**SOFTWARE OPTIMIZATION**

Getting software to run in a reasonable amount of time is always a key consideration when working with large datasets. A mathematical understanding of software algorithms is usually a key component of computer science and software engineering curricula, but not widely covered in other disciplines. Additionally, while software engineering texts and curricula highlight the importance of testing for non-functional requirements such as performance (Sommerville, 2015), they often fail to provide details on how best to evaluate software performance or how to plan for performance during the various phases of software lifecycle.

The survey of R packages that follows indicates that approximately 75% of packages do not use optimization related packages nor compiled code to improve performance. Following the analysis of R packages, this section will provide a starting point for additional study, research, and experimentation. The key aspects of software optimization discussed in this are: identify a performance target, understanding and applying Big O notation, and the use code profiling and benchmarking tools.

**IDENTIFY AND VALIDATE PERFORMANCE TARGET**

The first step to software optimization is to understand the functional and non-functional requirements of the software being built. Based on expected input, output, and platform the software will be run on, one can make a decision as to what is good enough for the software being developed. A pragmatic approach is best – do not spend time optimizing if it does not add value.

The Python Foundation’s Python language wiki provides excellent high-level advice (Python Wiki Contributors, 2018): First get the software working correctly, test to see if it is correct, profile the application if it is slow, and lastly optimize based on the results of code profiling. If necessary, repeat multiple cycles of testing, profiling, and optimization phases.

Once the functional requirements have been correctly implemented and validated, a decision point is reached: decide if the software is slow and in need of evaluation and optimization. While this may seem a trivial and unnecessary step, it should not be overlooked; a careful evaluation of costs of vs benefit from an optimization effort should be evaluated before moving forward. Some methods for gathering the performance target are through an evaluation of other similar software, interdependencies of the software and its interaction with other systems, and discussion with other experts in the field.

Once a performance target has been identified, development of acceptance tests for performance can begin. While performance testing is often considered an anti-pattern of testing (Moilanen, 2014) some repeatable tests should be created to track performance as development progresses. Often a ‘stress test’ or a test with greater than expected input/usage is the best way to do this. A good target is to check an order of magnitude larger input than expected. This type of testing can provide valuable insight into the performance characteristics of the software as well unearth potentials for failure due to unexpected load (Sommerville, 2015).

Here is an example of performance validation testing that can also serve as a basic reproducibility test calling the main function from PCCC using the microbenchmark package (one could also use bench, benchr, or other available R packages).

library(pccc)

rm(list=ls())

gc()

icd10\_large <- feather::read\_feather("../icd\_file\_generator/icd10\_sample\_large.feather")

library(microbenchmark)

microbenchmark(

ccc(icd10\_large[1:10000, c(1:45)], # get id, dx, and pc columns

id = id,

dx\_cols = dplyr::starts\_with("dx"),

pc\_cols = dplyr::starts\_with("pc"),

icdv = 10),

times = 10)

Unit: seconds

expr min lq mean median uq max neval

ccc 2.857625 2.908964 2.959805 2.920408 3.023602 3.119937 10

Results are from a system with 3.1 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3, PCI-Express SSD, running macOS 10.12.6 and R version 3.5.1 (2018-07-02).

As runs of a software can differ significantly from one run to the next due to other software running on the test system, a good starting point is to run the same test 10 times (rather than the microbenchmark default of 100 due to this being a longer running process) and record the reported mean run time value. microbenchmark also shows median, lower and upper quartiles, min, and max run times. The actual ccc() call specifics are un-important; the key is to test the main features of your software in a repeatable fashion and watch for performance changes over time. These metrics can help to identify if a test was valid and indicate a need for retesting; i.e. a large interquartile range may indicate not enough tests were run or some aspect of environment is causing performance variations. Software benchmarking is highly system specific in that changing OS version, R version, R dependent package version, compiler version (if compiled code involved), or hardware will change the results. As long as all tests are run the same on the same system with the same software, one can compare timings as development progresses.

Lastly, although the example above is focused on runtime, it can be beneficial to also identify targets for disk space used and memory required to complete all desired tasks. As an example, tools such as bench and profvis demonstrated in our ‘Code Profiling/Benchmarking’ section as well as object.size() from core R can give developers insight into memory allocation and usage. There are many resources beyond this work that can provide guidance on how to minimize RAM and disk resources (Kane, Emerson & Weston, 2013; Wickham, 2014; Wickham et al., 2016; Klik, Collet & Facebook, 2018).

