1 TODO

1.1 Questions

To discuss with Duncan, most important first:

1. Wednesday, April 18th is the last day to edit my JSM abstract. Here's what I currently have:

   Title: Automatic Parallelization of R Code

   Abstract: Conventional systems for parallel programming require users to modify existing serial code to take full advantage of a platform's computational capabilities. In this presentation we consider automated code analysis methods to detect the potential for parallel execution in R code. A system can then automatically transform the serial version of the code into an equivalent parallel version. We consider a motivating case study analyzing hundreds of gigabytes of California's highway traffic sensor data.

   The problem with the current abstract is that my research direction is focusing on the task graph, and the Hive / PeMS use case doesn't really use the task graph. So it may be challenging to integrate the two in a talk.

2. I listed all the ways I can think to change state in Section 5

   (a) What did I miss?

   (b) Let's rank these by importance and figure out which to do first.

   (c) Which can be collapsed into the "same thing"?

3. I could release the package 'autoparallel' on CRAN before my talk with the basic low level code for running group by type computations on Hive. Should I do this? Pros: Gives audience something they can possibly use. May help me get a job when the time comes. Cons: It doesn't really use any code analysis, so it's not too relevant. Users may ask for more features / bug fixes, which will take time.

4. Should we initially focus on static or dynamic scheduling for actually executing the script in parallel? See Section 7.

5. Should we mention pipeline parallelism in the context of code analysis? Ie. some code of the form `f(g(h(x)))` can run sequentially but still in parallel the same way that we would do `cat x | h | g | f` in the shell. The iotools paper has a nice example of it.

   No. To do this we need `h, g, f` to be vectorized possibly with `f` as a reduce type function capable of handling streams. If this is the case then we may as well just compose the functions into one and use map reduce.

6. Duncan mentioned wanting code analysis to figure out when plots create a new plotting frame. One reason I can think this would be useful is because then we can potentially

remove code before this, if it's not using multiple frames. Are there other reasons we care about new plot frames?

1.2 Plan

To help focus my efforts I'm going to lay out all the tasks that I could possibly work on that relate to this topic, and then order them by priority.

Task Graph Details This is the data structure that I'm thinking about to hold all the information on parallelism, so it needs to be fleshed out.

- What exactly does it contain?
- What are the mechanisms for extending it?

Dynamic Scheduler Literature Quick review of the CS literature about dynamic task scheduling.

Implement Scheduler Could be static or dynamic.

Minimizing Memory Use We could use the task graph to generate a script with sequential execution that removes variables after they are no longer needed. A basic version wouldn't be too difficult, and it illustrates the flexibility of the task graph. A more advanced version could potentially check which elements of large objects are accessed later and save only those, ie. only `head(x)` for a large object `x`.

Slides for JSM talk Draft due to Norm on May 17th. Completed slides can be uploaded in mid July.

Implement Task Graphs I've implemented the Use-def edges. The next step is probably process edges.

ADMM loop use case How does the expression graph work for modeling the ADMM method that repeatedly visits the same data?

Interactive Visualization of the task graph that shows expressions for the nodes and relationships for the edges.

Automatic Parallelization of R Code

Clark Fitzgerald, Duncan Temple Lang

April 17, 2018

2 Introduction

The main focus of this research is the automatic inference of the task graph for the R language.

2.1 Text Sections

Motivation describes why an R user might be interested in parallel code generation. It describes how the ideas can be useful in a typical R data analysis workflow. It also documents and motivates the design of the software.

Changing State briefly lists all the ways that expressions can change state in a way that affects future output or computations. It's intended to be comprehensive. We may not solve all of them, but we at least want to be aware of them. Then we'll prioritize them.

Challenges describes more fully all the different challenges and special cases that can arise from evaluating different R expressions in parallel. It's an extension of Changing State.

Background provides an overview of related projects to execute in parallel. Some are general systems and some are specific to R.

Task Graph defines and describes the task graph.

Scheduling discusses possible schemes for actually running expressions in parallel once we've inferred the task graph. It compares dynamic with static scheduling options.

2.2 Software Steps

We have a particular set of steps in mind to convert sequential R code into parallel.

- The user provides a working script.

