Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
`https://courses.softlab.ntua.gr/pl2/`

Γλώσσες Προγραμματισμού II

Αν δεν αναφέρεται διαφορετικά, οι ασκήσεις πρέπει να παραδίδονται στους διδάσκοντες σε ηλεκτρονική μορφή μέσω του συνεργατικού συστήματος ηλεκτρονικής μάθησης `moodle.softlab.ntua.gr`. Η προθεσμία παράδοσης θα τηρείται αυστηρά. Έχετε δικαίωμα να καθυστερήσετε το πολύ μία άσκηση.

Άσκηση 5 Παραλληλισμός σε Erlang ή Haskell

Προθεσμία παράδοσης: 10/1/2021

Ο σκοπός της συγκεκριμένης άσκησης είναι να σας εξοικειώσει με την παράλληλη/ταυτόχρονη εκτέλεση προγραμμάτων σε Erlang ή σε Haskell. Επιλέξτε μία από τις δύο γλώσσες και, στην περίπτωση της Haskell, επιλέξτε μία από τις δύο δυνατές υλοποιήσεις (parallel ή concurrent). Ακολουθεί η εκφώνηση στα αγγλικά.

You want to find the reverse image for a number of values computed by an unknown integer hashing function, which is implemented in Erlang (or Haskell).

For that purpose you will be given the function and a list of 2^{16} unique hash values. You know that each hash has been generated by an integer between 1 and $2^{27} - 1$.

Task in Erlang

Try to find the reverse image for as many input values as possible. Read the next section to see how your Erlang program should operate. (The description of the task in parallel or concurrent Haskell follows.)

Grading

Your solution should be scalable. Your submission will be benchmarked within a grading framework which will operate in the following way:

1. It will spawn your program in a new process and start a countdown.
2. When the countdown expires it will send a `finish_up` message to your program.
3. Your program must send a `{reply,List}` message back to the grader within 1 sec. or be disqualified.
4. Your program may also send the `{reply,List}` message at any earlier point.

A sample grading framework is included in `reverse_grader.beam` (link to download). It exports the following functions:

- `sample_fun()`: Returns a sample hashing function, which expects a value between 1 and $2^{27} - 1$ and returns a value in the same range.
- `sample_inputs(Fun)`: Given a hashing `Fun`, returns 2^{16} hash values generated by random input values from the domain $[1..2^{27} - 1]$.

- `estimate_timeout()`: Returns an estimation (in milliseconds) of the timeout that would be used, if you were running with 1 scheduler on your current platform trying to reverse values calculated with the function returned by `sample_fun()`.
- `base_score()`: Returns an estimation of the number of inputs from `sample_fun/0` that should be solved with 1 scheduler on your current platform to get full points.
- `sample_grade()`: Invokes the grader which will eventually spawn a new process and call your main function: `reverse_hash:solve(Fun, Inputs, P, Schedulers)`:
  - `Fun` is the hash function that you are trying to reverse (e.g. the one returned by `sample_fun/0`).
  - `Inputs` is a list of hash values that have been calculated with `Fun`.
  - `P` is the Erlang PID of the grader. After it spawns your process it will wait for a `{reply, List}` message, where `List` should be a list of 2-tuples `{Hash, ReverseImage}`, with `Hash` being one of the values in `Inputs` such that `Hash = Fun(ReverseImage)`. At some point the grader will send to the spawned process a `finish_up` message and give you 1 second to reply with your list. If you fail to do so, the grader will disqualify your program.
  - `Schedulers` is the number of usable schedulers that you have available. It can be changed from the default (1 scheduler per core) by passing the `+S` flag when starting the VM.

While grading your program will be run with 1, 2, 4 and 8 schedulers on an 8 core machine. To get 100% in this exercise, you have to be able to solve with 1 scheduler at least as many inputs for the `sample_fun/0` as the `base_score/0`, which uses a simple solver. You should then be able to solve in the same time twice as many inputs with 2 schedulers, four times as many inputs with 4 and eight times as many inputs with 8. You can test your solution with `sample_grade()` before submitting.

Task in Haskell

As there is clear distinction between concurrent and parallel programming in Haskell, you are free to choose which way you want to solve this problem. You only need to choose one way, either concurrent programming using threads (`forkIO`) or parallel programming using parallel map (`parMap`). The lectures mainly covered parallel programming in Haskell, but you can find all necessary information on concurrent programming in Haskell in the later part of Marlow's tutorial, which is linked on the course web page.

Depending on which approach you decide to take, do either (a) *or* (b) below:

(a) Implement the following function in `reverse_hash_solver_par.hs`:

```haskell
solver_par :: (Int -> Int) -> [Int] -> MVar () -> MVar [(Int, Int)] ->
                 Int -> IO ()
solver_par hash inputs signal box schedulers = ...
```

where its arguments are:

- `hash`: the hashing function used
- `inputs`: 2^{16} hash values in a list
- `signal`: an `MVar` set by the main thread to notify the solver to send it all solved cases soon
- `box`: an `MVar` to hold the solved cases, i.e., a list of pairs `(hash, rev_image)`
- `schedulers`: the number of schedulers available

Since the actual data is communicated using `MVars`, the return type is unit captured in an `IO` monad.

The main thread would fork another thread for `solver_par` to run, set the `signal` in 2 seconds, and wait for solved cases in `box` for at most 1 second. Then, the speed, measured in solved cases per time, is calculated, which is used to derive the final score. More about this in the Grading section.

(b) Implement the following function in `reverse_hash_solver_conc.hs`:

```
solver_conc :: (Int -> Int) -> [Int] -> Int -> [(Int, Int)]
solver_conc hash inputs schedulers = ...
```

where its arguments are:

- `hash`: the hashing function used
- `inputs`: 2^{16} hash values in a list
- `schedulers`: the number of schedulers available

The return value is the list of solved cases, with each element being a pair `(hash, rev_image)`.

The main thread would run `solver_conc` with appropriate arguments, and measure the total amount of time it takes. Then, speed, measured in solved cases per time, is calculated, which is used to derive the final score. More about this in the Grading section.

If you need to import additional modules, put those in the `reverse_hash_imports.hs` file; otherwise leave this file empty.

Grading

The grading scheme for this task is provided in `reverse_hash.hs` (link to download). Depending on the approach that you take, you need to modify it to include the right file and call the corresponding score calculation function, `get_score_par` or `get_score_conc`.

```
$ ghc -cpp -O2 reverse_hash.hs -threaded -rtsopts && ./reverse_hash +RTS -N8
For 1 scheduler(s) score: 1.00
For 2 scheduler(s) score: 0.91
For 4 scheduler(s) score: 0.84
For 8 scheduler(s) score: 0.78
```

While grading, your program will be run with 1, 2, 4 and 8 schedulers on a machine with at least 8 cores. For each case, your score will depend on the ratio between your solution's speed (solved pairs per time) and the base speed (result from `base_speed`).