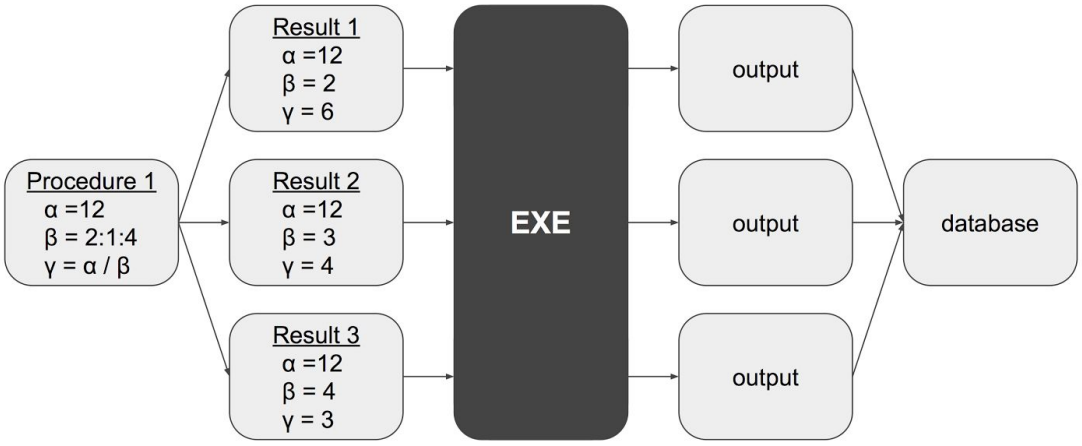
Dan Segal

Ahab and Julia

Bridging the Gap towards Many-Task Computing



Introduction

Ahab.io is a web platform designed for efficiently generating and running batch simulations. Its name hints at this concept: Automated High-throughput computing Around Black boxes (AHAB). The term high-throughput computing (HTC) here refers to Ahab's concern in optimizing computational resources over a long stretch of time. This philosophy rivals high performance computing (HPC), which focuses on maximizing FLOPS (the number of floating operations per second) — what most people are talking about when discussing parallel programming.

In this project, Julia was added to Ahab to bridge the gap between HTC and HPC. This synthesis of approaches (HTC and HPC) is conventionally called Many-task computing (MTC). Practically, this was accomplished by allowing Julia code to be run inside Ahab's black boxes: augmenting Ahab's existing HTC nature with Julia's own HPC flavor.

Background

Ahab's initial purpose was to run parameter sweeps over other people's code. This can be seen in the picture on the first page. A parameter sweep is handled by varying several variables over different values. In the picture's example, alpha is kept at 12 and beta is varied from 2 to 4 in steps of 1 (i.e. 2, 3, 4); gamma is then derived from a run's current value for alpha and beta.

Colloquially, this process is called an embarrassingly parallel problem. In other words, you wrap your code in a bunch of nested for-loops. Ahab works by providing a way to remove these for-loops from your code and build organizational structure around it.

An off-topic point for the interested reader, Ahab is four years old. The first year was spent making a python tool. The second year was spent making a website to learn web development — done using Michael Hartl's rails book. The third was spent full-time, engineering a project with the goal of making a scalable website with a modern architecture, namely: Rails, Ember, AWS, Redis and Docker. This is now year four: adding HPC through Julia and setting sail.

Black Boxes

Black boxes are a paradigm in computer science where code is treated simply as a function that takes in input and spits out output. In Ahab's context, whatever code you write is a black box. And with that, Ahab just runs parameter sweeps around it and digests the generated output—placing the information back into a database for you to analyze later.

Docker

For safety and simplicity reasons, each black box is run on its own sandbox server. This is why Ahab uses Docker: a popular tool for generating server images (inside containers). To avoid a long tangent onto what exactly Docker is, an introduction will be done through analogy to shipping containers—those big metal corrugated boxes that get loaded onto barges.

250 years ago, if you grew coffee, you would have had to deal with a large number of issues that aren't farming. You would have wasted time making pots, aligning various connected shipping lines, etc. Shipping containers changed this by standardizing a shipping device and leaving you with two goals: grow the coffee and pack the coffee as densely as possible in it.

Docker works by prepackaging containers with the minimum amount of server setup needed to accomplish a goal, usually hosting websites. For example, if you were Apple, you might have 10 Docker containers for your landing page and 3 containers for running the online shop. This allows you to scale the number of servers for a website's different functions based on demand.

Ahab uses Docker to allow you to run batch simulations in a similar fashion to how Heroku lets you host websites: I give you the keys to some servers, then you run your code and I parse the output. Ahab just uses Docker because running the first job is as easy as running the millionth.

Julia

The three stages of Julia love are:

1) Being drawn in by its parallel-ness
2) Learning about multiple dispatch
3) Discovering all its niche capabilities

Julia was added to Ahab because of the first reason: it's publicly known parallel capabilities. Its goal was to imbue a high-performance element into the existing high-throughput framework.