  But how do they debug it without running it?

- The **static analyzer** infers what it can from each expression individually. Currently we rely on the CodeDepends package for this [Temple Lang et al., 2017].
- The static analyzer infers what it can about the *relationships between expressions*. This is the whole initial focus of this project. We can potentially integrate / combine it with the rstatic package. So the static analyzer builds the task graph.
- A **scheduler** takes the task graph and actually runs the expressions, possibly in parallel. Other appropriate names for the scheduler might be evaluator or executor. Schedulers are modular in the sense that we can plug in different scheduling schemes and platforms. We can potentially use a project like the drake package as a scheduler [Landau, 2018].

The distinction between the static analyzer and the scheduler is important. The static analyzer is responsible for detecting and describing all the dependencies in the code, while the scheduler must honor all of these dependencies. We can use the results of the static analysis for other things besides parallel execution, ie. as a tool for the user to understand the nature of the computations that are happening. The scheduler should be the one that actually generates code, because then it can better tailor it to the platform.

3 Motivation

3.1 User Narrative

Kate is a hypothetical R user analyzing some data. She's working along, digging into her data, making exploratory plots, and recording the steps that she would like to save in a script called "analysis.R".

She runs a chunk of her code and waits for it to finish. After a few minutes it's still running with no signs of being finished, so she hits CTRL-C to interrupt the process and starts thinking about how to speed it up. She knows that she can get more speed by modifying her code and using R's recommended "parallel" package to take advantage of parallelism features offered by her Linux operating system. But this is purely a performance optimization, and she hesitates because her collaborator will need to run the code on a Windows machine, so she doesn't want to tailor her code for this just yet. She also hopes to run this same analysis on her department cluster, so she'll have to write another version specialized for this platform.

Kate is focused on expressing a particular data analysis task in code. Specializations and modifications for performance distract her from her focus [Matloff, 2015]. Wouldn't it be nice if she had a system that could take her code and tailor it to the particulars of the data and platform that she was working on? Then she could just take this piece of code and call `parallelize(code)`. This passes the technical problem of transforming serial source code to parallel source code off to another system. The system guarantees that it will do semantically the same operations, just in a different way and hopefully faster.

Once Kate has a script that she's ready to share with her collaborator on the Windows machine she (or her collaborator) can call:

```
parallelize(file = "Analysis.R"
    , saveas = "AnalysisWindowsCollaborator.R"
    , platform = "Windows"
    , ncores = 4
    )
```

With no effort on her part she has produced a version that runs efficiently on her collaborator's Windows machine. In the same way, once she decides to run a larger version on the department cluster she can take her original script and run a similar command with say `platform = "SLURM"`.

This separates the software into a naive version, which is much like a specification, from a version specialized for higher performance through parallel processing. It promotes simple, reusable code. It allows Kate to be more productive by focusing on the data analysis and less on the system details. Of course this is the goal of any abstraction in software. The difference here is that we are not introducing a new abstraction. Rather we are inferring and building an abstraction out of existing code. This leaves Kate free to use any R code rather than having to express her computations using one particular API or framework.

One of the most appealing aspects of this approach is that she can potentially target new parallel systems as they mature and become more widely available, and she isn't required to know the intricacies of these systems to use them. For example, she might want to use her Graphical Processing Unit (GPU) to do some of the computations. If she can use the same code as in her normal single threaded R environment this is a huge win.

3.2 Code Analysis

Kate may also like ask other questions of the code, for example:

1. About how long will it take? If it's a couple days then maybe she should move it to a server right now.
2. Where are the bottlenecks? Then she can rewrite only these parts if she needs more performance.
3. How many processors should she use for this ideally? Maybe she can't get any benefit from more than 3 processors. Then she might save herself the trouble of moving the data and just run it on her laptop.
4. How much memory is necessary for these computations? Will more memory help?

Code analysis can help us to answer some of these questions.

3.3 How we'll do it

Now we explain at a high level how we'll go about analyzing the code and determining what kind of parallelism is possible. We'll also hint at some of the challenges.

