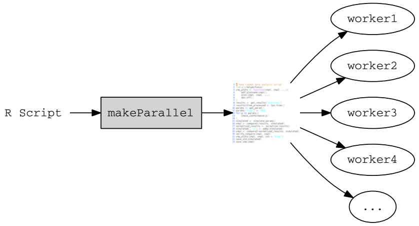
Automated Parallelization of R Code

Clark Fitzgerald
Duncan Temple Lang

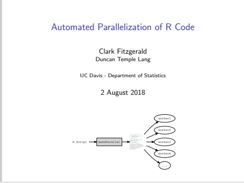
UC Davis - Department of Statistics

2 August 2018



2018-08-29

Automated Parallelization of R Code


Automated Parallelization of R Code

Clark Fitzgerald
Duncan Temple Lang

UC Davis - Department of Statistics

2 August 2018

Speakers start every 20 minutes, which means I will have a little less than 20 minutes. So 10 slides is probably plenty. Room capacity for classroom seating is 252 people.
What do I want to convey to the audience? The big picture.

- It's better not to have to change your code. More specialization = less portability.
- The goal of the makeParallel package is to programmatically adapt code to run most efficiently for the given data and available platform
- We can build on and apply existing CS research in scheduling algorithms. R's functional programming model allows us to do more.
- The Hive / Highway data analysis might help motivate it.

What should I avoid talking about?

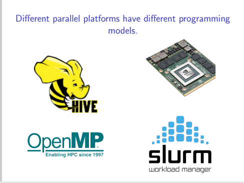
- Implementation details. I don't think people will care about the technicalities around FIFO's.

Different parallel platforms have different programming models.

HIVE

OpenMP®
Enabling HPC since 1997

slurm
workload manager


Different parallel platforms have different programming models.

Many possible different parallel machines, want to rapidly deploy These are what I mean by platforms- combinations of hardware and software. Emphasize- code is totally different. That's why we need to automate it. If we want to use these tools we *must* use their programming models. Otherwise we lose out on all the benefits.

So the code must change.

Changing *our* code to make it parallel distracts from the higher level data analysis.

- Subset
- Plot
- Fit Model
- Simulate
- etc.

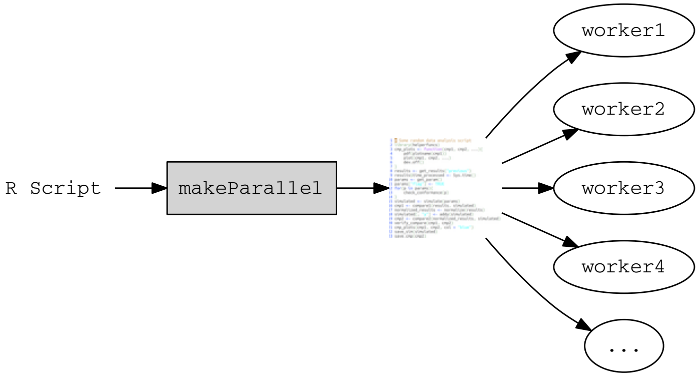
└─Changing *our* code to make it parallel distracts from the higher level data analysis.

Changing *our* code to make it parallel distracts from the higher level data analysis.

- Subset
- Plot
- Fit Model
- Simulate
- etc.

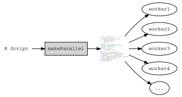
A strength of R is that it's high level, and it works well for data analysis.

The idea is to have a *system* makes the code parallel so that *humans* don't have to.



└─The idea is to have a *system* makes the code parallel so that *humans* don't have to.

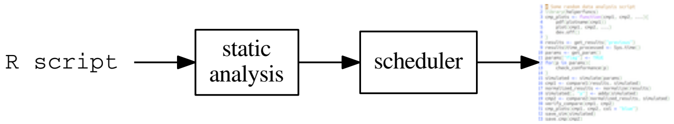
The idea is to have a *system* makes the code parallel so that *humans* don't have to.



