Parallel and Concurrent Programming in Haskell

Lab Exercises

Simon Marlow
simonmar@microsoft.com
Microsoft Research Ltd., Cambridge, U.K.

June 18, 2012

1 Introduction

Lines beginning with $ are commands that you type at the command-line (a terminal in Linux, or a command-prompt in Windows). For example:

```
  $ ghc-pkg list
```

to see what Haskell packages are installed.

If that doesn't work, you may need to add the relevant directories to your PATH:

```
$ PATH=/export/opt/HASKELL/2011.4.0.0/bin:/export/opt/GHC/7.4.1/bin:$PATH
```

Next, install the extra packages that we'll need for the exercises using cabal:

```
$ cabal update
$ cabal install monad-par-0.1.0.3
$ cabal install remote-0.1.1 derive-2.5.8
$ cabal install parallel-3.2.0.2
$ cabal install HTTP-4000.2.3 xml-1.3.12
```

Next, get the sample code and unpack it.

```
wget http://community.haskell.org/~simonmar/par-tutorial-cadarache.tar.gz
tar xvzf par-tutorial-cadarache.tar.gz
```

Build the code. There is a Makefile, so you should be able to say make to build all the programs:

```
  $ cd par-tutorial/code
  $ make
```

Alternatively, you can compile any individual program like this:

```
  $ ghc -threaded -rtsopts -eventlog -O2 sudoku1.hs
```

2 Lab 1: Parallel Haskell

Test that you can get parallel speedup. Compile `sudoku1` as above, and run it on one processor like so:

```
$ ./sudoku1 sudoku17.1000.txt +RTS -s
```

Next, the static parallel version:

```
$ ./sudoku2 sudoku17.1000.txt +RTS -s -N2
```

Next, the `parMap` version:

```
$ ./sudoku3 sudoku17.1000.txt +RTS -s -N2
```

(the last one should be the fastest)

Exercise 1.1. This exercise is to speed up a program that performs full-text indexing and searching of documents. The sequential program can be found in `code/index/index.hs`, and you will need to download and unpack the set of sample documents:

```
$ wget http://community.haskell.org/~simonmar/par-tutorial-ex1.1-docs.tar.bz2
$ tar xvjf par-tutorial-ex1.1-docs.tar.bz2
```

The sample documents are a selection of messages from the `haskell-cafe` mailing list. Try out the program like this:

```
$ ghc -O index.hs
$ ./index docs/*
search (^D to end): <type your search terms here>
```

For example, you could enter ”parallel concurrent” and the program would list the filenames of all the documents that contain both those words. To benchmark the program, run it like this:

```
$ echo "concurrent parallel" | ./index docs/* +RTS -s
```

The program works by creating a mapping from words to the set of documents that contains that word. Documents are numbered by their order on the command-line, so that we can represent a set of documents by `IntSet`.

```haskell
type DocSet = IntSet
type DocIndex = Map Word DocSet
```

The program spends most of its time building up the `DocIndex` for the whole set of documents, so that's where you want to focus your attention. There are two functions provided for building the index. The first, `mkIndex`, builds a `DocIndex` for a single document, given its number and the document represented as a lazy `ByteString` (if you haven't come across the latter yet, don't worry - just think of it as a string that is read from a file on demand).

```
mkIndex :: Int -> L.ByteString -> DocIndex
```

The second function you need is `joinInidices`, which combines a list of `DocIndex` into a single `DocIndex`:

```
joinIndices :: [DocIndex] -> DocIndex
```

The part of the program that builds the `DocIndex` looks like this:

```
    -- indices is a separate index for each (numbered)
       document
    indices :: [DocIndex]
    indices = zipWith mkIndex [0..] ss

    -- union the indices together to
    index = joinIndices indices
```

So there are two steps going on: first we call `mkIndex` on all the files `ss`, and then we combine all those `DocIndex` values into a single `DocIndex`.

Your task is to parallelise this as much as possible. Don't expect to get a *lot* of speedup on this program: a factor of 2 on 4 cores would be good. The main goal is to get some speedup, and to get a feel for what kinds of things are effective, while gaining some familiarity with the programming models.

- You can use either the `Eval` monad and Strategies, or the `Par` monad, or even both if you like. After trying with one, why not try with the other.

- Start by making the `mkIndex` calls happen in parallel. They are completely independent, so this part should not be too hard.

- The program builds a large data structure in the heap, and spends a lot of time in the garbage collector. So you might find that it runs faster and scales better when given a larger "allocation area":

  ```
  $ echo "concurrent parallel" | ./index docs/* +RTS -s -A128M
  ```

- Then look at `joinIndices`. This is an associative operation: for example, `joinIndices [a,b,c,d]` is the same as

  ```
  joinIndices [joinIndices [a,b],joinIndices [c,d]]
  ```

So rather than a single `joinIndices` call, we could express it as a tree-accumulation. Your job is to experiment with different ways to build up the index using calls to `joinIndices`, and see how much parallel speedup you can get.