The easiest pieces of code to parallelize are expressions which are embarrassingly parallel, such as R's apply family of functions, ie. `lapply, sapply`. But even these are not guaranteed to be safe, because they have an implicit order. For example, we may be able to add elements to a plot using `sapply(sample(1:n), plot_one)`. To be consistent with base R's "canvas" graphics model this `sapply` must run in order. Figure 1 shows two different plots that come from executing the same `sapply` in a different order. The other problem is that there are certain operations such as plotting that we don't want to do in parallel.

We can also potentially go beyond embarrassingly parallel problems through task parallelism. This means executing multiple different statements simultaneously.

3.4 Task Graph

The execution model for R is simple. The interpreter evaluates each expression in the order that it sees it. Run expression 1, then expression 2, ... until there's no more input.

Figure 1: Running apply style code in a different order will produce different results if there is an implicit dependency.

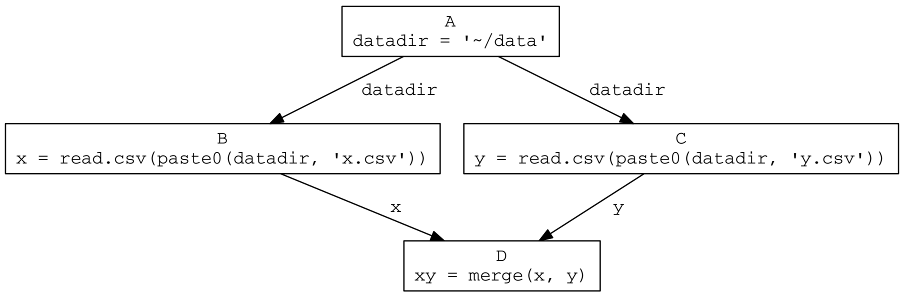
A
datadir = '~/data'
datadir
datadir
B
x = read.csv(paste0(datadir, 'x.csv'))
C
y = read.csv(paste0(datadir, 'y.csv'))
x
y
D
xy = merge(x, y)

Figure 2: Task graph for Listing 1. Nodes correspond to top level expressions.

If an expression does not depend on the result of a previous expression, then we may be able to run that code in parallel. Listing 1 shows a simple example. The first line defines `datadir`, and then 2 and 3 can run simultaneously. The last line must wait until lines 2 and 3 have both completed and defined the variables `x` and `y`.

Listing 1: Simple script

```
datadir = '~/data'                      # A
x = read.csv(paste0(datadir, 'x.csv'))  # B
y = read.csv(paste0(datadir, 'y.csv'))  # C
xy = merge(x, y)                        # D
```

Figure 2 plots the dependency structure between lines in Listing 1 using a directed graph. The nodes of the graph are the top level R expressions, and the edges come from the definitions and uses of the variables. This graph captures the true constraints on the evaluation order of the expressions. Once we have the complete graph we can run some subgraphs together in parallel. A real R script will have a much richer graph, possibly with many different branches where child graphs can be run in parallel.

3.5 Challenges

To do this efficiently we need to think about minimizing data motion.

Consider the different ways that we could execute Listing 1 given 2 existing R processes which we'll refer to here as workers. Start with worker 1 executing block A, the first line. At this point `datadir` exists on worker 1, but not on worker 2. Then we have three options to prepare workers 1 and 2 to execute blocks B, C simultaneously:

1. Run A on worker 2.
2. Transfer `datadir` to worker 2.
3. Kill worker 2 and fork worker 1 to create a new worker 2.

In the first two cases we then have to decide which of workers 1 and 2 are going to execute B, C. If the expressions used more existing variables this could matter. We have to consider these kind of decisions more generally every time that we execute an expression in parallel.

The point is that even in this simple example of task parallelism there are many issues to consider, which is why we would like a system to help us with it.

4 Task Graph

The **task graph** is the data structure that shows task parallelism in the expressions. Each node is an expression and each edge represents relationships between expressions.

The purpose of the task graph is to represent the general structure in the code. It respects the semantics of the R language and the ordering of the statements in the original script. It's an intermediate and augmented representation of a script that can more readily be mapped into an efficient execution plan on different platforms.

4.1 Task

The task graph should provide sufficient information for a scheduler to execute the program while preserving the semantics. At a minimum the scheduler must be capable of taking a newly created R process and bringing it to a state so that it can run a new expression in the program.