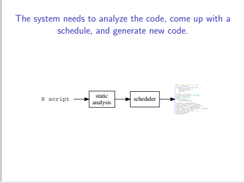
It's better to not have to change your code. More specialization equates to less portability.
"Premature optimization is the root of all evil." -Don Knuth
'makeParallel' here is a 'black box' that represents my ideal machine.
We feed the machine some regular R code. Nothing fancy, suppose we just use the base package.
Then the machine generates a plan, or ideally executable parallel code.
The code doesn't necessarily have to be R code- it just has to do what the R script expressed.
This is akin to compilation.

I'm working on building such a machine.

The system needs to analyze the code, come up with a schedule, and generate new code.



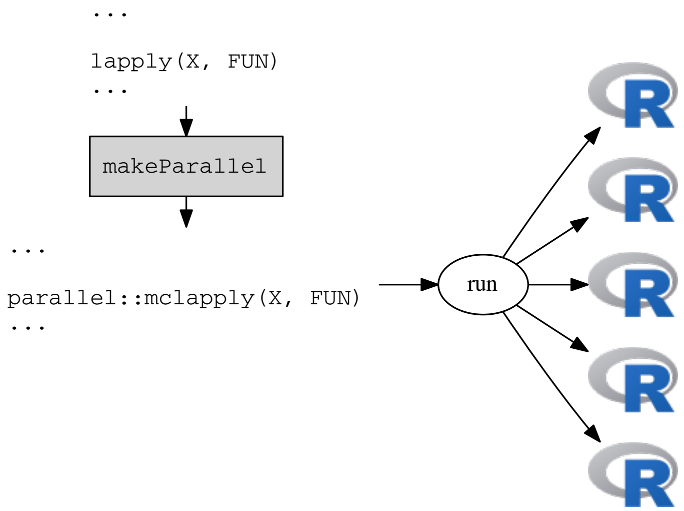
```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
    pdf(plotname(cmp1))
    plot(cmp1, cmp2, ...)
    dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
    check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

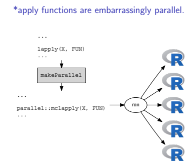

The system needs to analyze the code, come up with a schedule, and generate new code.

- statically analyze the code to understand the semantics and dependency structure.
- Use the results of that analysis to come up with a schedule.
- Create executable code

We're talking about altering the code itself before it even runs. This differs from defining objects with methods and relying on dispatch. Although method dispatch could be very helpful when it comes to generating the code.

*apply functions are embarrassingly parallel.




*apply functions are embarrassingly parallel.

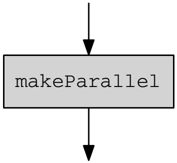
```
...
lapply(X, FUN)
...
```

```
...
parallel::mclapply(X, FUN)
...
```

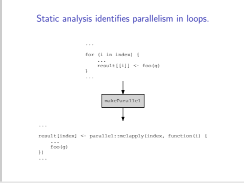
This is a familiar example of how we might parallelize code.
The ... mean that this is one statement in a larger script, so we're analyzing this in context.
Serial code goes in, parallel code comes out.
Even this is not just a matter of trivial substitution, ie. redefining lapply.
Hence a need exists for something like this.

Static analysis identifies parallelism in loops.

```
...

for (i in index) {
    ...
    result[[i]] <- foo(g)
}
...
```



```
...

result[index] <- parallel::mclapply(index, function(i) {
    ...
    foo(g)
})
...
```



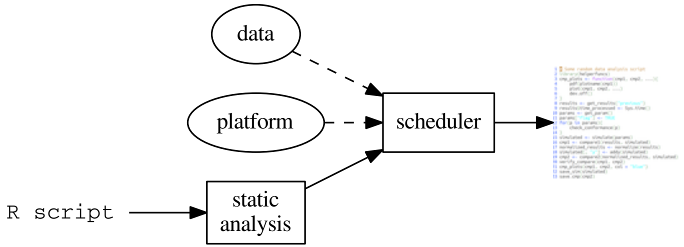
Static analysis identifies parallelism in loops.

