in% s
  ```

- Use C or other code. Requires knowledge of other programming languages, and how to integrate these in to R
Measuring performance: timing

Use `system.time` to measure total evaluation time;

- Use `replicate` to average over invocations

```r
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])
> replicate(5, system.time(rowSums(m))[[1]])
```
Measuring performance: comparison

identical and all.equal ensure that ‘optimizations’ are producing correct results!

> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)
> identical(c(1, -1), c(x=1, y=-1))
> all.equal(c(1, -1), c(x=1, y=-1), check.attributes=FALSE)
Measuring performance: profiling with `Rprof`

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

$by.self

<table>
<thead>
<tr>
<th></th>
<th>self.time</th>
<th>self.pct</th>
<th>total.time</th>
<th>total.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;apply&quot;</td>
<td>0.16</td>
<td>80</td>
<td>0.20</td>
<td>100</td>
</tr>
<tr>
<td>&quot;FUN&quot;</td>
<td>0.02</td>
<td>10</td>
<td>0.02</td>
<td>10</td>
</tr>
<tr>
<td>&quot;lapply&quot;</td>
<td>0.02</td>
<td>10</td>
<td>0.02</td>
<td>10</td>
</tr>
<tr>
<td>&quot;unlist&quot;</td>
<td>0.00</td>
<td>0</td>
<td>0.02</td>
<td>10</td>
</tr>
</tbody>
</table>

$by.total

<table>
<thead>
<tr>
<th></th>
<th>total.time</th>
<th>total.pct</th>
<th>self.time</th>
<th>self.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;apply&quot;</td>
<td>0.20</td>
<td>100</td>
<td>0.16</td>
<td>80</td>
</tr>
<tr>
<td>&quot;FUN&quot;</td>
<td>0.02</td>
<td>10</td>
<td>0.02</td>
<td>10</td>
</tr>
<tr>
<td>&quot;lapply&quot;</td>
<td>0.02</td>
<td>10</td>
<td>0.02</td>
<td>10</td>
</tr>
<tr>
<td>&quot;unlist&quot;</td>
<td>0.02</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Using Multiple CPUs

Modern computers have multiple processors, each with multiple cores.

▶ Strategy: develop efficient single-processor code first, then parallelize at a ‘coarse’ level
▶ Often requires efficient data input and memory management.
▶ Approaches: high performance numerical algorithms; multiple processors on a single computer; clusters; specialized (e.g., GPU).

Configure R with parallel BLAS for numerically intensive operations.

▶ Benefit for large, matrix-oriented calculations only.
Using Multiple CPUs II

Use *multicore* and other single-computer solutions.

- ‘Shared-memory’ copy-on-change semantics, so memory efficient.

- Easy to use.
  ```
  > library(multicore)
  > test <- function(FUN)
  + system.time(FUN(1:4, function(i) Sys.sleep(1)))
  > test(lapply)    # 4 seconds
  > test(mclapply)  # 1 second
  ```

- Not available on all platforms; care required for package use.

- *foreach* & friends provide alternative interface.

- Files (e.g., SQL, ncdf) can be tricky – open inside *FUN*. 
Using Multiple CPUs III

Use *Rmpi* and other cluster-based solutions.

- Easy to use, in principle.
  
  ```
  > library(Rmpi)
  > mpi.spawn.Rslaves(nslaves=4)
  > test(mpi.parLapply) # 1 second
  ```

- Real-world use requires mastering cluster and job-management software, e.g., *slurm*, *SGE*.

- Entire *R* session for each instance – memory management very important.

- *Communication costs* (moving data between workers) need to be managed.
Managing Data

Selectively input data.
- `colClasses`, `skip`, `nrows` and similar arguments to `scan`, `read.table`, etc.
- ‘Stream’ across large files.

Use R packages that represent big data on disk.
- `ff`, `bigmemory`
- Requires specialized approaches to manage data and for common analyses.

Query a data base to retrieve relevant data.
- `RSQLite` for easy, self-contained moderate access.

Use high-performance data formats.
- Domain specific, e.g., BAM and `Rsamtools`.
- General purpose, e.g., NetCDF and `ncdf4`.
Case study

Fitting GLM to GWAS SNPs

- 500000 snps, 2000 individuals
- \( y \sim \text{age} + \text{gender} + \text{snp}[,i] \)

Iterations

1. \texttt{glm}: 10’s of snp / second.
2. \texttt{glm.fit}, common model matrix, smart start, \ldots: 1000 snp / second.
3. \texttt{Rmpi, ncdf}: complete analysis in 8s.

A better way?

- \texttt{snpMatrix2}
Resources