The following are some basic questions the scheduler will ask of the task graph, and we should have little trouble inferring them from the code:

- Which expressions are ready to execute given the current state of evaluation
- Which variables should be available
- Which expressions must happen in the same process, ie. plotting calls to one device
- Which libraries should be loaded

The task graph should know more if possible:

- How long it takes to execute each expression
- What the side effects are, and whether it's safe to execute the same expression again without changing state
- The size of each variable
- Which expressions can themselves be run in parallel, ie. `lapply(x, f)`

Besides the information about the code the scheduler must also know the characterisitics of the system, ie. number of processors, RAM, cost of serialization between nodes. It's no problem to get this.

4.2 What task graph should provide

What do we want to know about the expressions? In general we would like to know all the ways that they use or update global state. This is in Section 5. We can build directly off the CodeDepends package for this.

Table 1: Expression dependency graph

From	**To**	**Type**	**Weight**	**Value**
1	3	def-use	3	varname: x, size: 300
2	3	process	0	plotting

What information should the edges contain? They need to be able to convey all the relevant pieces of information based on the change of state. Let's use types to categorize the edges. Below are two basic types of edges that map directly to implementation by the scheduler. This simplicity is helpful because it's less complexity for the scheduler to handle. This simplicity also limits us in the sense that it assumes R's normal execution model. It also may make some things more difficult, for example, algebraic reductions of expressions.

definition - use I expect most edges come from the definition use chains of variables, ie. there is an edge $i \to j$ of type 'use-def' if expression i defines `x` and expression j uses `x`. This relationship is also called flow dependency, true dependency, or read-after-write (RAW). The scheduler must make this version of the variable `x` available before it can evaluate expression j.

The semantics described above mean that we do not need to add edges for write-after-read (WAR) or write-after-write (WAW) dependencies. Hence we have a task graph rather than a dependency graph as distinguished in [Sinnen, 2007].

same process an edge of type 'process' from $i \to j$ means expression i must run before expression j in the same process. Example: plotting. Note that R's conventional sequential execution model can be understood as a set of process edges, $i \to i + 1$ for each statement i in the program. If the code analysis fails for ambiguous or hard to analyze blocks of code we can also fall back on this same process constraint for some blocks of code. This means if we don't know what to do, then just execute the statements in order in the same process.

Are there more basic types? Not off the top of my head.

We can extend this data model to handle different types of relationships / constraints between expressions by adding new edge types. For example, for code that uses the DBI package to interact with a database we might define and add a 'database' type of edge, along with special behavior.

One physical way to represent the graph is with a data frame with each row representing an edge, as shown in Table 1. This is extensible because we can add new types of edges that may be package specific. We can put anything we like in the Value column. We need to represent how large each object is, because then we can use this along with the network speeds to determine how long it will take to transfer that object between processes.

With the library calls, if we have statically resolved the packages where the symbols come from then we may as well save this in the expressions, because it may be useful for other tasks as well.

One way to do this is to maintain the search path along with a list of defined global variables up to that point in the script.

Today (12 Apr 2018) I was not properly distinguishing definitions from uses. I was thinking that this statement as only redefining `x`: `x$a = 10`. But `x` is also an input, as CodeDepends correctly identifies. `x` must exist for this statment to run, and we need the existing value for `x` to produce the correct new value for `x`.

5 Changing State

These are all the ways I can imagine for R code to change state. Within each category I've listed them approximately in the order of how frequently I see them used. Some of the code examples here overlap categories ie. manipulating a graphics device may also create a file.

Thank you Nick Ulle for looking these over and suggesting more.

- Variable updates
  - Simple assignment

    `x = 1:10`
  - Removing objects

    `rm(x)`
  - Update object metadata, ie. class, names, other attributes

    `attr(x, "dim") = c(2, 5)`
  - environments

    `environment(f)$x = 20`
  - Call method in reference class possibly mutating object (Implemented with environments, but conceptually different)

    `RCobj$method()`
  - Complex assignment. Possibly nonlocal and / or difficult to analyze.

