Parallel Computing Through Code Analysis

Clark Fitzgerald

May 25, 2017

Abstract

Conventional systems for parallel programming require users to modify existing code to take full advantage of a platform's computational capabilities. In this proposal we consider automated code analysis methods to detect the potential for parallel execution in R code. The results of the analysis can then be used to programmatically rewrite and execute semantically equivalent parallel instructions, without requiring the user to modify their original code. We consider a motivating case study analyzing the operating characteristics of California's highway traffic sensor stations on hundreds of gigabytes of traffic sensor data.

1 Code, Data, Platform

For our purposes in this document, **code** is a script to be executed. **Data** could be data in memory, single or multiple files, a memory-mapped file, a database, a parallel file system, etc. **Platform** is the computing setup, such as a laptop with 4 cores, a cluster of machines communicating over a network, or a server with 2 GPUs and a high bandwidth connection. Knowing the combination of {Code, Data, Platform} allows one to select an appropriate strategy for efficient computation. Figure 1 illustrates the high level goal of modifying the

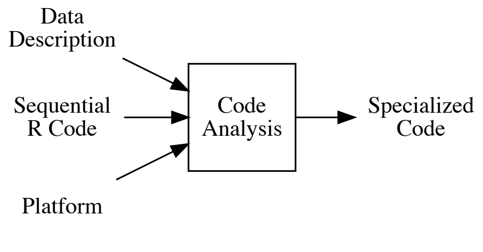

Figure 1: Given sequential code and a description of the data we seek a modified computational parallel plan tailored to a specific platform.

code to run efficiently with a particular platform and data, while preserving the semantics of the input code.

This project is in the spirit of compiling R, since it treats R code as a set of high level directions, and generates alternative code for execution [5]. In the future we hope to connect and extend this project using recent work on compilation. Note that the system does not invent new algorithms on the fly. However, it should be capable of tuning existing parameters to the problem at hand.[1]

2 Motivating Example

The example in this section is interesting for the following reasons:

- The results are relevant to domain scientists in traffic engineering.
- The size of the data exceeds the memory of a single server.
- Parallel programming can increase the speed by 1-2 orders of magnitude.

The California Department of Transportation (CalTrans) collects traffic data through sensors embedded in highways. The sensors measure three quantities: count of passing vehicles (flow), time during which a vehicle is directly above the sensor (occupancy), and average velocity [3]. Every thirty seconds they produce a new data point. $43,680$ sensors in California measuring 3 parameters multiplied by $2 \times 60 \times 24$ measurements per day results in 377 million new data points per day. CalTrans provides the raw data for public download. It's organized into files containing observations for one day for each of 12 management districts in California.

Traffic engineers model traffic flow as a function of occupancy for each sensor as in figure 2. This relationship is called the *fundamental diagram* because it captures the operating characteristic of this section of road [1]. Robust regression can be used to fit the fundamental diagram for one station, approximating the minimization of the L1 norm of the residuals as in [6]. From the R language this can be done easily and efficiently, for example with the `rlm` (robust linear model) function in the MASS package [13]. `rlm` can be used as a building block for a user defined `fit_fd` function, and the fundamental diagram can be fit on a single station with `fit_fd(station1)`. Then all stations can be fitted by grouping the data by station and applying `fit_fd()` to each group. R code expresses this computation succintly:

```
by(alldata, INDICES = station, FUN = fit_fd)
```

I would like to apply this code to a subset of the data consisting of observations in the San Francisco Bay Area for part of 2016. This data is 134 GB on disk. It won't fit into memory, so this code won't run. Even if it did fit in memory, it would be slow because `by()` won't run `fit_fd()` in parallel across the 1000's of different stations[2].

[1]An example of a parameter to be tuned is the chunk size n_j as described in 3.1.

[2] Depending on overhead it may be less efficient to run `fit_fd()` in parallel.