Remember that rearranging the order of `joinIndices` can have an effect on the sequential performance too - so if you improve over the performance of the original sequential program, use your program as the new baseline for calculating parallel speedup.

3 Lab 2: Concurrent Haskell

The sample code corresponding to the examples in the notes can be found in `par-tutorial/code`. Try the `fork` example from Section 3.1:

```
  $ ghc -threaded -rtsopts -eventlog --make fork.hs
  $ ./fork
  ABABABABABABABABABABABABABABABABA...
```

Exercise 2.1. In `par-tutorial/code/bingtranslator.hs` there is a program that translates a line of text into multiple languages using the Bing Translate API. For example:

```
$ ./bingtranslator "translate this"
"translate this" appears to be in language "en"
ar:
bg:
zh-CHS:
zh-CHT:
cs: peloit
da: overstte dette
... etc.
```

Compile the program - for this you may need to install the `xml` and `utf8-string` packages first:

```
  $ cabal install xml utf8-string
  $ ghc -threaded -rtsopts -eventlog --make bingtranslator.hs
```

When the program is compiled, run it yourself with some sample text.[1] Note how the translations appear slowly - the program queries the Bing API for each translation in sequence.

Exercise 2.1. Convert the program to perform all the translations concurrently. Use the `Async` API (the code is in the `bingtranslator.hs` source file).

Exercise 2.2. The program also makes two initial queries to the Bing API: one to get the list of supported languages, and another to detect the language in which the initial text is written. Make these two queries concurrently.

A sample solution can be found in `bingtranslatorconc.hs`.

[1]On Windows you may not see the international characters appear correctly on the console. First, switch the codepage to UTF-8 with `chcp 65001`. Then redirect the output of `bingtranslator` to a file, and open the file in Notepad to see the output correctly.

Exercise 2.3. Write a simple game that behaves as follows:

- A sequence of digits (initially empty) is displayed.
- Each second, another digit is pushed on the left end of the sequence.
- When the user types a digit, all matching digits are removed from the sequence.
- If the sequence becomes ten digits long, the game is over.
- The user scores one point for every digit successfully deleted.

You will need to start your program like this:

```
main = do
 hSetBuffering stdout NoBuffering
 hSetBuffering stdin NoBuffering
 hSetEcho stdin False
```

to ensure that keypresses are received immediately and not echoed to the screen. You also need to be able to update the string of digits on the screen; one way to do this is to move the cursor backwards and overwrite the old string with spaces, and then write the new string. You can do this as follows:

```
     putStr (replicate n '\8')
     putStr (replicate n ' ')
     putStr (replicate n '\8')
```

where `n` is the length of the string you want to overwrite.

Hints. One way is to structure the program as three threads, with a single `MVar`. The `MVar` stores events, where an event is either a key press, or a time event indicating that a second has elapsed and another digit should be added. One thread listens for key presses (using `getChar`) and puts them in the `MVar`, another thread repeatedly waits for one second (using `threadDelay`) and then puts the time event in the `MVar`. The third thread repeatedly takes an event from the `MVar` and updates the screen in response.

Exercise 2.3.1. Make it so that new digits are pushed faster as the current score increases. For this you will need to treat the current score as a piece of shared state between the timer thread and the main thread.

Sample solution: `code/game.hs`

4 Lab 3: Software Transactional Memory (STM)

Exercise 3.1. Implement a bounded channel type, with the following signatures:

```
data BoundedChan
newBoundedChan   :: Int -> STM (BoundedChan a)
readBoundedChan  :: BoundedChan a -> STM a
writeBoundedChan :: BoundedChan a -> a -> STM ()
```

`newBoundedChan` takes the size of the channel, and `writeBoundedChan` blocks if the channel is full.

(After you've dong this, if you're feeling very brave, try implementing it with `MVar` instead of `STM`. Obviously the functions must all be in the `IO` monad rather than `STM`.)

Exercise 3.2. (optional: for the performance-obsessed) Write a program in which two threads communicate over a channel, one writing a large number of items to the channel and the other reading them. Measure the time it takes, using (a) the `Chan` type, (b) the `TChan` type, (c) the bounded channel from Exercise 3.1 (choosing a suitable bound). Can you explain the differences in performance?

Exercise 3.3. Write a program that corrects your punctuation as you type.

In this exercise we are going to write a program that does the following:

- Accepts characters typed by the user, and echos them to the screen.
- Automatically applies a correction to the input string. The correction is required to happen in a separate thread, because in principle computing the correction might take time, and we want the input to remain responsive.