    `assign(varname, 10), x <<- 10, delayedAssign(varname, 10)`
  - Environment manipulation

    `lockEnvironment(), makeActiveBinding()`
- Plotting
  - Open new device

    `pdf("plot.pdf")`
  - Start a new plot frame

    `plot(x, y)`
  - Write to existing plot frame

    `lines(x, y)`
  - Close a device

    `dev.off()`

- Process state

  All the items listed here can potentially be solved with a "preamble", a script that synchronizes the state before we call any of the actual code.

  - Attaching a library updates the search list, making more objects available and possibly masking others.

    ```
    library("ggplot2")
    ```

  - Sourcing a file. These can be recursively expanded so that we only need to look at the code they contain.

    ```
    source("script.R")
    ```

  - Data frames, lists, or data files can also be attached to the search path, so they act basically like library.

    ```
    attach(iris)
    ```

  - Global options control the default behavior of subsequent R code that accesses options.

    ```
    options(contrasts = c("contr.sum", "contr.sum"))
    ```

  - Hooks allow the user to provide code that will run when certain events happen.

    ```
    .Last = function(...), on.exit(), trace()
    ```

  - Package modification. We hope nobody does this in data analysis code.

    ```
    assignInNamespace("ls", function(...)  "ls!", "base")
    ```

- System state

  - Set the working directory

    ```
    setwd("..")
    ```

  - Set environment variables

    ```
    Sys.setenv(MC_CORES = 4)
    ```

  - System calls from R run OS commands, so they can do nearly anything.

    ```
    system2(...)
    ```

- Files

  - Create files

    ```
    write.table(temp, "temp.txt")
    ```

  - Delete files

    ```
    unlink("temp.txt")
    ```

  - Directly manipulate connections

    ```
    open(con); seek(con, 100)
    ```

- Package specific

  Packages may offer many other conceptual models, these are just a couple common ones.

  - Database connections and cursors. These are similar to general file connections. See the documentation in the DBI package.
  - Special reference objects, ie. ff and bigmatrix
  - Proxy objects for communication between languages
  - R6 is similar to reference classes

For everything here I was thinking about code evaluated at the top level. Is anything different for code evaluated inside functions? Certainly we have to be more careful with lazy evaluation, scoping and closures.

Does `stop()` fit in somewhere here? It affects the state in the sense that subsequent code won't run. So it's like control flow.

6 Challenges

Most (all?) of these issues come from modifying global state.

I used the R language reference manuals as a primary reference, and have tried to remain consistent in how I use the terms and concepts [R Core Team, 2018b] [R Core Team, 2018a].

Connections, options, graphics devices are all global variables that are mutable.

6.1 Global Variables

This code dynamically looks up the variable z from the global environment.

```
f = function() z
z = 10
f1 = f() # 10
z = 20
f2 = f() # 20

# Same thing essentially happens when the default .z=z is used:
g = function(.z = z) .z
```

To handle it correctly we need to ensure that the expression sees the correct version of texttt z.

6.2 Graphics Devices

Plotting commands such as the following must be executed in order on a single worker, so we should group them together in a preprocessing step.

```
png("histogram.png")
hist(x)
dev.off()
```

More generally opening a graphics device marks the beginning of a group, and closing the device marks the end of a group. Every (necessary) call between these that updates or queries the graphic device must run in order to be consistent with R's canvas model of graphics. Ordering isn't a concern in this case, since if it's a single block we have no opportunity for parallelism.

What calls open a graphics device? This can be done explicitly, ie. through `png(), pdf(), ....` It can also be done implicitly, ie. running a single expression from the command line:

```
Rscript -e "plot(1:10)"
```

produces a new file with this plot called `Rplots.pdf` in my current working directory.

How can we tell if an expression updates the graphics device? We can recursively examine all parts of the expression to see if any call a function in the namespace of the graphics package. Subsection 6.3 has more details.

Not all functions in the graphics package actually update the graphics device, for example `hist(..., plot = FALSE)` just computes the bins for a histogram. `strwidth` can compute the width of a string based on the current state of the graphics device.