```
...

for (i in index) {
    ...
    result[[i]] <- foo(g)
}
...
```

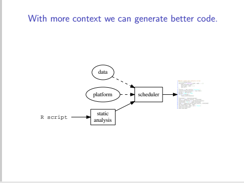
```
...

result[index] <- parallel::mclapply(index, function(i) {
    ...
    foo(g)
})
...
```

To find out if a for loop can be run in parallel we need to check for true dependency in the loop iterations.
You might look at this and say it should just be an 'lapply' or a 'replicate' already. I agree. But this comes from real user's code, and the body of the loop had many more lines.
My philosophy is that it's more useful to be able to statically analyze R code in the wild, as it comes to me, rather than how it "should be".
makeParallel can do this now.

With more context we can generate better code.

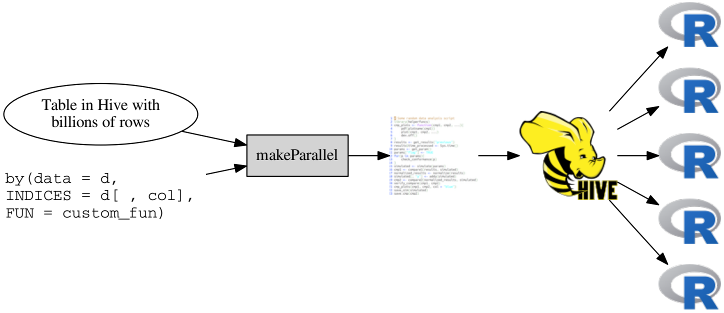


```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
    pdf(plotname(cmp1))
    plot(cmp1, cmp2, ...)
    dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
    check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```


With more context we can generate better code.

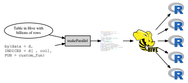
By platform I'm referring to a set of hardware and software.
To schedule statements we need to know how long they will take.
Given the data or object sizes we know how long the transfers will take, and this helps with scheduling because we want to avoid the expense of transferring large objects.
Ideally we would infer these things, or we can run the code serially and measure them.
I think it's reasonable to make some of them available, because the standard advice for improving code performance is to first profile it. This requires running at least some of it. As we run code we can measure whatever we like.

We can generate code that will run on large systems.



```
by(data = d,
INDICES = d[ , col],
FUN = custom_fun)
```

We can generate code that will run on large systems.

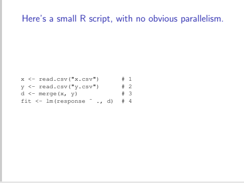


```
by(data = d,
INDICES = d[ , col],
FUN = custom_fun)
```

We've had R to SQL translators for many years. This is more than that, the idea is to run *arbitrary* R code, not just the subset that translates easily to SQL.
We've had R / Hadoop interfaces for many years. The problem is the same as every other parallel model- you have to write code to work with that model, rather than bringing the code you have.
Overhead from data motion can be very expensive. Relates to Michael Lawrence's talk. We want to move the code to the data.
This approach scales well. Billions of observations, terabytes of data.
'Group By' operations map directly to Hive / Hadoop. I combined R and Hive to process 3 billion rows, 300 GB size data in 12 minutes on a small 4 node cluster.
If you analyze large tables on a regular basis then they're probably stored in some larger data warehousing system.
This is not fully implemented in makeParallel. But the possibilities are rich.
It's a fair amount of technical boilerplate code to get this to work. But

Here's a small R script, with no obvious parallelism.

```
x <- read.csv("x.csv")        # 1
y <- read.csv("y.csv")        # 2
d <- merge(x, y)              # 3
fit <- lm(response ~ ., d)   # 4
```

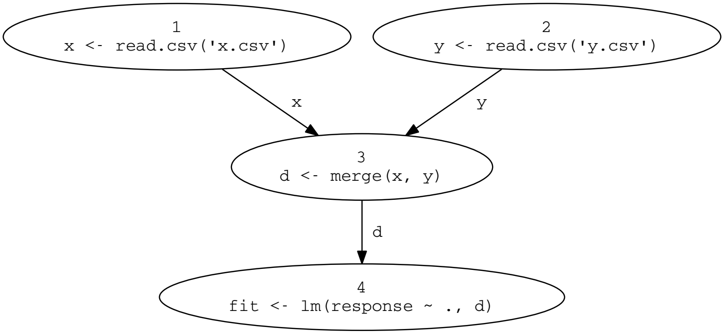


Here's a small R script, with no obvious parallelism.