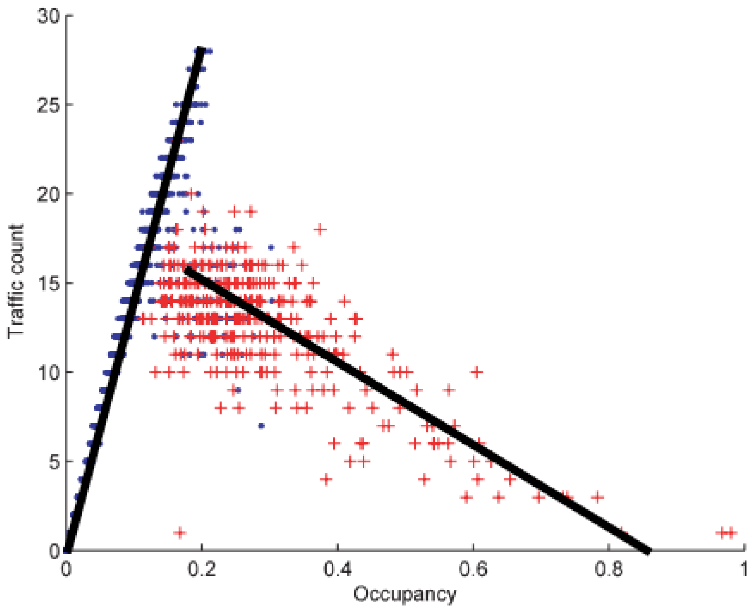

Figure 2: The fundamental diagram in traffic engineering.

It would be easier to perform this computation if the data were organized in files for each station rather than in files for each day. One approach is to reorganize the data on disk into this file structure. I did this using a simple single threaded R program, and it took 23 hours to run on the 2016 Bay Area data. Throughput for a conventional hard disk[3] is around 100 MB/s, so an approximate lower bound for reading then writing this reorganized data is $2 * 134000/100$ seconds, or 45 minutes for 134 GB. The simple code is 30 times slower than this. Small inefficiencies add up.

Another drawback is that the programmer must write very specific instructions to reorganize the data for computations grouped by station. Any parallel programming will make this even more specific to the data and platform. Subsequent slight modifications of the analysis may warrant a completely different approach. For example, using least squares instead of robust regression allows the use of an updating algorithm such as provided by the `biglm` package [7], so there is no need to have all the data for one group in memory at once.

What if we had a system that could inspect the idiomatic R code above that works on small data sets, and then automatically take steps to scale and to parallelize the operations? It should also be capable of tuning for the specifics of the platform and the data. This is the goal of the research.

[3]Other technologies such as parallel filesystems can increase throughput.

3 Simple Example

We use an intentionally simple example to see exactly what translated code might look like, and to illustrate a common strategy for parallel computing. The reader familiar with parallel R can skip this section. Consider computing the mean,

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

where the x_i's are i.i.d. $\sim t(d)$. In R this code is written:

```
xbar = mean(rt(n, d))
```

Ordinarily, to evaluate this R will first build an intermediate vector $x = (x_1, \ldots, x_n)$ and then compute the mean. Eventually that unreferenced intermediate vector will be garbage collected.[4] Execution time increases linearly with n. Once n becomes large enough this vector will no longer fit into available physical memory, so the operating system will use swap space. On a machine with 8 GB memory this happens when $O(n) = 10^9$, causing the execution time to increase by an order of magnitude, roughly from 1 minute to 15 minutes. Once n exceeds memory and swap space R will not be able to allocate a large enough object, and the computation will fail with a memory error.

3.1 Sequential Execution

The following example illustrates how single threaded R can be modified to work with a data set that does not fit into memory. We do this by breaking the computation into chunks. The code below not run in parallel, but it does use the same functional programming patterns as parallel code.

Suppose $n = pn_j$ for integers p, n_j. The mean can be expressed as

$$\bar{x} = \frac{1}{n} \sum_{j=1}^{p} \sum_{i=1}^{n_j} x_{ij} = \frac{1}{p} \sum_{j=1}^{p} \frac{1}{n_j} \sum_{i=1}^{n_j} x_{ij} = \frac{1}{p} \sum_{j=1}^{p} \bar{x}_{\cdot j} \tag{2}$$

Conceptually, p is the number of chunks and n_j is the size of each chunk. Equation 2 can be directly translated into R code as follows:

```
partial_means = replicate(p, mean(rt(n_j, d)))
xbar = mean(partial_means)
```

In this code `replicate()` evaluates the expression `mean(rt(n_j, d))` p times sequentially, storing the intermediate results. If M is the size of physical memory in bytes then this

[4]Reducing the use of unnessary intermediate vectors would be helpful, but it's a second order consideration.

code is high performance in the sense that execution time will continue to be linear while $n < O(M^2)$, just as it was for small $n < O(M)$. The memory footprint is bounded since the intermediate vectors will be of length n_j. So small chunk sizes use less memory. On the other hand, large chunk sizes improve speed when computing on vectors in R, since the overhead of the R interpreter is amortized. One goal of the system is to balance this tradeoff in chunk sizes. Preliminary results suggest using chunk sizes with $n_j \geq 1000$ elements.

