

ECS145 Term Project: Package “findR4”

Stephen Buckley, Dustin Cai, Robin Chang, Katie Kwak

2019-03-19

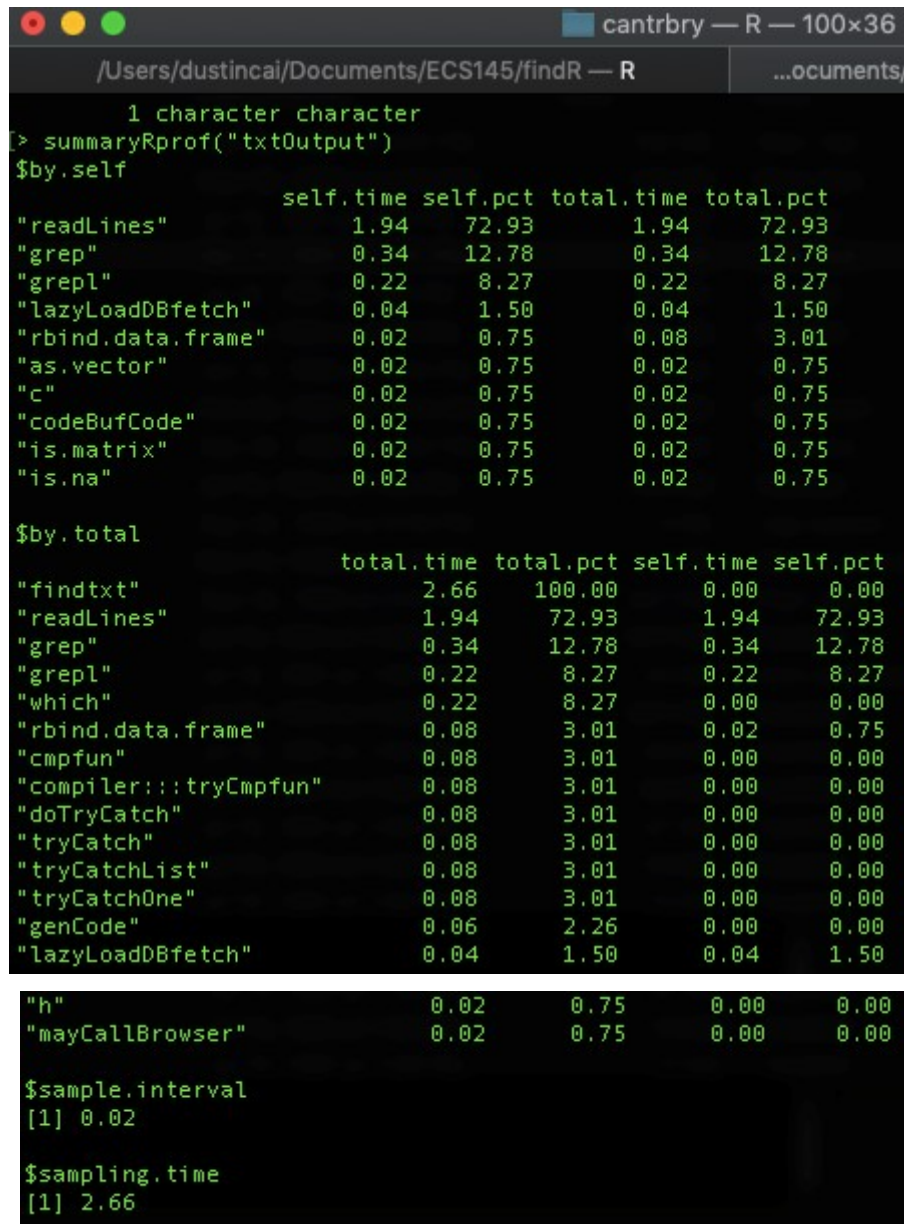
Contents

Introduction / Vital Information	2
Implementation / Solution Design	4
Process Description	5
Test Description and Results	5
Conclusion	8
Group Member Contributions	9
Appendix Code Listings	9
References	9

Introduction / Vital Information

The findR package is designed to scan for patterns in a user-defined path of directories and its subdirectories. It is a package consisting solely of R files: findPDF, findRmd.R, findRscript.R, findtxt.R, and reminder.R. Each R file searches through files of their respective formats for the given pattern and prints the total number of matches. The package also allows you to copy the files found into a new folder.