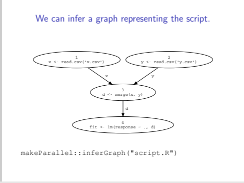
```
x <- read.csv("x.csv")      # 1
y <- read.csv("y.csv")      # 2
d <- merge(x, y)            # 3
fit <- lm(response ~ ., d)  # 4
```

Each of these functions can be parallelized, but that's not the point here– I'm just using them because they're familiar.

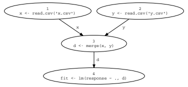
We can infer a graph representing the script.



```
makeParallel::inferGraph("script.R")
```



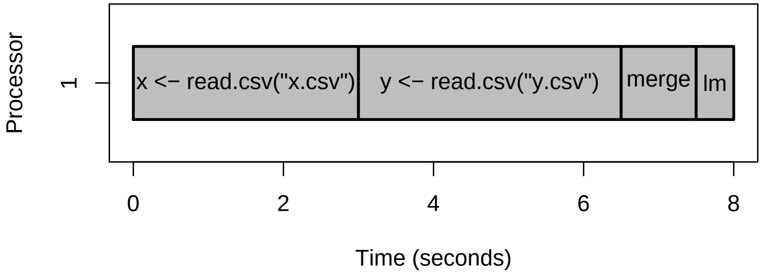
We can infer a graph representing the script.



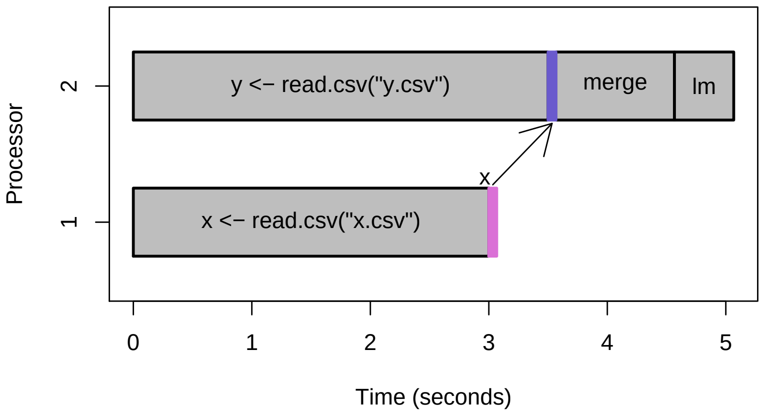
```
makeParallel::inferGraph("script.R")
```

Nodes represent statements.
Edges represent dependencies between statements.
We add edges from the statements that define variables to the statements that use the variables. So there's an edge from the statement `x <- read.csv("x.csv")` to `d <- merge(x, y)`.
Looking at this graph we see that it's possible to run two statements at the same time.

This is how R normally spends its time running this script.



The sibling nodes in the graph can run at the same time.

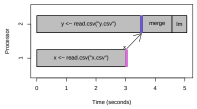


2018-08-29

Automated Parallelization of R Code
└─Task Parallel

└─The sibling nodes in the graph can run at the same time.

The sibling nodes in the graph can run at the same time.



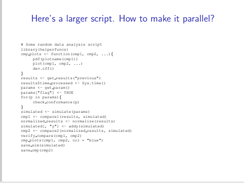
Explain this slide thoroughly.

The grey blocks represent statement execution times.

The colored blocks represent transfer times.

Here's a larger script. How to make it parallel?

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
     pdf(plotname(cmp1))
     plot(cmp1, cmp2, ...)
     dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
     check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```


Here's a larger script. How to make it parallel?

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
     pdf(plotname(cmp1))
     plot(cmp1, cmp2, ...)
     dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
     check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

Wall of code.
This is just some typical data analysis script. It's meant to be just a wall of code, so don't look at it.
We don't know what any of the functions do without examining their code.
There are no calls to apply style functions, so we can't get an easy win.
Although maybe we can parallelize the for loop.

