Concurrency in Go

March 1, 2013

Objective:

Your assignment will be to make a sequential Symbolic Regression engine run in parallel using Go's built in concurrency features. There will be two tasks: (1) syncronous execution of the sub-search processes and (2) asyncronous communication between the sub-search processes. You will only be concerned with data flow through the search process. The low-level details have been taken care of for you.
All questions can be directed to Tony Worm (verdverm@gmail.com)

Background:

Symbolic Regression (SR) is a generalization of (non)linear regression. SR searches for the form of a model when the equation is unknown. The standard implementation is an evolutionary algorithm known as Genetic Programming. Equations are represented as an AST and are evaluated, selected, and breed much like Darwin's model of evolution.

A single search process will initially create a pool of equations, known as a population. The entire population is evaluated on the training data, producing a per-equation error value. The population is then sorted with the Pareto non-dominated sort, which balances accuracy(error) and complexity(size). A new population of children is then created by selecting two parents psuedo-randomly, crossing subtrees of their ASTs, and possibly mutating each offspring. The offspring are then evaluated to determine their fitness. During normal iteration, the parent and offspring populations are combined and sorted together. This Allows offspring to replace parents, but also permits good parents to survive in the population. This process is repeated for a number of iterations, also known as generations, at which point a set of equations are returned. The returned set provides a range of equations from simple to complex with increasing accuracy. This is what island.go does.

To improve the consistancy of the SR algorithm, several of the searches are run simultaneously. Each individual search is known as an Island and are coordinated by a master Search process (search.go). Your assignment is to use Go's concurrency features to manage the flow of information between the Search process and the Island processes.

Task 1: Installation

Download and install the Go software (golang.org). The home page has a download link and instructions. If you are using linux, many distros have the Go language in the repositories now, so you can check there too. There are also many good resources for learning Go's syntax on the website. See the "What's next" section on the installation page.
The homework code is located at github.com/verdverm/go-eureqa. You should be able to **go get github.com/verdverm/go-eureqa** to obtain the homework code. It should be located in the Go working directory setup during the installation process. You may also download a zip or use git if you wish.

Once you have the code, navigate to the directory and run **go build**. You can then run the program by **./go-eureqa**. (Windows may have alternate syntax for running the program)

Four data sets are provided. Use the -data="filename" argument to the go-eureqa program. (You don't need to include the data/ dir when supplying the filename)

You should only need to make modifications to three files.

- main.go: sets up parameters, initializes a search, and calls the search run function.

- search.go: the master search process sets up the islands and cooridates the search process.
- island.go: implements the SR algorithm

The other files are helpers. data.go handles reading the data in and is used by the islands. eqn_funcs.go are implementations of the SR algorithms individual functions. The go-symexpr (which you may not see in the local directory) is a library which implements symbolic expressions. This should automatically be downloaded and installed when you **go build** the assignment.

Task 2: Running in parallel

Currently, the islands are run sequentially, in a loop, in the Search.runSearch() function. In order to make the islands run concurrently, you will need to launch them as gorountines using the **go** keyword.

Modify the Island.initIsland() function to return a channel on which commands can be sent to the island from the main search process. Add a slice of **chan int** to the Search struct for saving each islands returned channel. The commands that should be implemented are: **start,stop, and quit**. You can assign a unique integer to each command or use Go's const to give them names. The Island goroutines can be launched in the Search.initSearch() function, but they should not start processing until they have received the start command.

You will need to add a function to island.go called **runIsland**. This function will use a select statement on the command channel and run Island.step() in the default case, but only if the start command has been received. Receiving the stop command should cause the island to stop processing, but should not cause the goroutine to exit. This is what the quit command is for, and should be a round trip communication.

Finally, you will need to modify the Search.runSearch() function. There should be a loop at the beginning which sends the start command to each island. The current, first loop should be changed so it only receives results from the islands. This will provide the synchronization of the islands so that they stay at the same iteration. After the maximum number of iterations is reached, each island should be sent the stop command. Once all islands have stopped processing, they should send a final report. Once the search process receives the final reports, it should send the quit command to each island and wait for the return message before exiting.

Task 3: Adding migration

Migration is the term used in the SR field to describe the sharing of current results between islands. It allows each island to share what it has learned thus far in its searching with its neighbors. This allows information to flow between islands, creating a coordinated search. You will implement the sharing of current best equations using buffered channels. The communication topology will be a ring, where each Island is connected to two neighbors.

The setup will take place in Search.initSearch() and the operation will take place in island.go

Add two parameters to the SR_Params struct: (1) NumMigrants, the number of equations to send on the channel. (2) MigrationRate, the number of iterations between migrations. In general, we want to send a few equations to our neighbors every few iterations.

Add a slice of **chan []*Eqn**to the Search struct to hold the will need to have channels its two neightbors. Look at the reporting mechanisms for an example of a channel that sends a slice of equations. Two islands will share a channel and migration should be bi-directional. (hint: there is a simple loop and modulous pattern for matching islands and channels)

Implementing the migration mechanisms for each island will require several small changes.

- a slice in the Island struct for holding the incoming migrants.
- a function for receiving the migrants and storing them.
- a function or addition to another function for for sending migrants.
- modifying the Island.selectEqns() function to include the migrants in the selection step.

Task 4: Writeup

Create a document describing the changes you made, referencing file and line numbers.

You should also report the best 8 equations returned for each data set.

Is this needed? If so, what else would you like here?

If you want to:

The current SR implementation is very basic, and likely incapable of finding the equations in the provided data sets. Here are some topics for enhancements. *These enhancements are not required for the assignment.* They will, however, vastly improve the effectiveness of the algorithm. Make sure you have completed the assignment befor

- Asyncronous reporting
- Co-evolution of fitness predictors
- Age-layered populations
- Brood selection