Calling the package specifically outputs the number of (ex. PDF files) specified files within a given path and outputs the number of pattern finds within that directory and its subdirectories. We analyzed the speed of the findR functions using the functions Rprof() and summaryRprof() to see where the program spends most of its time. This allowed us to strategize on where our improvements would make the most difference.



```
1 character character
[> summaryRprof("txtOutput")
$by.self
              self.time self.pct total.time total.pct
"readLines"      1.94    72.93      1.94    72.93
"grep"           0.34    12.78      0.34    12.78
"grepl"          0.22     8.27      0.22     8.27
"lazyLoadDBfetch" 0.04     1.50      0.04     1.50
"rbind.data.frame" 0.02     0.75      0.08     3.01
"as.vector"       0.02     0.75      0.02     0.75
"c"              0.02     0.75      0.02     0.75
"codeBufCode"     0.02     0.75      0.02     0.75
"is.matrix"       0.02     0.75      0.02     0.75
"is.na"           0.02     0.75      0.02     0.75

$by.total
              total.time total.pct self.time self.pct
"findtxt"        2.66    100.00      0.00      0.00
"readLines"      1.94    72.93      1.94    72.93
"grep"           0.34    12.78      0.34    12.78
"grepl"          0.22     8.27      0.22     8.27
"which"          0.22     8.27      0.00      0.00
"rbind.data.frame" 0.08     3.01      0.02     0.75
"cmpfun"         0.08     3.01      0.00      0.00
"compiler:::tryCmpfun" 0.08     3.01      0.00      0.00
"doTryCatch"     0.08     3.01      0.00      0.00
"tryCatch"       0.08     3.01      0.00      0.00
"tryCatchList"   0.08     3.01      0.00      0.00
"tryCatchOne"    0.08     3.01      0.00      0.00
"genCode"        0.06     2.26      0.00      0.00
"lazyLoadDBfetch" 0.04     1.50      0.04     1.50

"h"              0.02     0.75      0.00      0.00
"mayCallBrowser" 0.02     0.75      0.00      0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 2.66
```

[Prof results on findtxt.R from findR package]

Here we tested the `findtxt()` function searching for the pattern “and” in a collection of large text files. The total execution time was 1.34 seconds. The `readLines()` function was understandably the most expensive part of the process, taking up 0.98 seconds and creating a bottleneck for the function there.

```
34   if (length(fls) > 0) {  
35     # Scan R Markdown files for pattern  
36     hits <- NULL  
37     for (i in 1:length(fls)) {  
38       if (case.sensitive == FALSE) {  
39         pattern <- tolower(pattern)  
40         a <- tolower(readLines(fls[i], warn = F))  
41       } else {  
42         a <- readLines(fls[i], warn = F)  
43       }  
44     }  
45   }  
46 }  
47  
48  
49  
50  
51
```

[Snippet of `readlines()` from `Findtxt.R`]

Next, we tested `findPDF()`:

