Parallel Computing Through Code Analysis

Clark Fitzgerald

UC Davis

10 May 2017

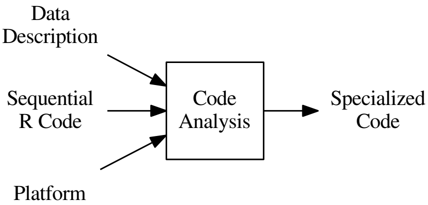


Table of Contents

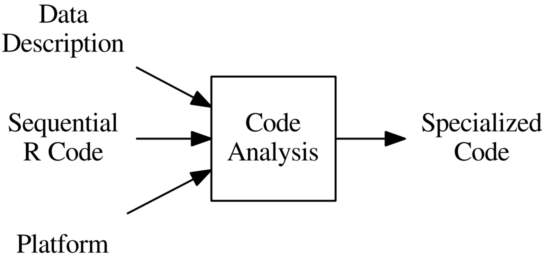
Modern platforms provide incredible computing power.

But they require expertise.

The broader goal is for users to write higher level code that also performs better.

- Parallel programming is a means to this end
- Compilation is another way
- Expertise in system rather than end user

We take a holistic approach to the computation.



The R language offers several benefits.

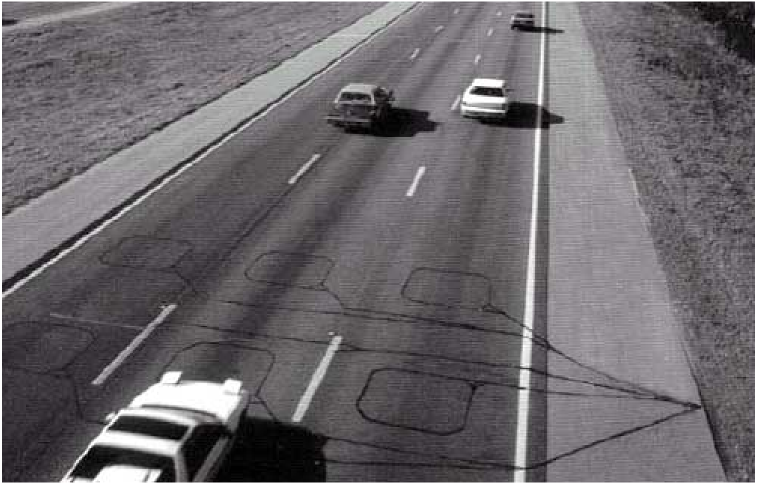
- Functional languages simplify parallel computing
- Widely used for statistics and data analysis
- Supports metaprogramming aka “programming on the language”

Table of Contents

The purpose of this example is to motivate the proposed research.

- Working with Professor Michael Zhang from Civil Engineering
- Illustrates complexity when computing with larger data sets

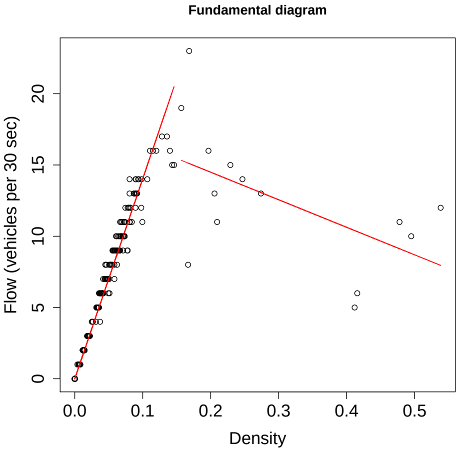
Loop detectors count vehicle flow, measuring velocity and density (time sensor is activated).