Use the following guidelines to structure the program:

- Store the current input string in a `TVar`.
- Use three threads:
  - The main thread reads characters from `stdin` and appends them to the string in the `TVar`.
  - The `render` thread watches for changes to the `TVar` (as in the windowing example in the lecture/notes). When a change is detected, it renders the new string. Hint: you will need to delete the old string by emitting the correct number of `'\8'` backspace characters before printing the new string using `putStr`.

– The `correcter` thread also watches for changes to the `TVar`, and when it detects a new string it applies a correction function to it and writes the result back to the `TVar`. The correction function can do whatever you like; e.g. my correcter in the sample solution capitalises the first character after a full stop.

Note that you will need to set `stdin` and `stdout` to no-buffering mode, and disable echoing. Your `main` function should start like this:

```
import System.IO
import Control.Concurrent
import Control.Concurrent.STM

main = do
  hSetBuffering stdin  NoBuffering
  hSetBuffering stdout NoBuffering
  hSetEcho stdin False
  tvar <- newTVarIO ""
  forkIO (render tvar)
  forkIO (correcter tvar)
  ...
```

then you need to implement `render`, `correcter`, and the code at the end of `main` that reads the input characters.

Sample solution: `code/correcter.hs`

Exercise 3.4. Modify exercise 3.3 so that your text is automatically translated into Japanese as you type it, using the Bing translation service (see exercise 2.1) in the background.

5 Lab 4: Server applications

In this exercise we'll add some extensions to the sample chat server.

Compile the chat server like so:

```
$ cd code/chat
$ ghc --make -i.. Main.hs -o chat
[1 of 2] Compiling ConcurrentUtils  ( ../ConcurrentUtils.hs, ../ConcurrentUtils.o )
[2 of 2] Compiling Main             ( Main.hs, Main.o )
Linking chat ...
```

Check that you can run it and that it works properly.

```
$ ./chat
Listening on port 44444
```

Now switch to another window, and connect a client:

```
$ nc localhost 44444
What is your name?
a
*** a has connected
```

Now switch to a third window, and connect another client:

```
$ nc localhost 44444
What is your name?
b
*** b has connected
```

when you connect client `b`, you should see a message on client `a`'s session: `*** b has connected`. Now try typing messages into each client and check that the messages are broadcast properly. Try kicking a client with `/kick b`.

The following exercises (4.1.1–4.1.6) are to add various enhancements to the chat server, and they get progressively harder. Feel free to skip to exercise 4.2 at any time you like.

Exercise 4.1.1. Add a `/names` command to list the currently connected users.

Exercise 4.1.2. Add a timeout to the "What is your name?" question. You probably want to use `System.Timeout.timeout`.

Exercise 4.1.3. Add a timeout to the client loop: an inactive client should be autonatically disconnected after a fixed time limit.

Exercise 4.1.4. `broadcast` is inefficient because it uses an unbounded transaction (see the section on performance of STM in the notes). Change the server to use a single broadcast channel; you can either use `TChan` with `dupTChan`, or alternatively build your own. Note that this will mean that `runClient` will need to check the broacast channel in addition to its `clientSendChan` and the `clientKicked` variable.

Exercise 4.1.5. Add a `/nick` command to change the current client's name. Careful! The name is stored as an immutable value in the `Client` record, and it is the key of the `clients` map. Some refactoring of the program will be needed to make the name modifiable. Hint: give each client a unique number, and use this as the key in the map.

Exercise 4.1.6. Add flood prevention: prevent a client from issuing more than a certain number of messages in a given time.

Exercise 4.2. The sample program in `code/arithgame.hs` is a simple single-player arithmetic game. The exercise is to make it into a multiplayer game, that works as follows:

- All current players are shown a random arithmetic question, e.g. `6 * 19`.
- The first player to type in the correct answer gets a point; all the other players are notified who answered correctly and another question is shown.
- If nobody answers correctly within ten seconds, another question is shown.
- After ten questions, the game is over and the scores are shown.
- New players can connect any time, and are asked for their name when connecting (as in the chat server).

You might want to start with the chat server as a basis for this program, since some of the functionality is similar: clients need to connect and tell the server their name.

6 Lab 5: Distributed programming

A key-value store is a simple database that supports only operations to store and retrieve values associated with keys. Key-value stores have become popular over recent years because they offer scalability advantages over traditional relational databases, in exchange for supporting fewer operations (in particular, database joins).

This exercise is to use the `remote` framework to implement a *distributed fault-tolerant key-value store* in several stages. You probably won't get to the end in the time available, but I think it's a fun exercise and I hope you enjoy it nonetheless!