\$by.total	total.time	total.pct	self.time	self.pct
"findPDF"	98.42	100.00	0.00	0.00
"doTryCatch"	97.84	99.41	0.00	0.00
"suppressMessages"	97.84	99.41	0.00	0.00
"try"	97.84	99.41	0.00	0.00
"tryCatch"	97.84	99.41	0.00	0.00
"tryCatchList"	97.84	99.41	0.00	0.00
"tryCatchOne"	97.84	99.41	0.00	0.00
"withCallingHandlers"	97.84	99.41	0.00	0.00
"pdfutils::pdf_text"	97.70	99.27	0.00	0.00
"poppler_pdf_text"	97.70	99.27	0.00	0.00
".Call"	97.44	99.00	97.44	99.00
"grep"	0.30	0.30	0.30	0.30
"grepl"	0.26	0.26	0.26	0.26
"loadfile"	0.26	0.26	0.00	0.00
"which"	0.26	0.26	0.00	0.00
"readBin"	0.24	0.24	0.24	0.24
":"	0.14	0.14	0.00	0.00
"asNamespace"	0.14	0.14	0.00	0.00
"getExportedValue"	0.14	0.14	0.00	0.00
"getNamespace"	0.14	0.14	0.00	0.00
"loadNamespace"	0.14	0.14	0.00	0.00
"namespaceImportFrom"	0.14	0.14	0.00	0.00
"methods::cacheMetadata"	0.12	0.12	0.00	0.00
"lazyLoadDBfetch"	0.10	0.10	0.10	0.10
"<Anonymous>"	0.10	0.10	0.00	0.00
".updateMethodsInTable"	0.08	0.08	0.00	0.00
"FUN"	0.04	0.04	0.04	0.04

[Rprof results on findPDF.R from findR package part I]

".checkGroupSigLength"	0.04	0.04	0.00	0.00
"lapply"	0.04	0.04	0.00	0.00
".recMembers"	0.02	0.02	0.02	0.02
"attr"	0.02	0.02	0.02	0.02
".getGeneric"	0.02	0.02	0.00	0.00
"cmp"	0.02	0.02	0.00	0.00
"cmpCall"	0.02	0.02	0.00	0.00
"cmpfun"	0.02	0.02	0.00	0.00
"compiler:::tryCmpfun"	0.02	0.02	0.00	0.00
"exists"	0.02	0.02	0.00	0.00
"genCode"	0.02	0.02	0.00	0.00
"get"	0.02	0.02	0.00	0.00
"getGeneric"	0.02	0.02	0.00	0.00
"getGroupMembers"	0.02	0.02	0.00	0.00
"getInlineHandler"	0.02	0.02	0.00	0.00
"methods:::getGenerics"	0.02	0.02	0.00	0.00
"rbind.data.frame"	0.02	0.02	0.00	0.00
"tryInline"	0.02	0.02	0.00	0.00
"unique"	0.02	0.02	0.00	0.00
"unlist"	0.02	0.02	0.00	0.00
\$sample.interval				
[1] 0.02				
\$sampling.time				
[1] 98.42				

[Rprof results on findPDF.R from findR package part II]

Here we used the function `findPDF()` to search for the string “and” on 93.5mb worth of PDF data. We ran Rprof on the package according to 0.02 sample intervals with a total runtime of 98.42 seconds.

We found that the most expensive portion of the `findPDF()` function was the try catch statements where the PDF is converted into a readable format for the `grep` function. This bottleneck process took 97.84 seconds of the 98.42 total runtime. The listed times are when the R process is running and executing the R commands. After analyzing both functions, we were able to pinpoint the areas where our time would best be spent.

Implementation / Solution Design

We determined that the effect of the bottlenecks could be minimized if we were to minimize the cost of `grep()`, a command-line function that searches a data set for a regular expression. This function is already able to be called with .txt files, and “findPDF.R” uses `readlines()` to convert to a text readable format. By that assumption, we were able to conclude that if we were to make the `grep()` call function less costly for the “findtxt.R”, that the overall package would be made more efficient as a result.

In our attempt to improve the runtime of the function, we introduced multi-threading into the package through a new C++ file and make the call to `grep()` from there. This approach drew inspiration from Python’s threading methods. This way we take advantage of C++’s efficiency and apply that to the slowest portion of the process while still retaining the core of our R package. Our implementation also removed the loop implementations from the initial findR package and replaced them with the equivalent for loops in C++ in order to lower the overall cost of findR. We understand that for loops in R are slow because it is an interpreted language and carries a lot of overhead every time a user-defined function is called. While R needs to create an environment for execution, assign arguments, etc, C just pushes the arguments onto a stack and

pops the arguments off. The internal C loop is faster than the `apply()` of R; as `apply()` does more work requiring name lookups.