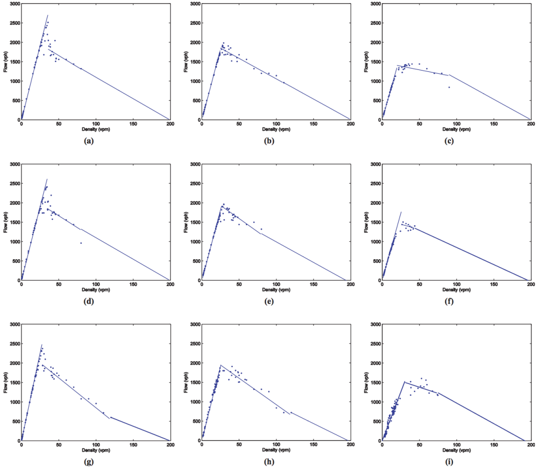
Caltrans Performance Measurement System (PeMS) records loop detector data for the whole state.

- Each sensor measures 3 quantities
- Data point every 30 seconds
- $43,680$ sensors in California
- $\implies$ 377 million data points per day

The *fundamental diagram* in traffic engineering shows the relationship between flow and density.



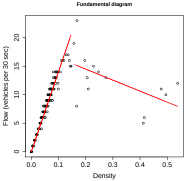
Each station has a fundamental diagram, which can be fit in parallel.



Source: Li, Zhang 2011

Using more data allows new types of analysis.

R expresses statistical computation well.




```
by(data=single_day, INDICES = station, FUN = my_fd)
```

- For a single day with 377 million observations this can be done on a single machine.
- A sensible way to run this in parallel is to `fork()` the process after reading in the data.
- So you write a bunch of code to do that :)

Small changes can require totally different computations.

If we compute on one year then this will exceed memory.

```
by(data=one_year, INDICES = station, FUN = my_fd)
```

A different model, such as least squares, may be able to process the data as a stream.

```
by(data=one_year, INDICES = station, FUN = my_lm_fd)
```

Access to the underlying database may allow us to run code directly inside the database.

```
SELECT station, my_fd(...) FROM data GROUP BY station
```

Table of Contents

This simple example shows how to write parallel code in R.

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))
```

We can express the mean as a weighted mean.

Suppose $n = n_j p$, where n_j is the chunk size and p is the number of chunks.

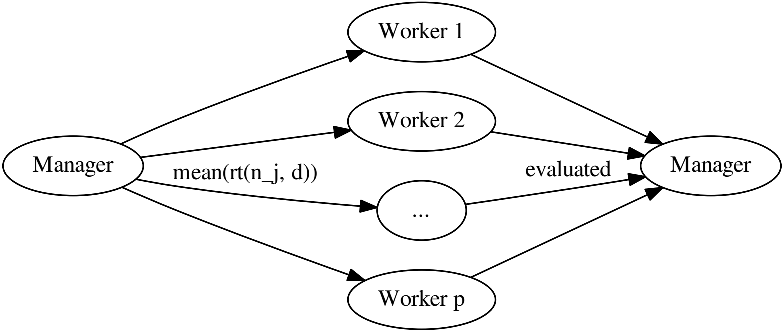
$$\bar{x} = \frac{1}{n} \sum_{j=1}^{p} \sum_{i=1}^{n_j} x_{ij} = \frac{1}{p} \sum_{j=1}^{p} \frac{1}{n_j} \sum_{i=1}^{n_j} x_{ij} = \frac{1}{p} \sum_{j=1}^{p} \bar{x}_{\cdot j} \tag{2}$$

The weighted mean can be directly translated into R code.

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

- While not parallel, this effectively removes the memory limits.
- How to choose n_j and p?

The same computation can be evaluated on many workers simultaneously.



Here is one way to parallelize this code.

```
library(parallel)
p = floor(detectCores(logical = FALSE) / 2)
n_j = n / p
cluster = makeCluster(p)

expr = substitute(mean(rt(n_j, d)),
list(d = d, n_j = n_j))

partial_means = unlist(
    clusterCall(cluster, eval, expr))

xbar = mean(partial_means)
```

We're considering a system that transforms expressions.

Input:

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

Output: (omitting boilerplate)

```
p = floor(detectCores(logical = FALSE) / 2)

expr = substitute(mean(rt(n_j, d)),
    list(d = d, n_j = n_j))

partial_means = clusterCall(cluster, eval, expr)
xbar = mean(partial_means)
```

This can be difficult because R is implemented in C.