In terms of what was needed to do this:

- Julz was written as a tool to run Julia code from the terminal (similar to Rails for Ruby)[1]
- Docker containers were instructed to come shipped with Julia & Julz (code on next page)
- The workers that run the black boxes needed to be updated to run the new code type
- Tests were made to cover the new way of running black-box code (i.e. Julz on Julia)

Some notes:

- The Julz Dockerfile is on the following page
- Julz Job Tests were omitted because they probably aren't very interesting to the reader

- The script file "result_script_job.rb" is proprietary software and cannot be included.
- One note is that its new code: clones a public repo from github, edits its config/input.jl, and then makes a system call to "julz run"

[1] The Julz framework is discussed in the Appendix

The Julz Dockerfile[2]

```
1.   FROM ruby:2.2.6-onbuild
2.   MAINTAINER dan segal <dan@seg.al>
3.
4.   # -----------
5.   #  get julia
6.   # -----------
7.
8.   RUN apt-get update \
9.       && apt-get install -y --no-install-recommends ca-certificates \
10.      && rm -rf /var/lib/apt/lists/*
11.
12.  ENV JULIA_PATH /usr/local/julia
13.  ENV JULIA_VERSION 0.5.0
14.
15.  RUN mkdir $JULIA_PATH \
16.      && apt-get update && apt-get install -y curl \
17.      && curl -sSL
   "https://julialang.s3.amazonaws.com/bin/linux/x64/${JULIA_VERSION%[.-]*}/julia-${JULIA
   _VERSION}-linux-x86_64.tar.gz" -o julia.tar.gz \
18.      && curl -sSL
   "https://julialang.s3.amazonaws.com/bin/linux/x64/${JULIA_VERSION%[.-]*}/julia-${JULIA
   _VERSION}-linux-x86_64.tar.gz.asc" -o julia.tar.gz.asc \
19.      && export GNUPGHOME="$(mktemp -d)" \
20.  # http://julialang.org/juliareleases.asc
21.  # Julia (Binary signing key) <buildbot@julialang.org>
22.      && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys
   3673DF529D9049477F76B37566E3C7DC03D6E495 \
23.      && gpg --batch --verify julia.tar.gz.asc julia.tar.gz \
24.      && rm -r "$GNUPGHOME" julia.tar.gz.asc \
25.      && tar -xzf julia.tar.gz -C $JULIA_PATH --strip-components 1 \
26.      && rm -rf /var/lib/apt/lists/* julia.tar.gz*
27.
28.  ENV PATH $JULIA_PATH/bin:$PATH
29.
30.  # ----------
31.  #  get julz
32.  # ----------
33.
34.  RUN apt-get update
35.  RUN apt-get install -y python-pip
36.
37.  RUN pip install -U 'julz'
38.
39.  # -----------------
40.  #  run ruby script
41.  # -----------------
42.
43.  CMD [ \
44.    "ruby", \
45.    "-r", \
46.    "./result_script_job.rb", \
47.    "-e", \
48.    "ResultScriptJob.new.perform" \
49.  ]
50.
```

[2] The “get julia” section came from the standard Julia dockerfile

Conclusion

Julia is a fun programming language that is still very much in its infancy. As such, it offers a hopefully long trajectory of increasing parallel capabilities in the years to come. That is why Ahab chose Julia as its primary language to explore the high-performance computing landscape. With Julia in its pocket, hopefully Ahab can now move one step closer to many-task computing and rival other large-scale simulation power houses.

Appendix

In addition to bringing Julia to Ahab, two other projects were done during this class. Julz, the first project, aimed to bring a framework resembling Ruby on Rails to Julia. It is was used in the Ahab project because I believe it allowed a very standard approach to running simulations. Julia Observer, the second project, was a recent undertaking that scrubs METADATA.jl and the Github API to create a package browser for Julia.

Julz

Julz is a command line utility for creating scalable Julia projects. Its goal is to remove many of the petty decisions involved in starting a project and establish a standard that can be adopted by the entire Julia ecosystem. In this way it channels Ruby on Rails' mantra of "convention over configuration": tell people where files should go, but allow them to tweak it if they so desire.

The problem Julz attempts to resolve is architecting a Julia project that is resilient to both the frequent on-boarding/turnover of engineers as well as the mass accumulation of files. In this way, a person reading a codebase for the first time does not get bogged down by the sheer number of files that have accumulated over time inside a shallow depth src directory.

The way to alleviate this problem is to treat the src folder and the test folder as boxes that hold smaller boxes of like content. In a Julia context, this means that the src and test folder each have corresponding subdirectories for: methods, types, functions, etc. Each subdirectory then has an associated generator command that can be used to make files of that format.[3]

[3] Example on next page.

For example, generating a Rational type with two integer fields from the command line would use boiler-plate templates to compile:

- a working rational.jl file in the 'src/types' directory
- an associated rational_test.jl in the 'test/types' directory

This may not seem like a big idea at first, but getting every project to adopt this level-deeper folder structure is the cornerstone to a successful framework. With it, packages can be created that expand on the native collection of data structures, i.e. to produce singletons, compilers, etc. Here, the addition of generators just make everything less user-error prone and promotes adequate test coverage of code.

Julia Observer

Quite simply, Julia Observer is a package browsing tool for the Julia language. All it does is scrub METADATA.jl, collect needed information from the Github API, and store everything in a database. It's based off the Ember addon browsing tool: Ember Observer.

"I personally think it's a pretty fun toy. The last-minute 2016 stocking-stuffer for your hard-to-shop-for computationalist." -dan, 25 A.D.[4]

[4] End of Document.