Process Description

The new R files individually calls `cGrep` from the `threadedFind.cpp` file and initializes the list of character vector files as well as whether or not the user wants the pattern to be case sensitive. `Rcpp` was called in our “`threadedFind.cpp`” file when the character vector list of file names needed to be accessed. Our new code uses the for loops to run two threads in parallel after splitting the list of files in half and providing one half to each thread to run on. While this solution design does not speed up the `grep` command, it does divide the labor across multiple threads, which in theory should increase the speed of the overall application. Each thread calls the `threadFunc()` function and uses `grep` to search through its own list of files which is distributed upon thread creation.

The `execl()` function was introduced to enable running `grep()` in the command line within each directory on the given list of files. We then came across the issue that `execl()` would not return after it is successful at starting the program. To resolve this, we decided to `fork()` the process and have the child make the call to `grep`, which then passes the result to the parent process via pipes. This is all within one for loop each time a thread is called in our `cpp` file which is more efficient compared to R’s loops space exhausting for loop calls.

We ran into complications with the ordering of our output so we decided to try pipelining the output files so that multiple instructions could be overlapped in execution. We did this by opening a file descriptor with `fd[2]`. This theoretically duplicates the file descriptor 1 (standard output) to the side of the pipe that we would like to write to. Then we closed the other end of the pipe while inside the child process with `close(fd[0])` so that the parent process can read the output of the child process by duplicating the descriptor file 0 to `fd[0]` which is the side of the pipe to read from. Then we close the side of the pipe we are not using in the parent process which is done with `close(fd[1])`.

Finally, `execl()` is then called from the child process with `execl()` (referenced in line 65 of `threadedFind.cpp`); and the command takes the input from `file[0]` and sends it to standard output where we can call `grep` in the command line with the specific file being searched.

With this well thought out solution, in terms of the bottleneck implementing `grep()` in `findtxt.R` with `cpp` and running our threads in parallel as well as pipelining the forked processes, we could assume that we would see less expensive memory access for function calls, and more efficient run times as a result while running `Rprof()` profiling results if the entirety of our new package was implemented correctly.

Test Description and Results

In our `FindR4` package, you can see that the times are faster than the pure R package by referencing:


```

Number of text files scanned: 13
Number of text files with matching content: 6
Total number of matches: 38158
Warning messages:
1: In grep(pattern, a) : input string 1 is invalid in this locale
2: In grep(pattern, a) : input string 1 is invalid in this locale
3: In grep(pattern, a) : input string 2 is invalid in this locale
4: In grep(pattern, a) : input string 4 is invalid in this locale
5: In grep(pattern, a) : input string 5 is invalid in this locale
6: In grep(pattern, a) : input string 7 is invalid in this locale
7: In grep(pattern, a) : input string 1 is invalid in this locale
8: In grep(pattern, a) : input string 3 is invalid in this locale
9: In grep(pattern, a) : input string 1 is invalid in this locale
> Rprof(NULL)
> summaryRprof("OUTPUT")
$by.self
      self.time self.pct total.time total.pct
"readLines"    0.98   74.24      0.98   74.24
"grep"          0.14   10.61      0.14   10.61
"grepl"         0.12    9.09      0.12    9.09
"which"         0.02    1.52      0.14   10.61
".deparseOpts"  0.02    1.52      0.02    1.52
"factor"        0.02    1.52      0.02    1.52
"Make.row.names" 0.02    1.52      0.02    1.52

$by.total
      total.time total.pct self.time self.pct
"findtxt"       1.32   100.00    0.00    0.00
"readLines"     0.98   74.24    0.98   74.24
"grep"          0.14   10.61    0.14   10.61
"which"         0.14   10.61    0.02    1.52
"grepl"         0.12    9.09    0.12    9.09
"rbind.data.frame" 0.04    3.03    0.00    0.00
".deparseOpts"  0.02    1.52    0.02    1.52
"factor"        0.02    1.52    0.02    1.52
"Make.row.names" 0.02    1.52    0.02    1.52
"%in%"          0.02    1.52    0.00    0.00
"as.data.frame.character" 0.02    1.52    0.00    0.00
"as.data.frame"  0.02    1.52    0.00    0.00
"cbind.data.frame" 0.02    1.52    0.00    0.00
"data.frame"     0.02    1.52    0.00    0.00
"deparse"       0.02    1.52    0.00    0.00
"mode"          0.02    1.52    0.00    0.00