```
> head(rt, 4)

1 function (n, df, ncp)
2 {
3     if (missing(ncp))
4         .Call(C_rt, n, df)
```

Options:

- Start from `replicate(p, mean(rt(n_j, d)))`
- Allow users to indicate how `rt` is vectorized
- Analyze the preprocessed C code
- Rewrite the C code in R, then analyze the R code

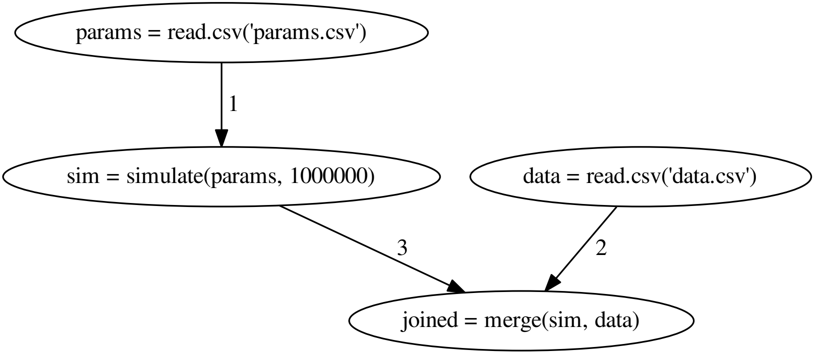
Table of Contents

CodeDepends is a tool for analyzing code as a data structure.

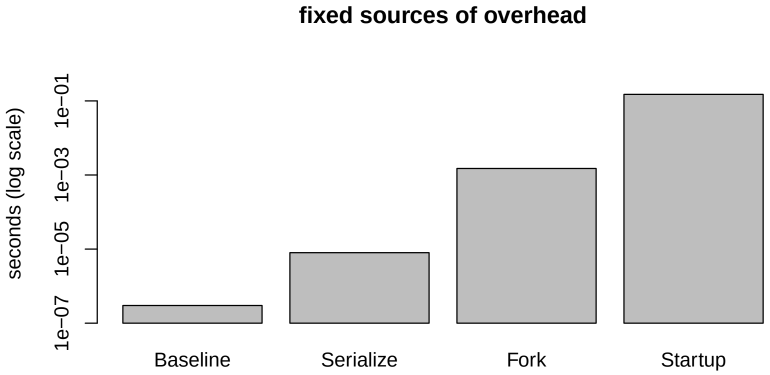
Consider this script:

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

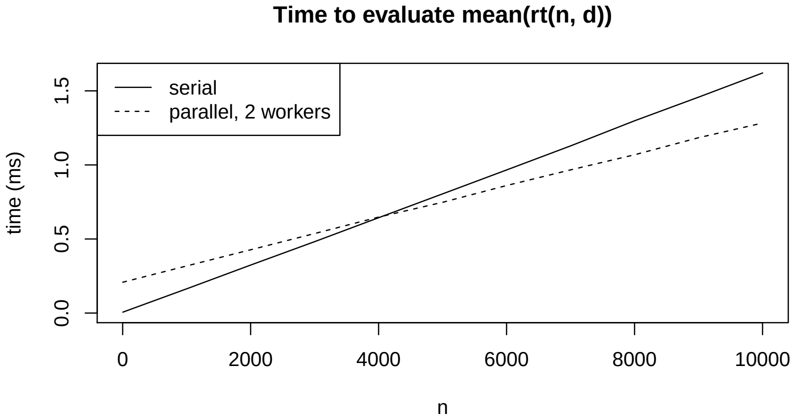
The expression graph represents the dependencies between expressions.



Is it worth it to go parallel?



Given an existing SNOW cluster with 2 workers we see benefits from parallelization when $n > 4000$.



Timings on a 3.4 GHz Intel i3 CPU

Factors to consider

Parameters

- number of processor cores to use
- size of each chunk
- which functions to combine in one processing step

Constraints

- number of cores available
- network bandwidth
- disk IO speed
- available memory

Idiomatic R already expresses computation in a natural parallel way through “apply” functions.