There are two *independent* sequences of statements that can run at the same time.

Worker A

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
     pdf(plotname(cmp1))
     plot(cmp1, cmp2, ...)
     dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
     check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results,
simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

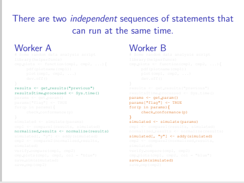
Worker B

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
     pdf(plotname(cmp1))
     plot(cmp1, cmp2, ...)
     dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
     check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results,
simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

2018-08-29

Automated Parallelization of R Code
└─Task Parallel

└─There are two *independent* sequences of statements that can run at the same time.


There are two *independent* sequences of statements that can run at the same time.

Worker A

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
    pdf(plotname(cmp1))
    plot(cmp1, cmp2, ...)
    dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
    check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results,
simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

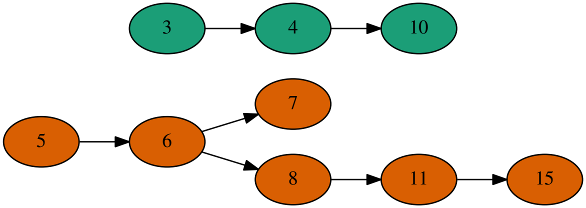
Worker B

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
    pdf(plotname(cmp1))
    plot(cmp1, cmp2, ...)
    dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
    check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results,
simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

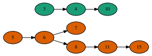
If the script spends most of its time executing these two sequences of statements *and* the two sequences take about the same amount of time then we can potentially get close to a 2x speedup by running them in parallel.

How do we detect these patterns more generally?

This graph represents the two groups of independent statements.



This graph represents the two groups of independent statements.

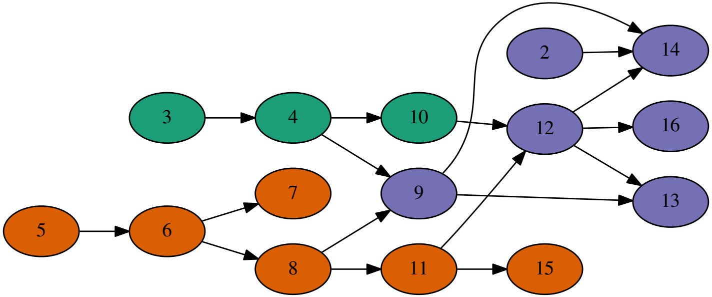


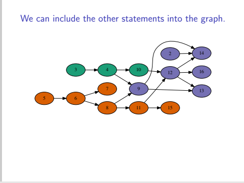
We add an edge from 3 to 4 because statement 3 defined the variable `results`, and 4 updated `results` as follows: `results$time_processed <- Sys.time()`.

There are other statements besides the two independent groups.

```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
     pdf(plotname(cmp1))
     plot(cmp1, cmp2, ...)
     dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
     check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```

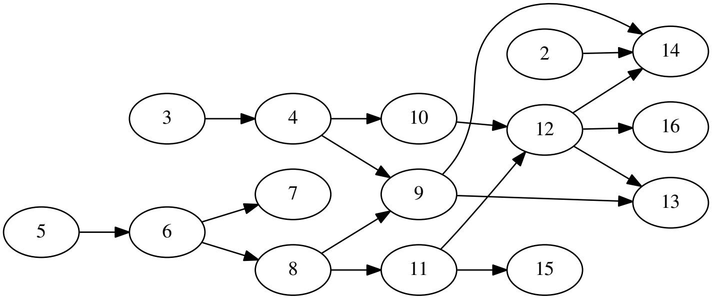
We can include the other statements into the graph.




We can include the other statements into the graph.

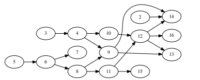
This is the full graph.

The (NP hard) problem is how to schedule the statements on multiple processors given the constraints of the graph.