```

[Rprof results on findtxt.R from findR4 package part I]

```

$sample.interval
[1] 0.02

$sampling.time
[1] 1.32

```

[Rprof results on findtxt.R from findR4 package part II]

```

> r <- findPDF(path = ".", pattern = "and")
Number of PDF files scanned: 6
Number of PDF files with matching content: 3
Total number of matches: 3595
> Rprof(NULL)
> summaryRprof("results")
$by.self
              self.time self.pct total.time total.pct
".Call"          95.82   98.91      95.82   98.91
"grep"           0.30    0.31       0.30    0.31
"grepl"          0.26    0.27       0.26    0.27
"readBin"        0.24    0.25       0.24    0.25
"lazyLoadDBfetch" 0.06    0.06       0.08    0.08
"is"             0.04    0.04       0.04    0.04
".resetInheritedMethods" 0.02    0.02       0.04    0.04
".registerS3method" 0.02    0.02       0.02    0.02
"%in%"           0.02    0.02       0.02    0.02
"find.package"   0.02    0.02       0.02    0.02
"getNamespaceName" 0.02    0.02       0.02    0.02
"mkenv"          0.02    0.02       0.02    0.02
"normalizePath"  0.02    0.02       0.02    0.02
"parse"          0.02    0.02       0.02    0.02

$by.total
              total.time total.pct self.time self.pct
"findPDF"          96.88   100.00      0.00    0.00
"doTryCatch"       96.32    99.42      0.00    0.00
"suppressMessages" 96.32    99.42      0.00    0.00
"try"              96.32    99.42      0.00    0.00
"tryCatch"         96.32    99.42      0.00    0.00
"tryCatchList"     96.32    99.42      0.00    0.00
"tryCatchOne"      96.32    99.42      0.00    0.00
"withCallingHandlers" 96.32    99.42      0.00    0.00
"pdfutils::pdf_text" 96.10    99.19      0.00    0.00
"poppler_pdf_text" 96.10    99.19      0.00    0.00
".Call"           95.82    98.91     95.82    98.91
"grep"            0.30    0.31       0.30    0.31
"loadfile"        0.28    0.29       0.00    0.00
"grepl"           0.26    0.27       0.26    0.27
"which"           0.26    0.27       0.00    0.00

```

[Rprof results on findPDF.R from findR4 package]

Now comparing the original findPDF() function to our new function, the time spent went from 98.42 to 96.88 seconds, increasing our speed and decreasing our time by 1.54 seconds!

We were aware that, in theory, a multithreaded approach would increase the efficiency of the package. However, due to the multitude of factors to consider, we were skeptical if this would prove true in practice. After testing our new implementation, we found that our design did in fact improve the speed of findtxt() and findPDF().

