R profiling

Stat 580: Statistical Computing

- Theme: Black - White
- Printable version

References

- "Advanced R", by Hadley Wickham.
- "Writing R Extensions", by R Core Team.

Performance

- R is not a fast language
  - easy for you (to do data analysis and statistics)
  - not easy for the computer
  - improve speed of your code:
    - profiling
    - R's C interface
    - Rcpp

Why is R slow?

- R as both language and implementation of that language
  - R-language: defines what R code means and how it should work
  - implementation: reads R code and computes a result.
    - the most popular implementation is the one from r-project.org. (GNU-R)
- Both language and implementation (GNU-R) limitations can be the reasons of slowness.
  - We will look into a few reasons from the language performance perspectives.
- Poorly written R code

Microbenchmarking

- For investigation of (language and implementation) performance, we need measurement of performance of a very small piece of code
  - take microseconds or naoseconds
  - for deeper understanding
  - not very practical for optimizing your code

Microbenchmarking

```
library(microbenchmark)

x <- runif(100)
microbenchmark(sqrt(x), x^0.5)
```

- `microbenchmark()`
  - more accurate replacement of `system.time(replicate(1000, expr))` expression
  - randomized ordering (the default)
  - warm-up iterations performed before the actual benchmark

Example

```
Unit: nanoseconds
    expr  min     lq    mean median     uq   max neval
 sqrt(x)  839  890.0 1865.58  979.5 2666.5 35456   100
   x^0.5 4837 4956.5 5497.08 5070.5 5176.5 14552   100
```

- `sqrt()` is a lot faster! Why?
- Should I replace every occurrence of `x^0.5` by `sqrt(x)`?
  - What is the unit?
  - set argument `unit=eps`:

```
Unit: evaluations per second
    expr       min        lq      mean    median        uq       max neval
 sqrt(x) 217627.86 1281293.6 1272596.7 1456671.2 1579778.8 1683501.7   100
   x^0.5  71797.82  253132.6  245315.7  258431.5  265287.3  270270.3   100
```

Extreme dynamism

- R is extremely dynamic, you can modify almost anything
  - after it is created
  - outside of the local environment (e.g., <<–)
  - to completely different object (atomic vector to function)
    - `x <- c(1,2); x <- function(){return(c(3,4))}`
  - more...

Example

```
x <- 0L
for (i in 1:1e6) {
  x <- x + 1
}
```

- R doesn't know that x is always an integer.
- R has to look for the right + method in every iteration.

Extreme dynamism

- Both good and bad:
  - Good (to us): flexibility, minimal upfront planning
    - Recall: in C, we have to plan ahead by declaring the variables as the right types.
  - Bad (to the computer): difficulty to predict exactly what will happen with a given function call
    - difficult for an interpreter or compiler to make an optimization
    - if an interpreter can't predict what's going to happen, it has to consider many options.

Name lookup with mutable environments

- Scoping rules allow the same name to be used for different objects.
- R uses lexical (static) scoping (vs. dynamic scoping):
  - if an "unknown" variable is used in a function, its value are searched for in the environment in which the function was defined.

```
a = 1
f <- function(){
  return(a)
}

g <- function(){
  a = 2
  f()
}

g()
```

Example

```
a <- 1
f <- function() {
  g <- function() {
    print(a)
    assign("a", 2, envir = parent.frame())
    print(a)
    a <- 3
    print(a)
  }
  print(a)
  g()
  print(a)
}
f()
```

Name lookup with mutable environments

- Name lookup has to be done each time (partly due to extreme dynamism)
- Even worse, almost every operation is a lexically scoped function call.
  - e.g. +, –
  - e.g. (, {
  - Since these functions are in global environment, R has to look through every environment in the search path.
  - R asks: Who knows someone wouldn't change + to –?
    - extreme dynamism: R has to worry that these functions are defined (in environments) on the search path

```
f <- function(x){
  (2 * x) ^ 2
}
```

Example

```
random_env <- function(parent = globalenv()) {
  letter_list <- setNames(as.list(runif(26)), LETTERS)
  list2env(letter_list, envir = new.env(parent = parent))
}
set_env <- function(f, e) {
  environment(f) <- e
  f
}
f2 <- set_env(f, random_env())
f3 <- set_env(f, random_env(environment(f2)))
f4 <- set_env(f, random_env(environment(f3)))

microbenchmark(
  f(1),
  f2(1),
  f3(1),
  f4(1),
  times = 10000
)
```

Lazy evaluation overhead

R uses lazy evaluation:

- function arguments are evaluated lazily - evaluated if they're actually used.

```
f <- function(x) {
1+1
}
f(cat("hello!\n"))
```

R uses promise object to contain the expression and environment needed to compute the result.