```
x = replicate(5, rnorm(n_j), simplify = FALSE)

partialmeans = lapply(x, mean)

by(data, INDICES = station, FUN = piecewise_rlm)
```

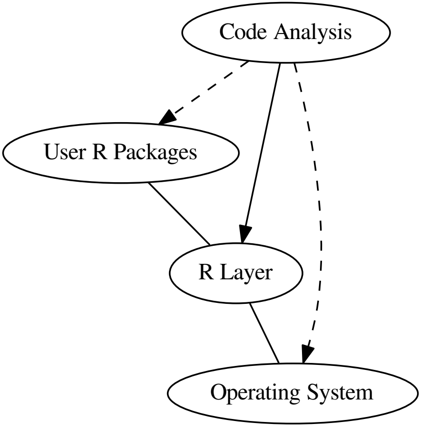
Also

```
apply, sapply, tapply, by, mapply, Map, vapply, outer
```

Layers mark ways for users to write parallel code for one platform.

- **User R Packages**: foreach, future, partools, ddR, biganalytics, RevoScaleR
- **R Layer**: SNOW, parallel, bigmemory, Rmpi, RCUDA
- **Operating System**: threads, processes, *NIX fork(), memory maps, network sockets, MPI

How can we transform R code into a lower layer?



Knowledge of the data allows us to generate more specialized code.

- File size
- Dimensions of table / matrix / array
- Column classes
- Randomized rows
- Sorted / grouped
- Possible values for factor
- Indexed
- Including sufficient statistics

Example: a data format that facilitates sampling.

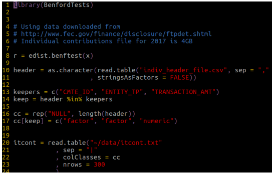
station, flow, occupancy, time

```
1         12        0.087     09:57:00
1         14        0.092     14:29:30

...

7         14        0.088     16:32:30
7         11        0.090     17:12:00
```

- ASCII fixed width format, c characters (bytes) per row
- sorted on station, then occupancy
- r rows per station
- $\implies$ new stations begin at byte $i \times c \times r$

Table of Contents

We propose using {Code, Data, Platform} to determine a parallel execution strategy.



```r
library(BenfordTests)


# Using data downloaded from
# http://www.fec.gov/finance/disclosure/ftpdet.shtml
# Individual contributions file for 2017 is 4GB

r = edist.benftest(x)

header = as.character(read.table("indiv_header_file.csv", sep = ","
                    , stringsAsFactors = FALSE))

keepers = c("CMTE_ID", "ENTITY_TP", "TRANSACTION_AMT")
keep = header %in% keepers

cc = rep("NULL", length(header))
cc[keep] = c("factor", "factor", "numeric")