Thinking more on this- the real constraint is that we need to execute a block of plotting commands in order on a single worker from the time a device is opened to the time that it's closed. Other workers are free to compute intermediate results as long as they are available to the plotting worker when they are needed. I have something in mind like the following:

```
pdf("a.pdf")
plot(a)
lines(f(b))
dev.off()
```

If `f(b)` is very expensive we may want to evaluate it in a separate process and pass in the result. This is the same problem Duncan described in his big nested function example: `a(b(c(f(x), g(y))), h(y))`

TODO: I don't see the string `Rplots.pdf` in options(). Where is this specified?

`plot(x)` overwrites `plot(y)` below since both call `plot.new()` internally. Thus we could remove `plot(y)` in a preprocessing step.

```
png("test.png")
plot(y) # This does nothing
plot(x)
dev.off()
```

Why would people write code like this? Either they don't know what's going on, or they do know what's going on and they overlooked it. Or they simply don't care because they get the result they want in the end.

In general things may get tricky if users switch between multiple graphics devices, ie. with `dev.next()`.

Is it enough to check if code comes from the graphics package? How can we know for sure if code (especially C code) does or does not use the graphics device?

The case for random number generation may well be similar.

The R internals manual [R Core Team, 2018a] has a large section on graphics devices. Digging into it a little we see two internal functions `.Call.graphics, .External.graphics`.

The R extensions manual states "The graphics systems are exposed in headers R_ext/GraphicsEngine.h, R_ext/GraphicsDevice.h (which it includes) and R_ext/QuartzDevice.h" [R Core Team, 2018c]. Including R.h will includes these header files as well I believe. Then we could recurse all the way down into the C code of a package to see if it uses anything defined in these header files.

In practice I think few packages use the C level graphics API. For example, ggplot2 is built on the grid package, and doesn't use C code. So for ggplot2 it suffices to know which functions in base and grid update, query, or start a new graphics device.

6.3 Resolving Functions

For some code analysis tasks we need to know specifically which function is being accessed. For example the function `base::t` is a generic function for transpose. If user code defines a function `t` then this will mask `base::t` when subsequent code calls `t()`.

A useful preprocessing step would be to go walk the code and resolve every function to a namespace if we can. This can happen in something like CodeDepends, since we probably don't actually want to change the code because it will lead to poor performance. A quick check on my laptop shows its 2 orders of magnitude slower to find `base::t` versus `t`. This is just the overhead of the double colon operator.

How can we look into functions recursively? This would be useful for random number generation and plotting.

Basically this is all about packages, because we can see what assignments happen at the top level. Non exported functions in packages will be an issue.

If we're looking at a generic method then we need to explore all methods that might be called. This could get out of hand.

From R Extensions [R Core Team, 2018c]:

"Sealing (package namespaces) also allows code analysis and compilation tools to accurately identify the definition corresponding to a global variable reference in a function body."

How can I use this? In the NAMESPACE file for the autoparallel package I have `importFrom(stats,median)`.

I can attach the package and find the definition:

```
library(autoparallel)
m = get("median", pos = "package:autoparallel")
```

Side note: Initial loading of a library can take several seconds, depending on how much code it brings in. Subsequent loading is a non op, and takes on the order of 25 microseconds. This could also be made faster in R.

Then I have to think about what method will be called.

Here's a related thread from CodeDepends: `https://github.com/duncantl/CodeDepends/issues/19`

I don't think that CodeDepends was designed to recursively dig through packages, which is what I'm considering now. Think about any of the many packages that depend on ggplot2, which in turn depends on grid. This is two layers of indirection from low lying graphics calls in grid, and there could be more. But because of sealed namespaces and lexical scoping we should be able to statically analyze this.

I need a function that looks at a function inside a package and returns the name and package of all other functions that it calls. Then I can call this recursively.

6.4 Global Variables in Packages

TODO - I did this in ddR when I maintained a "current driver".

6.5 Options

Options contain mutable global state. For example, the option `contrasts` determines which functions are called to build model matrices for factors, which changes the meaning of the results. Hence the two results below will be different.

```
options(contrasts = c("contr.treatment", "contr.poly"))
fit1 = lm(y ~ x, data = d)

options(contrasts = c("contr.sum", "contr.sum"))
fit2 = lm(y ~ x, data = d)
```

In general `options()` may change results. To guarantee the same results as R's standard execution model we need to ensure that when an expression implictly or explicitly calls `options()` then the subsequent expressions see those options.