```
makeParallel::inferGraph("script.R")
```

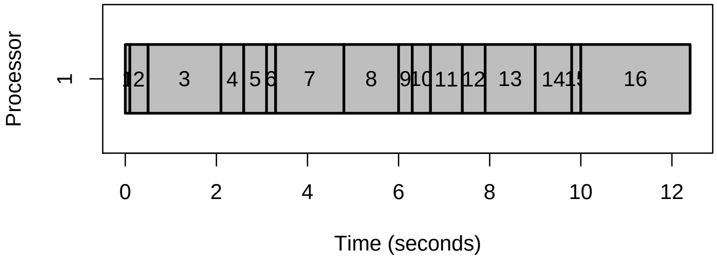
The (NP hard) problem is how to schedule the statements on multiple processors given the constraints of the graph.



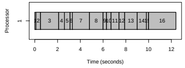
`makeParallel::inferGraph("script.R")`

This is called the task graph or dependency graph.
makeParallel can do this.
Of course, this is the object that I started with when I made these slides.
I eyeballed it and said that these two sequences can run in parallel.
This graph shows the essential ordering for the computation, so we can start applying scheduling algorithms with k workers.

This shows how the serial program spends its time.

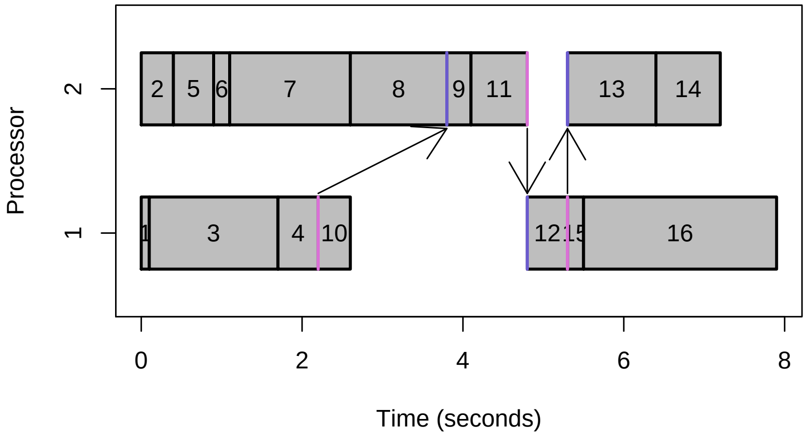


This shows how the serial program spends its time.

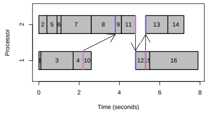


Numbered blocks represent the time to execute each statement.
This shows a valid schedule. It's how R, and most other programming languages work- execute each statement in order, on the same processor.
The grey blocks go past 12 seconds, so this takes more than 12 seconds to run.

Task scheduling is much richer than embarrassingly parallel problems.



Task scheduling is much richer than embarrassingly parallel problems.

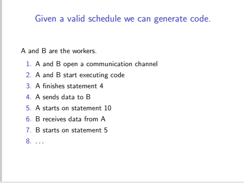


The arrows show the processors communicating with each other, passing data back and forth.
First of all we see a reduction in the total execution time from more than 12 seconds to less than 8 seconds. With more processors we might be able to do better.
We can tell this particular scheduler does a reasonable job for this particular problem because the processors mostly stay busy.
It isn't necessarily optimal- the problem is NP hard.
CS researchers have come up with many scheduling algorithm. So this is a place that we can stand on the shoulders of giants.
makeParallel generated this schedule.

Given a valid schedule we can generate code.

A and B are the workers.

1. A and B open a communication channel
2. A and B start executing code
3. A finishes statement 4
4. A sends data to B
5. A starts on statement 10
6. B receives data from A
7. B starts on statement 5
8. . . .


Given a valid schedule we can generate code.

A and B are the workers.

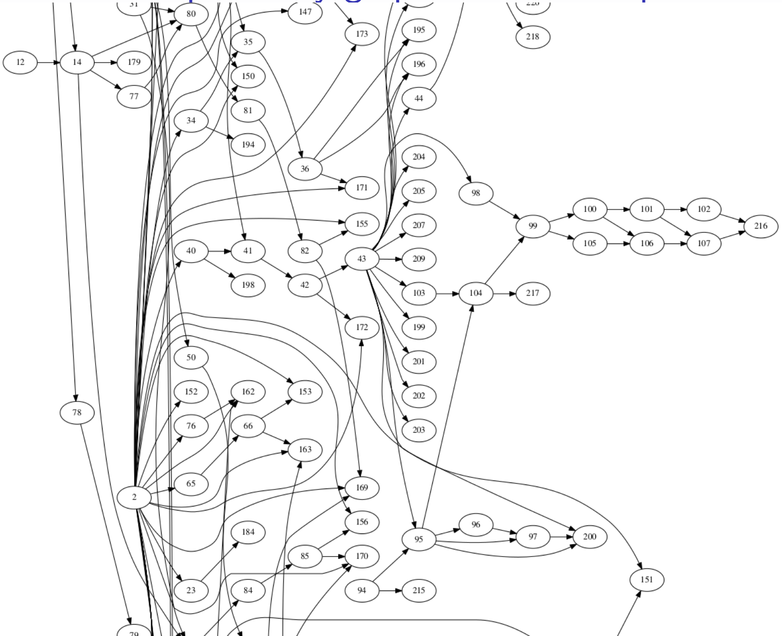
1. A and B open a communication channel
2. A and B start executing code
3. A finishes statement 4
4. A sends data to B
5. A starts on statement 10
6. B receives data from A
7. B starts on statement 5
8. . . .

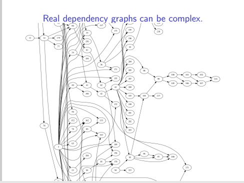
The system can generate code that manages communication and synchronization.

I would not attempt to manually write code like this.

Speaking from experience, it's painful, unintuitive, and error prone.