3.2 SNOW Cluster

A common way to write parallel R code is through the SNOW package, which stands for Simple Network Of Workstations. Clusters created by SNOW consist of independent worker R processes created by a manager process. The processes communicate over network sockets, and they may be on one or many physical machines. This is the type of cluster used by partools to implement Software Alchemy [9] [10]. The following code implements equation 2 in SNOW:

```
library(parallel)
p = floor(detectCores(logical = FALSE) / 2)
n_j = n / p
cluster = makeCluster(p)
clusterExport(cluster, c("n_j", "d"))
partial_means = unlist(
    clusterEvalQ(cluster, mean(rt(n_j, d))))
xbar = mean(partial_means)
```

Here's what each line of code does:

1. `library(parallel)` loads the parallel package, which R includes as a recommended package.
2. `p = floor(detectCores(logical = FALSE) / 2)` chooses the number of chunks in a principled way, to use half of the physical processors available on the platform. This respects other programs and users who may be on the system.
3. `n_j = n / p` chooses the chunk size based on n and p.
4. `cluster = makeCluster(p)` makes a local cluster with p workers. Given an existing cluster we would skip this step.
5. `clusterExport(cluster, c("n_j", "d"))` sends the variables n_j, d to the workers, so they will be available for the local computations. Using a system fork to create the cluster allows me to skip this step, since forked processes have access to the variables that existed when they were created.
6. `clusterEvalQ(cluster, mean(rt(n_j, d)))` sends the code `mean(rt(n_j, d))` from the manager to be evaluated on each worker.

7. `partial_means = unlist(...)` converts the list data structure to the intermediate numeric vector.

8. `xbar = mean(partial_means)` finally computes the mean from the intermediate result.

Given an existing cluster, this strategy is efficient, provided that the network latency is much less than the time required to evaluate the code. The manager sends only two numbers and the code `mean(rt(n_j, d))` for evaluation. The workers each return a single number. Therefore the program does parallel computing while avoiding large data transfers.

The code `xbar = mean(rt(n, d))` calls directly into C code, which makes analysis difficult. A practical way around this is to provide a mechanism for the user to annotate the function `rt()` indicating that it produces vectors and can be split using `replicate(p, rt(n_j, d))`. More ambitiously we could analyze the underlying C code after preprocessing to determine how it is vectorized.

4 Code Analysis

Code analysis involves the following steps:

- Parse code, gathering dependency information as explained in section 4.2.
- Use data description to determine if preprocessing of the data is required, for example reorganizing the files on disk as in section 2.
- Identify points to parallelize, see section 4.3.
- May experimentally evaluate expressions (after checking for side effects) to estimate how long they'll take.
- Relate potential parallel points to data description and timings to determine a parallel or serial strategy.
- Generate code that will run efficiently on the specific platform.

4.1 Overhead

Many of the performance characteristics of the different platforms can be understood in terms of the overhead to set them up, the speed of each machine, and then the latency and bandwidth for when data must be transferred. If the data is "small enough" then performance will be acceptable regardless of how well the platform is utilized or how efficient the code is. Conversely, if the data is "large enough" then *everything* matters, since small inefficiencies will be magnified when they occur millions of times, as in section 2. Therefore the focus here is on problems which take longer to run. A priori one doesn't even know if it's possible to run code more efficiently in parallel, because of the aforementioned overhead. Figure 3 compares the relative fixed sources of overhead associated with process based parallelism.