**BIG O NOTATION**

Big O notation is a method for mathematically determining the upper bound on performance of a block of code without consideration for language and hardware specifics. Although performance can be evaluated in terms of storage or run time, most examples and comparisons focus on run time. However, when working with large datasets, memory usage and disk usage can be of equal or higher importance than run. Big O notation is reported in terms of input (usually denoted as n) and allows one to quickly compare theoretical performance of different algorithms.

The basic steps for evaluating the upper bound of performance of a block of software code is to evaluate what code will run as n approaches infinity. Items that are constant time (regardless of if they run once or x times independent of input) are reduced down to O(1). The key factors that contribute to Big O are loops – a single for loop or similar construct through recursion that runs once for all n is O(n); a nested for loop would be O(n2). Search algorithms that rely on presorted data sets are usually a logarithmic power of n; for example binary search (Wikipedia contributors, 2018) is O(log2 n). When calculating Big O for a code block, function, or software system, lower order terms are ignored, and just the largest Big O notation is used; for example if a code block is O(1) + O(n) + O(n3) it would be denoted as O(n3).

Despite the value of understanding the theoretical upper bound of software in an ideal situation, there are many difficulties that arise during implementation that can make Big O difficult to calculate and which could make a large Big O faster than a small Big O under actual input conditions. Some key takeaways to temper a mathematical evaluation of Big O are:

* Constants matter when choosing an algorithm - for example if one algorithm is O(56n2), there exists some n where O(n3) is faster.
* Average or best case run time might be more relevant. Although Big O can indicate worst case performance, perhaps average case is more meaningful. Depending on input and operations performed, shortcuts might be implemented to make worst case performance less likely.
* Big O evaluation of algorithms in high level languages are often hard to quantify. In some cases, source code is not available or functions may be calling other functions that were written by someone else. This is best illustrated with an example – without looking at the R or C/C++ code behind each these function calls, it is impossible to know what Big O this should be evaluated to:

if (!missing(id)) {

ids <- dplyr::select(data, !!dplyr::enquo(id))

} else {

ids <- data.frame()[1:nrow(data), ]

}

dplyr::bind\_cols(ids, ccc\_mat\_rcpp(dxmat, pcmat, icdv))

For additional details on Big O notation, see the excellent and broadly understandable introduction to Big O notation by Justin Abrahms (Abrahms, 2016).

**CODE PROFILING/BENCHMARKING**

As discussed throughout this section, optimization is a key aspect of software development, especially with respect to large datasets. Although identification of performance targets and a mathematical analysis of algorithms are important steps, the final result must be tested and verified. The only way to know if your software will perform adequately under ideal (and non-ideal) circumstances is to use benchmarking and code profiling tools. Code profilers show how a software behaves and what functions are being called while benchmarking tools generally focus on just execution time – though some tools combine both profiling and benchmarking. In R, some of the common tools are bench, benchr, microbenchmark, tictoc, Rprof (R Core Team, 2018), proftools (Tierney & Jarjour, 2016), and profvis.

If, after implementation has been completed, the software functions correctly, and performance targets have not been met, look to optimize your code. Follow an iterative process of profiling to find bottlenecks, making software adjustments, testing small sections with benchmarking and then repeating the process with overall profiling again. If at any point in the process you discover that due to input size, functional requirements, hardware limitations, or software dependencies you cannot make a significant impact to performance, consider stopping further optimization efforts (Burns, 2012).

As with software testing and software bugs, the Pareto principle applies, though some put the balance between code and execution time is closer to 90% of time is in 10% of the code or even as high as 99% in 1% (Xochellis, 2010; Bird, 2013). Identify the biggest bottlenecks via code profiling and focus only on the top issues first. As an example of how to perform code profiling and benchmarking in R, do the following:

First, run profvis to identify the location with the largest execution time:

library(pccc)

icd10\_large <- feather::read\_feather("icd10\_sample\_large.feather")

profvis::profvis({ccc(icd10\_large[1:10000,],

id = id,

dx\_cols = dplyr::starts\_with("dx"),

pc\_cols = dplyr::starts\_with("pc"),

icdv = 10)}, torture = 100)

In Figure 5 you can see a visual depiction of memory allocation, known as a “Flame Graph”, as well as execution time and call stack. By clicking on each item in the stack you will be taken directly to the relevant source code and can see which portions of the code take the most time or memory allocations. Figure 6 is a depiction of the data view which shows just the memory changes, execution time, and source file.