Real dependency graphs can be complex.

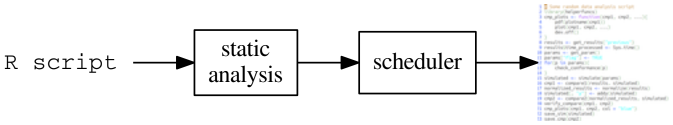




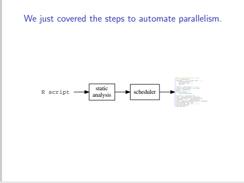
Real dependency graphs can be complex.

This is a subset of a graph for a real computation in R that takes a week to run.
The interesting thing about this particular one is that it's much more wide than it is deep - nodes have many siblings. Also there are many leaf nodes. This is all good for task parallelism, and difficult for map parallelism.
It's also bad because it probably means this program just repeated code where it should have been using something like an `lapply`. So then another approach would be to replace the repetitive code with `lapply` in a preprocessing step.

We just covered the steps to automate parallelism.



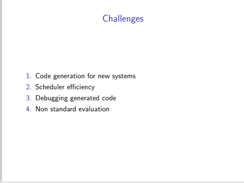
```
# Some random data analysis script
library(helperfuncs)
cmp_plots <- function(cmp1, cmp2, ...){
    pdf(plotname(cmp1))
    plot(cmp1, cmp2, ...)
    dev.off()
}
results <- get_results("previous")
results$time_processed <- Sys.time()
params <- get_param()
params["flag"] <- TRUE
for(p in params){
    check_conformance(p)
}
simulated <- simulate(params)
cmp1 <- compare1(results, simulated)
normalized_results <- normalize(results)
simulated[, "y"] <- addy(simulated)
cmp2 <- compare2(normalized_results, simulated)
verify_compare(cmp1, cmp2)
cmp_plots(cmp1, cmp2, col = "blue")
save_sim(simulated)
save_cmp(cmp2)
```


We just covered the steps to automate parallelism.

Static analysis is inferring the graph, hopefully later we can augment it with time / size inference
Scheduling is coming up with the plan, what to run where
Finally there's generating the code, whether that's putting `parallel::mclapply` in the right place or the full blown task parallel program.

Challenges

1. Code generation for new systems
2. Scheduler efficiency
3. Debugging generated code
4. Non standard evaluation


Challenges

1. Code generation for new systems
2. Scheduler efficiency
3. Debugging generated code
4. Non standard evaluation

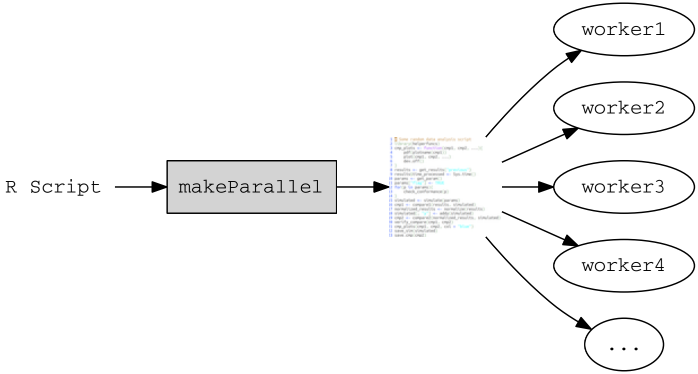
Writing a new code generator is a pretty technical job requiring deep knowledge of R, scheduling algorithms, and low level system concepts.
Standard scheduling algorithms don't necessarily work that well for R and data analysis. We need to tailor them, and this is something that I've been thinking about.
Things will go wrong with the generated code once we use this in more serious ways. It's going to be very difficult to debug some of the low lying stuff with remote workers and sockets. We'll probably need some kind of logging system.