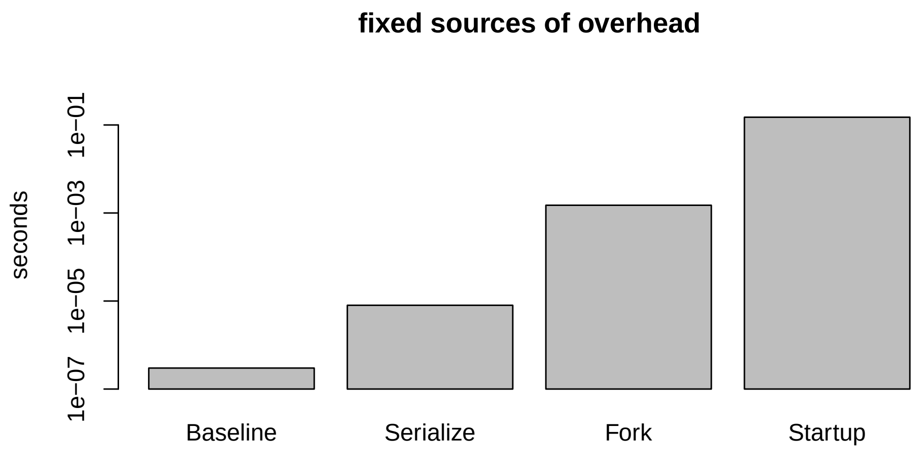

Figure 3: This figure shows the overhead associated with various operations by measuring the time to execute with small data. *Baseline* calls `twox(10)` where `twox = function(x) 2*x`. *Serialize* sends and receive an object on a local network socket connection. *Fork* calls a system level fork of the process to evaluate a function, returning the result to the current process. *Startup* starts a new independent instance of the R interpreter.

The evaluation of the following simple function captures the basic overhead associated with an interpreted language like R:

```
twox = function(x) 2*x
```

Every call to this function will incur a fixed overhead that is a couple hundred nanoseconds. Vectorization refers to passing longer vectors to the function to amortize this fixed cost. For example, if x is an integer vector of length 1 then the overhead takes 2 orders of magnitude more time than the actual computation. However, if x is of length 1500 then this overhead accounts for only 10 percent of the total execution time, so the fixed cost has been amortized.

4.2 Expression Dependency Graphs

We build on the CodeDepends package [12] for static code analysis. The evaluation model for interpreted languages is simple. Each expression of code is evaluated in the order that it appears in a text file. Informally each expression is a line of code. This can be viewed as a set of constraints on the evaluation order of the expressions:

- expression 1 executes before expression 2
- expression 2 executes before expression 3
- ...

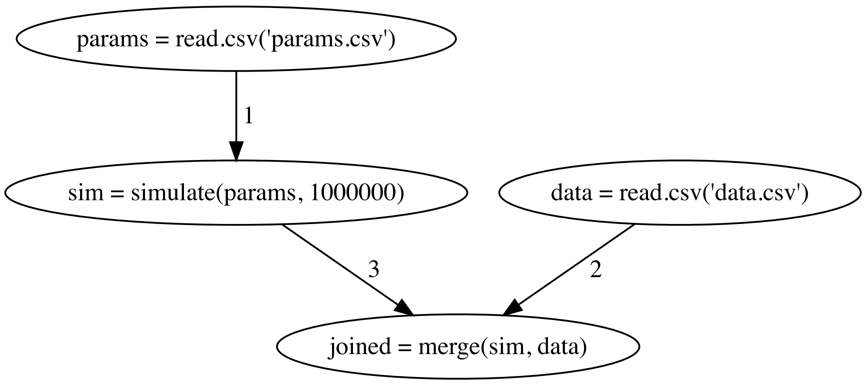
params = read.csv('params.csv')
1
sim = simulate(params, 1000000)
data = read.csv('data.csv')
3
2
joined = merge(sim, data)

Figure 4: Expression graph for the script on page 8. Nodes correspond to top level expressions.

What if these constraints are relaxed? Suppose expression 1 defines the variable `x`, which is not used until expression 17. Then one has the constraint:

- expression 1 executes before expression 17

This can be generalized into a directed graph which we'll refer to here as the **expression graph** by considering expressions as nodes and constraints as edges. The edges are implicit based on the order of the statements in the code. There is an edge from $i \to k$ if expression k depends on the execution of expression i. It's safe to assume $i < k$, because expressions appearing later in a program can't affect expressions which have already run. Hence the expression graph is acyclic, i.e. a DAG. The expression graph differs from a control flow graph because it treats control flow constructs such as `for()` loops as top level expressions and functions, just like the R interpreter. It would be possible to go into the control flow, but this can be done more efficiently at the R compiler level.