Once the bottleneck has been identified, if possible extract that code to a single function or line that can be run repeatedly with a library such as microbenchmark or tictoc to see if a small change either improves or degrades performance. Test frequently and make sure to compare against previous versions. You may find that something you thought would improve performance degrades performance. As a first step we recommend running tictoc to get general timings such as this:

library(tictoc)

tic("timing: r version")

out <- dplyr::bind\_cols(ids, ccc\_mat\_r(dxmat, pcmat, icdv))

toc()

tic("timing: c++ version")

dplyr::bind\_cols(ids, ccc\_mat\_rcpp(dxmat, pcmat, icdv))

toc()

timing: r version: 37.089 sec elapsed

timing: c++ version: 5.087 sec elapsed

As with previous timings, while we’re showing PCCC calls, any custom function of block of code you have can be compared against an alternative version to see which performs better. The above blocks of code call the core functionality of the PCCC package – one implemented all in R, the other with C++ for the matrix processing and string matching components; see sourcecode available at <https://github.com/magic-lantern/pccc/blob/no_cpp/R/ccc.R> for full listing.

After starting with high level timings, next run benchmarks on specific sections of code such as in this example comparing importing a package vs using the package reference operator using bench:

library(bench)

set.seed(42)

bench::mark(

package\_ref <- lapply(medium\_input, function(i) {

if(any(stringi::stri\_startswith\_fixed(i, 'S'),na.rm = TRUE))

return(1L)

else

return(0L)

}))

# A tibble: 1 x 14

expression min mean median max `itr/sec` mem\_alloc

<chr> <bch> <bch> <bch:> <bch> <dbl> <bch:byt>

1 package\_r… 547ms 547ms 547ms 547ms 1.83 17.9MB

library(stringi)

bench::mark(

direct\_ref <- lapply(medium\_input, function(i) {

if(any(stri\_startswith\_fixed(i, 'S'),na.rm = TRUE))

return(1L)

else

return(0L)

}))

# A tibble: 1 x 14

expression min mean median max `itr/sec` mem\_alloc

<chr> <bch> <bch> <bch:> <bch> <dbl> <bch:byt>

1 direct\_re… 271ms 274ms 274ms 277ms 3.65 17.9MB

The above test was run on a virtual machine running Ubuntu 16.04.5 LTS using R 3.4.4.

One benefit of bench::mark over microbenchmark is that bench reports memory allocations as well as timings, similar to data shown in profvis. Through benchmarking we found that for some systems/configurations the use of the “::” operator, as opposed to importing a package, caused a noticeable performance hit. Also widely known (Gillespie & Lovelace, 2017) and found to be applicable here is that the use of matrices are preferred for performance reasons over data.frame or tibbles. Matrices do have different functionality which can require some re-work when converting from one to another. Another key point found is that an “env” with no parent environment is significantly faster (up to 50x) than one with a parent env. In the end, optimization efforts resulted in reducing run time by 80%.

One limitation with R profiling tools is that if the code to be profiled executes C++ code, you will get no visibility into what is happening once the switch from R to C++ has occurred. As shown in Figure 7, visibility into timing and memory allocation stops at the .Call() function. In order to profile C++ code, you need to use non-R specific tools such as XCode on macOS or gprof on Unix based OSes. See “R\_with\_C++\_profiling.md” in our source code repository for some guidance on this topic.

Some general lessons learned from profiling and benchmarking:

* “Beware the dangers of premature optimization of your code. Your first duty is to create clear, correct code.” (Knuth, 1974; Burns, 2012) Never optimize before you actually know what is taking all the time/memory/space with your software. Different compilers and core language updates often will change or reverse what experience has previously indicated as sources of slowness. Always benchmark and profile before making a change.
* Start development with a high-level programming language first – Developer/Researcher time is more valuable than CPU/GPU time. Choose the language that allows the developer/researcher to rapidly implement the desired functionality rather than selecting a language/framework based on artificial benchmarks (Kelleher & Pausch, 2005; Jones & Bonsignour, 2011).
* Software timing is highly OS, compiler, and system configuration specific. What improves results greatly on one machine and configuration may actually slow performance on another machine. Once you decided to put effort into optimization, make sure you test on a range of realistic configurations before deciding that an “improvement” is beneficial (Hyde, 2009).
* If you’ve exhausted your options with your chosen high-level language, C++ is usually the best option for further optimization. For an excellent introduction to combining C++ with R via the library Rcpp, see (Eddelbuettel & Balamuta, 2017).

For some additional information on R optimization, see (Wickham, 2014; Robinson, 2017).