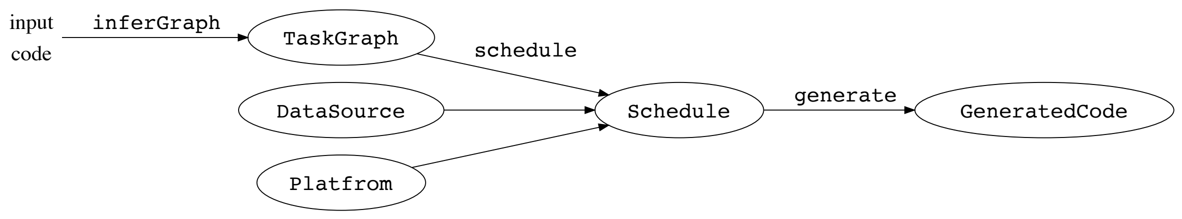

Figure 1: This graph shows the steps in automated parallelization. The nodes are classes in `makeParallel`.

0.1 Software Architecture

This builds on, and will eventually replace, the 'Concepts' vignette for `makeParallel`. In this form it's more useful for my dissertation.

This document should answer the following questions. What are the classes and methods, and how do they relate to each other? Why are they all necessary? Which slots are conceptually important? Which are truly modular, in terms of actually being independent of the other steps? Which ones would it actually be a good idea to subclass? Why do the methods dispatch on the arguments that they do? Who is going to actually create subclasses, and why?

TODO: talk more about data description.

0.1.1 Overview

Automated Parallelization (AP) is the process of transforming an original input program into a new program that's tailored for the platform and the data. Figure 1 shows the steps in AP as implemented in the methods and classes in `makeParallel`. `TaskGraph` contains the dependency relationships of the (sub)expressions, `DataSource` is a description of the data to be processed, and `Platform` represents the hardware and software to run the generated program on. `DataSource`, `TaskGraph`, and `platform` are the inputs to the `schedule` method. The `Schedule` class represents how and where the code should be run, i.e. which parts of the program should be parallel. `GeneratedCode` is a concrete implementation of a schedule, that is, actual code that runs. The function `makeParallel` is a simple wrapper to perform all of these steps in AP at once, namely `inferGraph`, `schedule`, and `generate`.

Each step in AP is complex, and requires a particular kind of expertise. Graph inference requires understanding the semantics and properties of the R language. Scheduling requires expertise on scheduling algorithms . Code generation requires knowledge about systems and networks . By separating these steps, we make it possible for future researchers to experiment with or improve just one aspect of AP, without requiring them to be experts on every aspect.

0.1.2 Details of each step

Users may apply various transformations to their code before using `makeParallel`, for example using functionality in `CodeAnalysis`. We encourage this, because many code transformations are generally useful for improving performance. For example, any program will be faster if we first eliminate unused or redundant computations .

AP begins with a call to `inferGraph` on the user's code to produce a `TaskGraph`. `inferGraph` analyzes the code and determines how the expressions depend on each other. It's an S4 method only for convenience to uniformly handle code that's in files, or expressions / other language objects. We do not expect users to define their own methods for `inferGraph`, nor to define subclasses for `TaskGraph`, because there's only one correct dependency graph for code. `TaskGraph` is a task graph in the classical computer science definition.

It would be possible to leave hooks in for users to specify how certain unknown functions behave, a la `CodeDepends` function handlers. I haven't thought through how this would work.

Scheduling offers the greatest opportunity for experimentation. In general, scheduling a task graph is NP hard, and many algorithms exist to solve this problem. . Users are free to implement and use their own schedulers, which allows researchers to directly compare the performance of competing algorithms on real problems and real data.

`makeParallel` includes several schedulers. The default scheduler is intended to be practically useful on real data analysis problems that are embarrassingly parallel, or that involve GROUP BY computations. `scheduleTaskList` is a reference implementation of list scheduling. .

For data analysis problems that are embarrassingly parallel, task scheduling is overkill, so we have developed several heuristics that quickly produce practical solutions.

What do I mean overkill? I mean that it doesn't make any sense to consider a task graph consisting of the exact same computation on 10 million of the same data elements. We certainly don't want to generate 10 million lines of code. We want to do blocks at a time, and use chunks and vectorization where we can.

0.1.3 Extending

What is the right way to extend the software? Which classes? Why would someone want to do that?

Motivation is very important- sketch out a few hypothetical ways for users to extend it. For extending to be practical it must at least make some things easier, rather than just get in the way and force users into a constricting framework. TODO: Explicitly point out what components are being reused. The best way to motivate it is to illustrate it with a realistic example that is not excessively complicated.

Are there any types of schedules that every platform / code generator should be able to handle? Every platform should be able to do some kind of map reduce computation.

Developers can extend `makeParallel` to handle different kinds of data, and to run on different platforms. (Thus developers are different from typical end users.) Classes `DataSource`, `Platform`, and `GeneratedCode` are the most likely candidates for extension. Apache Hadoop and related projects combine data storage with a compute platform, so in this case the developer must extend all of these classes for `makeParallel` to generally work.

Data can come in many possible organization formats, plain text or binary, row or column oriented, or indexed by columns in a table. Developers can express these special features by creating subclasses of `DataSource`, and then modifying the scheduler to use them.

Code can run on many possible platforms, for example a single Linux machine, SNOW clusters, MPI, SLURM, Apache Spark / Hadoop, or various cloud based compute services. Developers can record important characteristics of these platforms by creating subclasses of `DataSource` and `GeneratedCode` which are specific to the platform. Developers will also need to define a `generate` method to produce a `GeneratedCode` object.

Cite something to make the bibliography appear [1].

Bibliography

[1] R Core Team (2018). R language reference.