Comparing the original findtxt() function to our own with the same files and user-defined pattern, the time decreased from 1.34 to 1.32, increasing our speed by 0.2 seconds!

```

Katies-MacBook-Air-2:Term Project Katie$ R CMD INSTALL findR4_1.0.tar.gz
* installing to library '/Library/Frameworks/R.framework/Versions/3.4/Resources/library'
* installing *source* package 'findR4' ...
** libs
clang++ -std=gnu++11 -I/Library/Frameworks/R.framework/Resources/include -DNDEBUG -I"/Library/Frameworks/R.framework/Versions/3.4/Resources/library/Rcpp/include" -I/usr/local/include -fPIC -Wall -g -O2 -c RcppExports.cpp -o RcppExports.o
clang++ -std=gnu++11 -I/Library/Frameworks/R.framework/Resources/include -DNDEBUG -I"/Library/Frameworks/R.framework/Versions/3.4/Resources/library/Rcpp/include" -I/usr/local/include -fPIC -Wall -g -O2 -c threadedFind.cpp -o threadedFind.o
clang++ -std=gnu++11 -dynamiclib -Wl,-headerpad_max_install_names -undefined dynamic_lookup -single_module -multiply_defined suppress -L/Library/Frameworks/R.framework/Resources/lib -L/usr/local/lib -o findR4.so RcppExports.o threadedFind.o -F/Library/Frameworks/R.framework/.. -framework R -Wl,-framework -Wl,CoreFoundation
ld: warning: text-based stub file /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation.tbd and library file /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation are out of sync. Falling back to library file for linking.
installing to /Library/Frameworks/R.framework/Versions/3.4/Resources/library/findR4/libs
** R
** inst
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (findR4)

```

[Evidence of Successful Installation]

```

Katies-MacBook-Air-2:Term Project Katie$ R CMD build findR4
* checking for file 'findR4/DESCRIPTION' ... OK
* preparing 'findR4':
* checking DESCRIPTION meta-information ... OK
* cleaning src
* checking for LF line-endings in source and make files and shell scripts
* checking for empty or unneeded directories
* building 'findR4_1.0.tar.gz'

```

[Successful Build]

```

[> findtxt(path=".", pattern="a[e-h]", case.sensitive=FALSE)
Number of text files scanned: 2
Number of text files with matching content: 2
Total number of matches: 2
  path_to_file line
1 ./inst/txt1.txt    1
2 ./inst/txt5.txt    1

```

[Successful Execution]

Please Note: We had to rename the new package to “findR4” because there was confusion between findR (reference) and our modified version.

Conclusion

A wise man once said “All chemistry experiments work,” and that motto definitely held true for this experiment. A lot of thought and work was put into during the process of our experiment. After struggling to improve the runtime of the overall package, we understood the appeal and simplicity of writing a package with integrate a low level language in an attempt to improve the speed of a higher level Semi-functional language such as R (as side effects can be purposely made, of course). Attempting this experiment also brought into question of the goal of the package. While the purpose of findR is to search for a pattern in a

dataset and return the total number of matches, could using R have been the most efficient and least costly way to accomplish this? It is difficult to limit your options to just one language when taking into account the efficient use of both memory and runtime speed.

Our results satisfied our goal to speed up the runtime of the package through our use of loop efficiency, multi-threading, and pipelining implementations of Cpp in our R package.

Group Member Contributions

1. Stephen Buckley, sgbuckley@ucdavis.edu
Threading features of `threadingFind.cpp`.
Extracting output of shell through pipelining.
Introduced idea of parallel multi-threading.
2. Dustin Cai, dvcai@ucdavis.edu
Tested data with `Rprof` and analyzed results for bottlenecks.
Gathered large datasets to test `findR` functions.
Contributed to the report.
3. Robin Chang, rochang@ucdavis.edu
Understanding R package structures and interfacing R package to run C++.Contributed to using `Rcpp` in `cpp` files.
Contributed to written report analysis.
Helped with debugging of `cGrep()`.
4. Katie Kwak, kykwak@ucdavis.edu
Debugging `execl()` and `grep()` utilization.
Worked on communication between `findtxt` and `cpp`.
Utilized/added `rcpp sugar` to make R and C++ types compatible.

Appendix Code Listings

`findPDF.R`
`findRmd.R`
`findRscript.R`
`findtxt.R`
`reminder.R`
`threadedFind.cpp`

References

<https://www.dreamincode.net/forums/topic/239160-program-works-what-is-it-doing-fork-execl-pipe-example/>

"The Art of R Programming" by Norman Matloff