Performance and Parallel Evaluation

Martin Morgan (mtmorgan@fredhutch.org)
Fred Hutchinson Cancer Research Center
Seattle, WA, USA

18 June, 2015

Performance & Parallel Evaluation

R code

- Correct, then efficient

Parallel evaluation

- Computer
- Cluster
- Cloud

R code

Priorities

1. Correct!
2. Robust – works for most realistic inputs
3. Simple
4. Fast

R code: deadly sins

1. Unnecessary iteration

```
x <- 1:10; for (i in seq_along(x)) x[i] = log(x[i])
```

2. Copy-and-append iteration

```
answer <- numeric()
for (i in 1:10) answer <- c(answer, 1/i)
for (i in 1:10) answer[i] <- 1/i
```

3. Unneccessary evaluation

```
x <- 1:10
for (i in seq_along(x)) x[i] = x[i] * sqrt(2)
```

4. Re-implementation

R code: saving graces I

```
fun1 <- function(n) {
    ## How many sins?
    x <- numeric()
    for (i in 1:n)
        x <- c(x, log(i) * sqrt(2))
    x
}

fun2 <- function(n)
    log(seq_len(n)) * sqrt(2)
```

R code: saving graces II

1. Validation – `identical()`, `all.equal()`

```
identical(fun1(1000), fun2(1000))
```

```
## [1] TRUE
```

2. Timing – `system.time()`, *microbenchmark()*

```
library(microbenchmark)
microbenchmark(fun1(1000), fun2(1000))
```

```
## Unit: microseconds
##         expr      min         lq       mean
##  fun1(1000) 1726.480 1746.0545 2518.5444
##  fun2(1000)   36.131   36.8945   41.0648
##      median        uq      max neval
##  1771.0925 2556.803 5122.918   100
##    40.6885   43.264   63.635   100
```

R code: saving graces III

3. 'Experience' – available packages & functions
4. Profiling – `Rprof()`
5. Foreign languages – e.g., C, *Rcpp*

Parallel evaluation

- Most often: 'embarassingly parallel' evaluation of iterative `for` loops / `lapply()`

Other packages

- *parallel* – a base package; single computer
- *foreach* – popular 'for' loop paradigm
- *BatchJobs* – clusters with job schedulers
- *Rmpi* – classic HPC

BiocParallel

- Consistent interface
- Plays well with many *Bioconductor* packages

Parallel evaluation

```
library(BiocParallel)
fun <- function(i) {
    Sys.sleep(1)
    i
}
system.time(res1 <- lapply(1:5, fun))

##    user  system elapsed
##   0.002   0.000   5.007

system.time(res2 <- bplapply(1:5, fun))

##    user  system elapsed
##   0.035   0.024   1.085

identical(res1, res2)

## [1] TRUE
```

Parallel evaluation: *BiocParallel*

- Different `*Param()` objects for styles of computing, e.g.,
  - `SerialParam()`: no parallel evaluation
  - `MulticoreParam()`: separate forked processes on one computer
  - `BatchJobsParam()`: jobs submitted to a cluster queuing system
- `register()` a param or provide it as an argument for use in `bplapply()`.
- Sensible default values.

Parallel evaluation: processing large genomic files

Restrict input to minimum necessary data

- Select columns or fields of files to import, e.g., `colClasses` argument to `read.table()`; `ScanBamParam()` and `ScanVcfParam()`.
- Use a data base, hdf5, or other file format that allows queries or slices of the data to be imported.

Iterater through files to manage memory use

- File connections in base R
- `BamFile("my.bam", yieldSize=1000000)`

GenomicFiles

- Functions to help manage collections of genomic files

Parallel evaluation: extended example

Goal: for a vector of paths to bam files, `fls`, summarize GC content of each aligned read.

```
library(Rsamtools); library(GenomicFiles)
bfls <- BamFileList(fls, yieldSize=100000)
yield <- function(bfl) # input a chunk of alignments
    readGAlignments(bfl, param=ScanBamParam(what="seq"))
map <- function(aln) { # GC content, bin & cummulate
    gc <- letterFrequency(mcols(aln)$seq, "GC",
        as.prob=TRUE)
    cumsum(tabulate(1 + gc * 50, 51))
}
reduce <- `+`

gc <- bplapply(bfls, reduceByYield, yield, map, reduce)
gc <- simplify2array(gc)
```