- overhead of creating such objects
- recall that gcc compiler (with -Wall) will warn you if you have unused variables

Example

```
f0 <- function() NULL
f1 <- function(a = 1) NULL
f2 <- function(a = 1, b = 1) NULL
f3 <- function(a = 1, b = 2, c = 3) NULL
f4 <- function(a = 1, b = 2, c = 4, d = 4) NULL
f5 <- function(a = 1, b = 2, c = 4, d = 4, e = 5) NULL
microbenchmark(f0(), f1(), f2(), f3(), f4(), f5(), times = 10000)
```

Profiling your code

- Before you get your hands dirty to optimize your code, think about:
  - which part of the code is the bottleneck
  - Come up with solutions to optimize
    - which level: R (e.g., vectorization), lower level (e.g. C, Fortran)?
    - algorithm (approximations are sometimes allowed.)
  - Time investment vs. efficiency gain
    - choose a solution
    - do nothing

Profiling for speed

- To detect which part of the code is the bottleneck, we use profiler.
- R uses sampling profiler:
  - it checks which function is being used at fixed time intervals (e.g. every 20 msecs)
  - stochastic
  - variability in results due to the intervals between sampling
  - `Rprof()` is the profiler that comes with R
- Limitation: R profiling does not extend to C code, primitive functions or byte code compiled code.

Example

```
library(lineprof) # for the use of pause

f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}

tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
summaryRprof(tmp)
unlink(tmp)
```

Using lineprof package

```
install.packages("devtools")
devtools::install_github("hadley/lineprof")
```

- the fundamental unit of analysis in lineprof() is a line of code
- note that a line of code can contain multiple function calls
- but it's easier to understand the context

```
l <- lineprof(f())

library(shiny)
shine(l)
```

Improving speed

Simple strategies to improve speed of R code without resorting to lower level implementation:

- Vectorization
- Avoid growing an object
- Byte code compilation
- Parallelization

Vectorization

- use vector (matrix, array) rather than scalar as "your working object"
  - to make use of the vectorized function
  - the loops in a vectorized function are written in low-level language instead of R
  - you should find the existing R function that is implemented in low-level language and most closely applies to your problem.
- replace loops by vector/matrix operations (find a matrix algebra version)
- use `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()` instead of `apply` (if possible)
- vectorized subsetting (e.g. `x[is.na(x)] <- 0`)
- Other useful vectorized functions: `cut()`, `findInverval()`, `cumsum()`, `diff()`

Avoid growing an object

- Functions like `c()`, `append()`, `cbind()`, `rbind()` or `paste()` can be used to build a bigger object on top of an old object.
  - R must first allocate space for the new object and then copy the old one to the new location.
  - hurts the performance, especially using in a loop.

```
x <- NULL
for (i in (1:10)){
x <- c(x, i)
}
```

Example

```
random_string <- function() {
  paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings100 <- replicate(100, random_string())

collapse <- function(xs) {
  out <- ""
  for (x in xs) {
    out <- paste0(out, x)
  }
  out
}

microbenchmark(
  collapse(strings100),
  paste(strings100, collapse = "")
)
```

Bytecode compilation

- You can use bytecode compiler to improve the speed of the R code.
  - compile your R code into bytecode (more optimized because it is closer to the machine instructions)
  - fast and easy
  - considerable performance gain (usually 5-10%) (for the amount of effort you spend)

Example

```
# old R version of lapply
la1 <- function(X, FUN, ...) {
    FUN <- match.fun(FUN)
    if (!is.list(X))
    X <- as.list(X)
    rval <- vector("list", length(X))
    for(i in seq(along = X))
    rval[i] <- list(FUN(X[[i]], ...))
    names(rval) <- names(X)          # keep `names' !
    return(rval)
}

la1c <- compiler::cmpfun(la1)

x <- list(1:10, letters, c(F, T), NULL)
microbenchmark(
  la1(x, is.null),
  la1c(x, is.null),
  lapply(x, is.null)
)
```

Parallelization via foreach package

- It is easy to implement a type of parallel computing called "embarassingly parallel" in R
  - can be divided into small problems that can be solved independently
  - does not require exchange of information between workers
- You can use foreach package to implement a for loop using parallel computing
  - requires a "parallel backend" to "define the workers"
  - doParallel package provides the parallel backend

Example

```
library(doParallel)

cl <- makeCluster(3)
registerDoParallel(cl)
foreach(i=1:10) %dopar% {
  max(svd(matrix(rnorm(100000), nr=100))$d)
}
stopCluster(cl)
```