itcont = read.table("~/data/itcont.txt"
            , sep = "|"
            , colClasses = cc
            , nrows = 300
            )
```

Related Work

- Related CS literature mostly focused on algorithmic applications and lower level languages
- Software such as Dask, Theano, Tensorflow require the user to build computation graphs
- Primarily focused on numeric arrays

The next step is to build a prototype of the system.

Specifically beginning with:

- *code*: Apply family of functions
- *data*: Files on disk exceeding main memory
- *platform*: Single server

Then test it with the traffic sensor data.

Next I'd like to extend this to platforms that store data

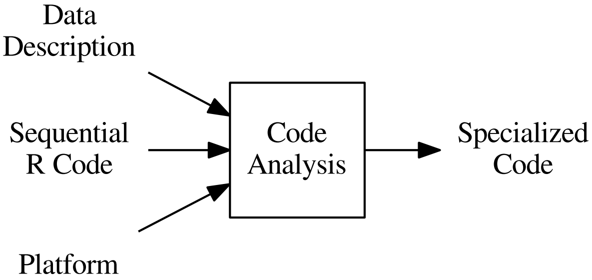
```
SELECT station, my_rlm(...) FROM data GROUP BY station
```

- Better data locality
- Many use cases
- Connection to compilation

Acknowledgements

- Faculty members
- Data Science Institute Affiliates and statistics students for applications and feedback
- Special thanks to Professors Duncan Temple Lang and Michael Zhang

Questions?



Data
Description
Sequential
R Code
Platform
Code
Analysis
Specialized
Code

Additional Slides

Follow up on the sensor example

The transformed program should:

1. Remove any observations it can
2. Reorganize files on disk based on station ID
3. Apply function to each station ID file

More Applications

- Benford test on election campaign contribution data
- Forest greenness satellite imagery (Andrew Latimer)
- Simulating spread of disease (Nistara Randhawa)

Preserving language semantics can be challenging.

For example, R's dynamic lookups

```
f = function() 0
g = function() f() + 1
f = function() 10
g()                       # Returns 11!
```

Compiled R code provides even more efficiency.

1. Parallelization will complement efforts to compile R
2. Compiled code potentially allows the use of shared memory threads
3. May follow the OpenCL kernel model

Last summer I worked on the Distributed Data Structures in R (DDR) project

- Relevant experience
- Idea: an abstraction layer for distributed and parallel data structures
- Created R lists and apply type functions to run on Spark

How do we detect if a function in R is vectorized, and in which arguments?

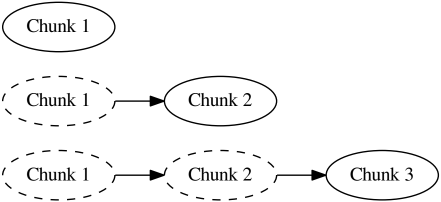
`rnorm()` is vectorized in the last two arguments, but semantically different for a vector in the first argument.

```
> rnorm(5, mean = c(1, 2), sd = c(2, 10, 200))
[1]   0.2134756  -4.1137221 256.0094734   0.4562226 -10.385537
```

It's all C Code.

```
> rnorm
function (n, mean = 0, sd = 1)
.Call(C_rnorm, n, mean, sd)
```

An iterator produces data on demand



- Most flexible of the above options
- Natural in pipeline parallel model
- Operate well with high performance IO libraries

- Unfamiliar to R programmers
- Not ideal if you need the whole data set

R in a database

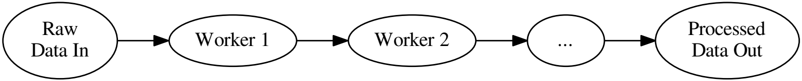
Suppose we want to call the vectorized function $f()$. If $f()$ is available both in R and the database then we have two options:

```
Option 1: Run f inside database, returning result

dbGetQuery(con,"SELECT f(x) FROM mydata;")

Option 2: First fetch x, then call f within R

f(dbGetQuery(con,"SELECT x FROM mydata;"))
```

By "programming on the language" we can modify existing code.

```
lapply_to_mclapply = function(expr)
{
    # Changes lapply to parallel::mclapply
    lapply = quote(parallel::mclapply)
    expr = force(expr)
    # Following Wickham's Advanced R book
    call = substitute(substitute(expr))
    eval(call)
}

> e1 = quote(xmeans <- lapply(x, mean))
> lapply_to_mclapply(e1)
xmeans <- parallel::mclapply(x, mean)
```

Pipeline parallelism is like a factory assembly line.




```
# Worker 1
x_chunk = rnorm(n_j)
serialize(x_chunk, worker2)


# Worker 2
x_chunk = unserialize(worker1)
partial_means[i] = mean(x_chunk)
```

Simple row based sampling misses the important areas of high density.