Note this call to `options` is unique in that it's often a simple literal expression and therefore fast to execute, so it's no problem to evaluate this code on all existing workers when we need to synchronize state.

Related concept: environment variables in the operating system.

6.6 Interactivity

Consider a call to texttttlocator() or imagine a program that prompts a user with textttmenu() or texttreadline() before overwriting a file. I'm not sure how people use these features in data analysis scripts, but we may be able to accomodate them by returning control to the manager process.

6.7 Connections

We definitely do not want to write to the same files in parallel. If the connection supports seeking we may be able to read in parallel by carefully managing the state.

Here's an example where the expressions must run sequentially (similar to the graphics device), but not necessarily in the same process (different from the graphics device):

```
f = file("result.txt", "w")
write.table(a, f, col.names = FALSE)
write.table(b, f, col.names = FALSE)
close(f)
```

Forked child processes inherit their parent's connections, which means that it is possible to evaluate texttttwrite.table(b, ...) in a worker process, leaving the manager free to continue execution. This requires inserting the appropriate calls to textttflush() the output to the connection.

6.8 Environments

Also includes reference classes.

7 Scheduling

There are two main paradigms for scheduling a DAG of expressions: static and dynamic. Static means the system specifies which worker will execute every statement in the script and when and how the workers will synchronize before anything begins to execute. In this sense R's normal behavior is static, because it executes each statement in order.

Here's an example of what a static execution scheme for a simple parallel program might look like:

1. Worker processes A and B begin execution simultaneously.
2. A executes statements 1, 3, 4 to create object `x`.
3. B executes statement 2 to create object `y`.
4. B transfers `y` to A.
5. A uses `x` and `y` to execute statement 5.

If the scheduling is not static, then it's dynamic. So dynamic means that the order of execution and synchronization is not completely determined a priori. A simple dynamic scheme is to give the next available expression to the next available worker and store all variables in a global store. This resembles the Linda approach [Carriero and Gelernter, 1989].

A dynamic execution model might execute the above example in the following way:

1. The manager tells A to execute statement 1 and B to execute statement 2.
2. A completes statement 1 and notifies the manager.
3. The manager gives A statements 3 and 4 to execute.
4. B completes statement 2, notifies the manager, then waits.
5. A completes statements 3 and 4 and notifies the manager.
6. The manager decides that it would be faster to run statement 5 on B
7. The manager tells A to transfer `x` to B so it can run statement 5.
8. The manager gives B statement 5 to execute.

In general there's a large cost for going from static to dynamic scheduling. But I think in the case of R that this cost will be less significant because we already have to pay for two major sources of overhead: multiprocessing and an interpreted language. In contrast, typical CS applications use threads in a compiled language, so they have orders of magnitude less overhead.

The dynamic approach appeals to me because I think the advantages will outweigh the potential overhead.

1. R's introspection lets us see how large every object is and therefore estimate the time required to transfer.

2. We can adjust to changing conditions. For example, we may detect multithreading in a worker and use fewer workers when that happens.
3. The worst case scenarios are not as bad. For example, if the static schedule is not ideal then the program will certainly be slow, and we can only know that after the fact.

The CS literature provides many options here.

8 Background

8.1 Concurrency control

Do we need traditional CS concepts like locks, mutexes, and semaphores? It seems like we don't, because we're not actually using shared data structures.

Can we understand any of what the scheduler must do in terms of these objects?

The scheduler should be preventing deadlocks by scheduling things correctly.

8.2 Task Graph

[Sinnen, 2007] defines the task graph and broadly describes the task scheduling problem.

The intro to the book [Sinnen, 2007] is very similar to the content in 3. It's reassuring that I came up with the same important concepts, but I wish I would have picked it up earlier.

8.3 Existing R packages

R's recommended parallel package includes high level functions to support task parallelism. `mcparallel()` evaluates an R expression asynchronously in a worker process and `mccollect()` brings the result back to the manager.

A couple existing R packages provide an abstraction that amounts to task parallelism.

drake describes itself as:

> The drake R package is a workflow manager and computational engine for data science projects. Its primary objective is to keep results up to date with the underlying code and data.