Listing 1: Simple script

```
params = read.csv('params.csv')
data = read.csv('data.csv')
sim = simulate(params, 1000000)
joined = merge(data, sim)
```

Figure 4 illustrates an expression graph. Let G_1 be the single node `data = read.csv('data.csv')` and G_2 be the subgraph containing the lines:

```
params = read.csv('params.csv')
sim = simulate(params, 1000000)
```

G_1 and G_2 are only connected by the final node `joined = merge(data, sim)`, so the expressions in G_1 and G_2 can run simultaneously. We use the expression graph to determine which variables need to be available to run a particular computation in parallel. The CodeDepends package also provides other important information, such as if the call has any side effects, ie. drawing lines on the current plot. This should not be done in parallel.

4.3 Entry points for parallelism

Three types of functions in R are of special interest when doing parallel computation: apply functions, vectorized functions, and reduce functions.

Apply functions are higher order functions which call the same function with different arguments. Conceptually, they are variations on the "map" step in the map reduce paradigm [2]. In base R, these include `lapply, apply, sapply, vapply, Map, mapply, by, tapply, outer, replicate`. These functions are essential to the idiomatic use of R as a functional language. They can all be parallelized. It remains to use the context provided by the data and the surrounding script to run them efficiently.

Vectorized functions apply functions elementwise. Basic math functions such as sin, log, and floor are simple examples of vectorized functions. A vectorized function f operating on an argument x of length n does:

$$f(x) = (f(x_1), \ldots, f(x_n)) \tag{3}$$

Partitioning x into p subvectors,

$$x = \left[(x_{11}, \ldots, x_{1n_1}), \ldots, (x_{p1}, \ldots, x_{pn_p})\right]$$

allows us to parallelize $f(x)$ by evaluating f on the subvectors as follows:

$$f(x) = \left[f(x_{11}, \ldots, x_{1n_1}), \ldots, f(x_{p1}, \ldots, x_{pn_p})\right].$$

If the subvectors are large enough then this can be efficient in R also. However, the overhead of splitting x into subvectors, evaluating in parallel, and recombining $f(x)$ may take more time than calling $f()$ itself. It becomes more efficient when combined with a reduce function, since there's less data to recombine.

The general form of the `Reduce()` function in R collapses elements into a single result by calling a function $f()$ repeatedly on the elements. For example `Reduce('+', 1:4)` is equivalent to `(((1 + 2) + 3) + 4)`. Common examples that can be written this way include `min, max, mean, sum, prod`. A more interesting example is to simultaneously join many data frames through `Reduce(merge, ...)`. Since we're computing on large data sets numerically stable algorithms should be used here, ie. Kahan summation [11]. Reduce style functions can be used to reduce the size of the data in a worker before transferring to the manager.

4.4 Data Description

The more we know about the structure and format of the data, the more efficiently we can write the code. For tabular data, we would like to know:

- **dimensions**- the number of rows and columns. Then we can determine whether there will be obvious memory issues and preallocate arrays.
- **data types**- boolean, float, character, etc. Specifying this avoids errors that can rise when inferring from text.
- **factor levels**- possible values for categorical data. Then we can preserve this information even if one value is rare and doesn't always appear in the data.
- **randomization**- has the data already been intentionally randomized? If it's random then we can easily statistically sample by reading the first rows.
- **sorted**- is the data sorted on a column? This allows streaming computations based on groups of this column.
- **layout**- are multiple files used? If each file stores data corresponding to some unit, ie. one file per day and we do a computation for each day then parallelization is natural across files.
- **index**- does data come from a database with an index? Then data elements can be efficiently acccessed by index.
- **offsets**- knowing a numeric array with n rows and p columns is stored in column major order potentially allows more efficient reads of subsets of the data.

The metadata mentioned above should be stored and preserved for subsequent computations. Not all of it is strictly necessary; indeed, the workhorse `read.table()` in base R reads tables in without any of this knowledge. In practice the metadata should be stored along with data, and this can be done using existing mature technologies such as Apache Avro.