Non standard evaluation complicates static analysis because it changes the semantics of the language, ie. we can use symbols that don't correspond to objects and change scoping. It seems that every NSE model will require its own custom analysis- so the formula in lm() will be different than dplyr, which is different from data.table, etc. I've left hooks in to write this, but it's not clear if things will work together well.

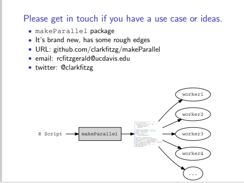
Prior Work

- Decades of CS literature - ‘Task Scheduling’ (Sinnen)
- Static Code Analysis in R
  - Byte Code Compiler (Tierney)
  - `rstatic` (Ulle), `RLLVMCompile` (Temple Lang)
  - `CodeDepends`, `CodeAnalysis` (Temple Lang et al.)
  - NIMBLE (de Valpine et al.)
- Task parallelism in R
  - `pqR` pretty quick R (Neal)
  - `future` (Bengtsson)
  - `drake` (Landau)

Please get in touch if you have a use case or ideas.

- `makeParallel` package
- It's brand new, has some rough edges
- URL: github.com/clarkfitzg/makeParallel
- email: rcfitzgerald@ucdavis.edu
- twitter: @clarkfitzg





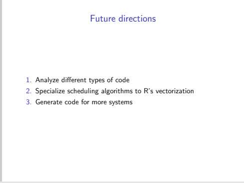
Please get in touch if you have a use case or ideas.

- `makeParallel` package
- It's brand new, has some rough edges
- URL: github.com/clarkfitzg/makeParallel
- email: rcfitzgerald@ucdavis.edu
- twitter: @clarkfitzg

We use parallelism to make code faster.
In other words it's a technical implementation detail.
When we write code to use a particular parallel model it becomes less portable.
We can think of programming languages / models as a continuum between high and low level. For example, Assembler is lower level than C, and C is lower level than R. For data analysis we want the code to be high level so that we can recognize the correspondence between the data analysis task.
Parallel programming is full of details at the low level. Packages have evolved to give higher level abstractions so that we don't have to worry about these details. This is good, because the low level details have nothing to do with the actual analysis.
Why it's not trivial substitution- nesting, extracting loop invariants, libraries, globals, checking side effects
We can also parallelize vectorized function in a similar way. So we **can** parallelize many statements in R. But what we really don't know a priori

Future directions

1. Analyze different types of code
2. Specialize scheduling algorithms to R's vectorization
3. Generate code for more systems



Future directions

1. Analyze different types of code
2. Specialize scheduling algorithms to R's vectorization
3. Generate code for more systems

I think there's big opportunity for integration. I.e. separate the code into parts that can run inside the underlying data store, and parts that actually require R.