So drake is in some sense GNU make tailored for R.

Rather than write a script the user explictly lists rules to create variables. drake statically analyzes the R code specifying the commands to writes their code in a data frame that acts as a Makefile. Here's a taste:

```
library(drake)
p = drake_plan(x = 1:10, y = sum(x), z = mean(x))
make(p)
```

Running this code results in the variables `x, y, z` being defined in the global workspace. They are also written to a cache in a new `.drake` directory. Variable caching is handled through the storr package, so one can use different mechanisms other than files.

For parallel execution it does keep existing workers around.

This is a pretty different use case from ”compiling” a script to be fast. Also writing dependency rules requires a completely different mindset from the programmer.

It’s worth noting that drake uses futures for the backend.

futures

Other in progress R packages: https://github.com/r-lib/async

8.4 Comparison to frameworks in general

There are positive and negatives to the automatic parallelization approach through code analysis compared to using a specific API . framework. Also similarities and differences from the ”framework” approach. By framework I’m referring to software that provides one user facing API with separate ”backends”. Example of R packages include DBI, dplyr, future, foreach ddR, SNOW. These are proven, they work.

TODO: What is common with a framework?

- Extensible,
- + Kate can use any R code she likes.
- + We’re not limited by the abstractions of a framework, ie. if one system supports more advanced features than another then we don’t have to rely on one abstraction based on the ”lowest common denominator” of features. For example, MPI can do more beyond SNOW.
- + We can apply these techniques to R code in packages that can’t rely on frameworks because it wants to avoid dependencies.
- - Debugging generated code is potentially complex because the debugger may need to know 1) the intent of the original code, 2) the system generating the code, 3) how the target platform works. Thus it’s very desirable to do debugging in the
- - ”Jack of all trades, master of none” - performance may be poor compared to hand written code.

8.5 Shared memory

8.6 Linda

Linda is a generative model of computation: all objects live in a global tuple space where any process can pick them up and use them [Carriero and Gelernter, 1989]. It’s agnostic to the actual language used. Focuses on simplicity. The authors claim communication between different computations is the fundamental issue to address. Good quote: ”Compilers can’t find parallelism that isn’t there.” It means that if one uses a strictly serial algorithm / set of steps then there's little hope to parallelizing it. Calls for a good, standard language for parallelism in 1989. 30 years later we still haven't found it! They also talk about the difficulty of choosing the grain of parallelism- a problem that I'm aware of, having bumped up against it several times.

> When a problem has a simple solution, a useful system will give programmers access to the simple solution. Forcing complex solutions to simple problems makes us suspect that a language has chosen the wrong abstraction level for its primitives, chosen operations with too many policy decisions built-in and too few left to the programmer. Such languages are ostensibly higher level than ones with more flexible operations, but this kind of highlevelness dissipates rapidly when programmers step outside the (often rather narrow) problem spectrum that the language designer had in mind.
>
> [Carriero and Gelernter, 1989]

Linda talks about the dining philosophers problem. Can I imagine a case in R when multiple processes can deadlock? It's certainly possible that they compete for resources, ie. disk IO.

Makes me think- for the past month or so I've been imagining something like a message passing system coordinated by a central manager. But Linda is a conceptually different model. ddR (and Spark) are models for distributed objects. It would be very nice if I could take whatever I infer about the code and put it into a model that I choose.

References

[Carriero and Gelernter, 1989] Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

[Landau, 2018] Landau, W. M. (2018). *drake: Data Frames in R for Make*. R package version 5.1.0.

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

[R Core Team, 2018a] R Core Team (2018a). R internals.

[R Core Team, 2018b] R Core Team (2018b). R language reference.

[R Core Team, 2018c] R Core Team (2018c). Writing r extensions.

[Sinnen, 2007] Sinnen, O. (2007). *Task scheduling for parallel systems*, volume 60. John Wiley & Sons.

[Temple Lang et al., 2017] Temple Lang, D., Peng, R., Nolan, D., and Becker, G. (2017). *CodeDepends: Analysis of R Code for Reproducible Research and Code Comprehension*. R package version 0.5-4.