4.5 Challenges

It would be quite ambitious to produce a complete system capable of producing optimal code on such different systems as multicore, GPU, and distributed. Some of the foundational capabilities aren't yet mature, i.e. compiling R code to an OpenCL kernel which runs on a GPU. Moreover, the semantics across the systems may differ. For example, a file backed `matrix` from the `bigmemory` package [4] has reference semantics, unlike an ordinary R matrix.

Non standard evaluation and dynamic scoping rules make some of the static code analysis difficult. For example, to fit a linear model with `lm(y~x, data)`, it's possibe that x is a global variable while y is a column name in `data`. Generally this can't be known until run time, since one could have the following code which randomly assigns x:

```
x = 1:10
if(sample(c(TRUE, FALSE), 1)){
    data$x = rnorm(10)
}
```

When identifying global variables to send to workers one can do the conservative thing and send both `x, data`.

Another challenge is handling mutable or more complex objects such as environments, closures, and reference classes. It's usually not possible to know the class of an object until runtime. But user and package code could be inspected to see if certain objects are used, for example by looking for calls to the function `setRefClass()`. The code analysis will have to respect the implicit connections between these objects.

5 Conclusion

Modifying code programmatically allows us to write the same base R code which will run in serial as in parallel. This provides several benefits. First, it's useful to have a working serial reference implementation. A serial version written for clarity should be simpler than any subsequent versions that attempt to improve performance [8]. Base R is stable, so once the code works it should continue to work. Finally, if the code runs on a different platform or with a different data set then one can automatically generate code to run more efficiently on that platform.

5.1 Related Technologies

Related technologies include Theano, tensorflow, dask, and arrayfire. With these packages the user explicitly builds computation graphs which are then executed in an efficient way based on the available architecture. They are more tailored to linear algebra / numerical computing, and are particularly popular for implementing neural networks. One possible path to explore is the translation of R code into these systems. This might be difficult when processing non numeric data.

5.2 Next Steps

My immediate plan is to build a prototype of the system targeting R's apply style functions on a single server *platform*, and *data* consisting of files on disk which may exceed memory. I'll test this prototype on the problem described in section 2. At a minimum this means preprocessing tools and a parallel tapply / group by. Once the prototype is functional I plan to extend the data model to streaming data through big data systems such as Apache Kafka. Running R along with a streaming system is interesting and relevant because it allows one to bring the statistical power of R beyond the offline analysis of static data and into large real time systems.

References

[1] Carlos F Daganzo. *Fundamentals of transportation and traffic operations.* Emerald Group Publishing Limited, 1997.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Zhanfeng Jia, Chao Chen, Ben Coifman, and Pravin Varaiya. The pems algorithms for accurate, real-time estimates of g-factors and speeds from single-loop detectors. In *Intelligent Transportation Systems, 2001. Proceedings. 2001 IEEE*, pages 536–541. IEEE, 2001.

[4] Michael J. Kane, John Emerson, and Stephen Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013.

[5] Duncan Temple Lang et al. Enhancing r with advanced compilation tools and methods. *Statistical Science*, 29(2):181–200, 2014.

[6] Jia Li and H Zhang. Fundamental diagram of traffic flow: new identification scheme and further evidence from empirical data. *Transportation Research Record: Journal of the Transportation Research Board*, (2260):50–59, 2011.

[7] Thomas Lumley. *biglm: bounded memory linear and generalized linear models*, 2013. R package version 0.9-1.

[8] N. Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*. Chapman & Hall/CRC The R Series. CRC Press, 2015.

[9] Norm Matloff, Clark Fitzgerald, with contributions by Alex Rumbaugh, and Hadley Wickham. *partools: Tools for the 'Parallel' Package*, 2017. R package version 1.1.6.

[10] Norman Matloff. Software alchemy: turning complex statistical computations into embarrassingly-parallel ones. *arXiv preprint arXiv:1409.5827*, 2014.

[11] Robert W. Robey, Jonathan M. Robey, and Rob Aulwes. In search of numerical consistency in parallel programming. *Parallel Computing*, 37(4–5):217 – 229, 2011.

[12] Duncan Temple Lang, Roger Peng, Deborah Nolan, and Gabriel Becker. *CodeDepends: Analysis of R code for reproducible research and code comprehension*, 2017. R package version 0.4-2.

[13] William N Venables and Brian D Ripley. *Modern applied statistics with S-PLUS*. Springer Science & Business Media, 2013.