Performance and Parallel Evaluation

Martin Morgan (martin.morgan@roswellpark.org)
Roswell Park Cancer Institute
Buffalo, NY, USA

15 July, 2016

Performance & Parallel Evaluation

My code is slow, how do I make it run faster?

Write better *R* code

- ▶ Correct, then efficient
- ▶ 10-1000× speed-up, great satisfaction

Parallel evaluation

- ▶ Computer: 5-10× speed-up, 2-5× frustration
- ▶ Cluster: 10-100× speed-up, 10-20× frustration
- ▶ Cloud: 100+× speed-up, 20-50× frustration

R code

Priorities

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

R code: deadly sins

1. Unnecessary iteration

```
x <- 1:10000; for (i in seq_along(x)) x[i] = log(x[i])
```

2. Copy-and-append iteration

```
answer <- numeric()
for (i in 1:10000) answer <- c(answer, 1/i)
for (i in 1:10000) answer[i] <- 1/i
```

3. Unneccessary evaluation

```
x <- 1:1000000
for (i in seq_along(x)) x[i] = x[i] * sqrt(2)
```

4. Re-implementation

R code: saving graces I

```r
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     median
##  fun1(1000)   1573.650  1587.4655  2406.3808   1628.533
##  fun2(1000)     21.088    21.8585    23.3476     22.882
##          uq        max neval
##   1654.2585  17768.803   100
##     23.8545     40.376   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
##   0.000   0.000   5.006

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

##    user  system elapsed
##   0.024   0.028   3.487

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: errors and debugging

- `bptry()` to see errors.
- `BPREDO` argument to `bplapply()` to evaluate just the errors.
- `BPPARAM=SerialParam()` to make problematic code run locally for easy debugging.
- See the vignette Errors, Logs, and Debugging

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.

Iterate 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.

```r
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)
```

Summary

- **Correct** first, performance second
- No need to worry about code that doesn't take very long!
- 'Embarassingly' parallel (`lapply()`-like) problems easily parallelized, especially on a single computer.
- Opportunity for very scalable computations, e.g., via AMI & StarCluster.

Acknowledgments

- Core: Valerie Obenchain, Hervé Pagès, (Dan Tenenbaum), Lori Shepherd, Marcel Ramos, Yubo Cheng.

https://bioconductor.org,
https://support.bioconductor.org