Haskell@Home - Crowdsourcing Computation

Quan Nguyen

March 2016

1 Introduction

1.1 Background

Folding@Home is a distributed computing project run by Stanford University's Pande Laboratory that utilizes a crowdsourced fleet of volunteer computers to run protein folding simulation folding. Volunteers run a client on their computers that receives "work units" from the Folding@Home servers, runs various computations on them, and then sends the results back to the servers. Folding@Home's distributed approach to biomedical computation has enabled the study of vastly more complex proteins over larger time scales than were previously feasible. The data set generated has contributed to drug design and research into diseases such as Huntington's, Alzheimer's, and various forms of cancer.

1.2 Motivation and goals

The motivation behind this project, Haskell@Home, is to generalize the computational model used by Folding@Home, allowing researchers in other fields to collect data more efficiently by utilizing the processing power of volunteer machines rather than their own. Haskell@Home is a library written in Haskell that attempts to abstract away the details of the client-server model, fault tolerance, and data serialization, allowing users to focus on defining their problems (work units) and solutions. The primary goal of the framework is to allow a researcher to focus only on defining their problem and an efficient solution algorithm, without needing any knowledge of the underlying framework details. Haskell@Home therefore exposes a very minimal interface with few parameters – little more than a `runServer` and a `runClient` method, in addition to the core typeclasses. The hope is that this framework will make it easier for scientists to collect large amounts of data that will greatly benefit their research efforts.

2 Core packages

2.1 `cereal`

The `cereal` package is the most popular Haskell library for binary serialization. The library supports converting to and from both lazy and strict bytestrings, and is used by Haskell@Home to read and write data structures on the network. In order to serialize a type, `cereal` requires that it either implement the `Serialize` typeclass with a `put` and `get` method or derive `Generic` (in order to use the generic Serialize implementations of `put` and `get`). Most Haskell@Home users will likely use the generic version, unless they require some special format or compression.

2.2 Web Application Interface and `simple`

The Web Application Interface, or WAI, is Haskell's generic interface between web servers and applications. WAI allows any application that uses it to be run on any "handler" (server) that supports it.

The main component of the interface is the `Application` type, which takes a request and returns a response.

Haskell@Home uses a library called `simple` which simplifies the process of implementing a WAI application. The package includes convenient tools for routing and creating standard HTTP responses (`200`, `404`, etc...). The library also defines the `Controller` monad, which is a wrapper around WAI's `Application`s that provides a convenient monadic syntax, but, most importantly, supports keeping track of application state, which Haskell@Home uses for fault tolerance and data storage. Haskell@Home runs these `Controller`s on the Warp server, a popular webserver commonly used with the Yesod web framework.

2.3 `http-streams`

The default Haskell http client library that most developers use is the `HTTP` package. Haskell@Home originally used this library but later switched over to the `http-streams` package instead. `HTTP` uses `String`s for request bodies by default (and uses instances of `Show` to serialize data). Using the `HTTP` library required packing the serialized ByteStrings generated by `cereal`, adding unnecessary complexity. In addition, the potentially large size of work units (e.g. protein data) makes sending strings suboptimal. The `http-streams`, by contrast, is built on the popular Bytestring streaming library, `io-streams`, which plays well with `Cereal`'s serialized Bytestrings.

3 Implementation

3.1 `WorkUnit`

In order to use the Haskell@Home framework, a developer needs to define what problem they want to solve, and the algorithm necessary to solve it. This concept is represented by the WorkUnit typeclass, which is implemented as follows:

```
class (Show w, Show s, Ord w, Serialize w, Serialize s) =>
      WorkUnit w s | w -> s where
      core :: w -> IO s -- Solve a work unit
      next :: w -> IO w -- Generate a new work unit
      idno :: w -> Integer -- Identify a work unit
```

The WorkUnit implementation makes use of the MultiParamTypeClasses and FunctionalDependencies to define the problem and solution within the same typeclass. The `core` method is the IO action used to find the solution, and is named as such because Folding@Home's computation processes are referred to as "cores". The `next` and `idno` methods are used internally by the server to manage and assign work units.

3.2 `DataStore`

Once a solution is found and send back to the server, there must be a way to integrate these solutions into some kind of data store. In order to provide this behavior, developers must implement an instance of the `DataStore` typeclass. This typeclass is defined as follows:

```
class WorkUnit w s => DataStore m w s | m -> w, m -> s where
      store :: w -> s -> m w s -> IO (m w s)
```

The `store` method type signature looks very similar to the `insert` function for Data.Map, except that the return type is IO. The IO return type allows for for potentially using a database connection as the datastore, while also allowing a pure data structure such as a Map to be used by simply wrapping it with `return`.

3.3 Client

The core functionality of the client consists of the `requestWorkUnit` function, which requests a work unit from the server and then runs an IO action to find the solution. For most users, this will simply be the `core` implementation for their work unit. However, `requestWorkUnit` allows users to wrap `core` in an external IO action in case they want to add extra functionality such as print statements. The `runClient` function calls `requestWorkUnit` in a loop, and acts as the "main" function for clients. There is also a version called `runClientWithDelay` which allows the client to sleep some amount of time after sending each solution, in order to limit CPU usage.

3.4 Server

The HTTP server is the central component of the Haskell@Home framework. At a basic level, the "server" is a `simple Controller` that defines two routes: `GET /work` and `POST /solution/:id`. The controller maintains state in the form of a `WorkUnitTracker` and a `TimeStampTracker`, both of which are accessed via MVars for thread safety. Their types are implemented as follows:

```
data (WorkUnit w s, DataStore d w s) =>
        WorkUnitTracker d w s = WT { pending :: Map Integer w
                                   , latest :: w
                                   , dstore :: d w s
                                   }

data TimestampTracker = TT { times :: Map Integer -> UTCTime
                           , dead :: [Integer]
                           }
```

The controller for the `GET /work` API route is in charge of assigning work units to requesting clients. In order to do so, it will take the `latest` work unit in the `WorkUnitTracker`, replacing it with a new one using the `next` method for the next time a request comes in. It will then add this work unit to `pending` (a map from the work unit's ID number to the actual work unit) and the `times` (a map from the work unit's ID to a timestamp) of the `TimestampTracker` before sending it back to the client.

The only exception is if there are any work units in the `dead` field of the `TimestampTracker`, then work units will be assigned from this list first. This "dead pool" keeps track of any work units that have been pending (without a solution arriving) for longer than some timeout (a parameter passed in to the `runServerWithTimeouts` function. The dead pool is managed by a separate thread which will iterate through the `times` Map each second and place any timed-out work unit ID's in the dead pool.

The controller for `POST /solution/:id` is in charge of accepting solutions identified by their work unit ID. Once the solution is received, the controller runs the `store` method to integrate it into the data store, as well as removes it from `pending`, `times`, and `dead` as necessary. If the ID in the POST request url is not part of `pending`, then the server assumes that the solution is a duplicate solution for a timed-out work unit, and simply ignores it and responds with an OK.

4 Case study - Primes Numbers

In order to demonstrate the feasibility of Haskell@Home, an example implementation of a naive prime number finder is included in the code repository. Most of the code for the implementation is in the `Primes.hs` file.

```
-- language pragmas and imports
newtype Primetest = PT { num :: Integer }
                  deriving (Show, Ord, Eq, Generic)
```

```
instance Serialize Primetest

instance WorkUnit Primetest Bool where
        core = isPrime -- Checks divisors up to sqrt of the number
        next p = return $ PT $ num pt + 1
        idno = num

newtype PrimeMemcache w s = PM { cl :: Client }

newPM :: IO (PrimeMemcache Primetest Bool)
newPM = PM <$> newClient [defaultServerSpec] defaultOptions

instance DataStore PrimeCache Primetest Bool where
        store wu sl cm = do
            set cli encWu encSl 0 900 -- No flags, 900s expiration
            return cm
                where cli = cl cm
                      encWu = B.pack . show . num $ wu
                      encSl = B.pack . show $ sl
```

This example makes use of the `memcache` library (written by David Terei) as the data store. The implementation demonstrates the relatively minimal boilerplate necessary for a fully working implementation. The client and server executables are similarly easy to write – a simple implementation could just use `runClient` and `runServerWithTimeouts` for most of the logic (the example client actually uses `requestWorkUnit` with a `threadDelay` call to demonstrate the dead pool functionality).

5 Future work

Due to time constraints, Haskell@Home is currently a barebones implementation of the desired framework. With additional time, there are several improvements that will likely be convenient for developers using the library.

5.1 Client verification

In its current state, the Haskell@Home server has no way of verifying that any solutions it receives are correct, or at least coming from a trusted client (i.e. a client that the developers themselves implemented). As a result, a malicious entity who wants to disrupt a project's research efforts could request work units and then send bogus solutions in large numbers.

There are a few potential ways of mitigating the problem of solution verification. The simplest solution takes advantage of the fact that some solutions may be fairly simple to verify. Although not an intended use case for Haskell@Home, an implementation of Bitcoin's proof of work scheme using Haskell@Home could verify solutions by simply checking that the generated signature is correct. As long as the verification algorithm is fairly efficient, the server will not be blocked for too long to be feasible.

Unfortunately, most problems will likely not have the easy-verification property. As a result, attempting to verify solutions directly would result in too many lost CPU cycles, causing a performance bottleneck on the server. Therefore, a more reasonable solution would be to verify clients directly. The Haskell@Home server should keep track of verified clients (e.g. by IP address), and accept solutions from those clients as normal. On a work request from an unverified client, the server will send back a work unit that has already been solved (by a verified client), and the client will have to send some number of correct solutions in order to become verified. An alternative scheme would be to send out

the same work unit to multiple unverified clients, and then accept the most popular solution between them, with the assumption that most clients will be trustworthy.

5.2 Automatic timeout estimation

Currently, the Haskell@Home server exposes two run modes – one with timeouts, and one without. The timeout version takes a parameter indicating how many seconds a work unit should be outstanding before being marked as dead. The problem with this approach is that it expects the developer to have a good idea of how long a solution will take to calculate. This is potentially challenging, since different volunteer machines may have different performance, and an incorrect timeout amount may result in the server responding too slowly to a dead work unit, or being too eager to mark a work unit as dead.

Therefore, a useful addition to Haskell@Home would be the ability to automatically estimate a reasonable timeout for work units. This would likely take the form of some kind of moving average scheme based on how long the most recent `N` solutions took to arrive after their work corresponding work units were assigned. This approach will have several benefits. The first is that the library will become a bit simpler to use, no longer requiring developers to figure out a reasonable timeout amount. The other benefit is that the server will be better able to react to changes in computation time. If for some reason a particular range of work units takes longer to solve than previous ones did, the moving average solution time will grow to more accurate reflect the fluctuation.