The interface exposed to clients is the following:

```haskell
type Database
type Key   = String
type Value = String

createDB :: ProcessM Database
set      :: Database -> Key -> Value -> ProcessM ()
get      :: Database -> Key -> ProcessM (Maybe Value)
```

where `createDB` creates a database, and `set` and `get` perform operations on it. The `set` operation sets the given key to the given value, and `get` returns the current value associated with the given key, or `Nothing` if the key has no entry.

Exercise 5.1. In `remote-db/db.hs` I have supplied a sample `main` function that acts as a client for the database, and you can use this to test your database. The skeleton for the database code itself is in `Database.hs` in the same directory: the first exercise is to implement a single-node database by modifying `Database.hs`. That is:

- `createDB` should spawn a process to act as the database. It can spawn on the current node.
- `get` and `set` should talk to the database process via messages; you need to define the message type and the operations.

When you run `db.hs`, it will call `createDB` to create a database, and then populate it using the `Database.hs` source file itself; every word in the file is a key that maps to the word after it. The client will then lookup a couple of keys, and then go into an interactive mode where you can type in keys that are looked up in the database. Try it out with your database implementation, and satisfy yourself that it is working.

Exercise 5.2. The second stage is to make the database *distributed*. The basic plan is that we are going to divide up the key space uniformly, and store each portion of the key space on a separate node. For example, operations on the key `"Simon"` might go to node 1, whereas operations on key `"Andres"` go to node 2, and `"Ralf"` is handled by node 3. The exact method you use for splitting up the key space is up to you, but one simple scheme is to take the first character of the key modulo the number of workers.

There will still be a single process handling requests from clients, so we still have `type Database = ProcessId`. However, this process needs to delegate requests to the correct worker process according to the key.

- Arrange to start worker processes on each of the nodes with `WORKER` role. (for how to do this, see the example code for multi-node ping in the notes, which is also in `remote-ping/ping-multi.hs`).
- Write the code for the worker process. You probably need to put it in a different module (e.g. called `Worker`), due to restrictions imposed by Template Haskell. The worker process needs to maintain its own `Map`, and handle get and set requests.
- Make the main database process delegate operations to the correct worker. You should be able to make the worker reply directly to the original client, rather than having to forward the response from the worker back to the client.

Compile `db.hs` against your distributed database, and make sure it still works.

Sample solution: `db2.hs`, `DatabaseDistrib.hs`, `Worker.hs`

Exercise 5.3. Make the main database process monitor all the worker processes. Detect failure of a worker and emit a message using `say`. You will need to use `receiveWait` to wait for multiple types of message; see the `ping-fail.hs` example for hints.

Note that we can't do anything sensible if a worker dies yet, that is the next part of the exercise...

Exercise 5.4. Implement *fault tolerance* by *replication.*

- Instead of dividing the key space evenly across workers, put the workers in pairs and give each pair a slice of the key space. Both workers in the pair will have exactly the same data.
- Forward requests to both workers in the pair (it doesn't matter that there will be two responses in the case of a `get`).

- If a worker dies, you will need to remove the worker from your internal list of workers so that you don't try to send it messages in the future.[2]

This *should* result in a distributed key-value store that is robust to individual nodes going down, at least if we don't kill too many nodes too close together.

Try it out - kill a node while the database is running, and check that you can still look up keys. If you got this far, well done!

Sample solution: `db4.hs`, `DatabaseRepl.hs`, `Worker.hs`

[2]A real fault-tolerant database would restart the worker on a new node and copy the database slice from its partner - by all means have a go at doing this but the provided solutions don't do this.

7 Lab 6: GPU programming

Your goal in this exercise is to **crack my password**.

My password is a dictionary word that can be found in the file

```
/usr/share/dict/american-english
```

I have hashed the password using a checksum algorithm called CRC32 (32-bit Cyclic Redundancy Check), and the hash of my password is

0xb4967c42

In order to find the password, you will have to compute the CRC32 value for every word in the file `/usr/share/dict/american-english`, and then find the word that has the CRC32 value `0xb4967c42`.

Some ordinary sequential Haskell code for computing CRC32 can be found in `code/crc32/CRC32.hs`. You can try it out by hand on a few words:

```
> :load CRC32.hs
[1 of 1] Compiling CRC32            ( CRC32.hs, interpreted )
Ok, modules loaded: CRC32.
*CRC32> crc32String "hello"
3387906425
```

You can see the value in hex using `printf`:

```
*CRC32> import Text.Printf
*CRC32 Text.Printf> printf "%x\n" crc32String "hello"
*CRC32 Text.Printf> printf "%x\n" (crc32String "hello")
c9ef5979
```

The CRC32 code is quite simple:

```
crc32 :: [Word8] -> Word32
crc32 msg = go 0xffffffff msg
  where go crc []     = crc
        go crc (w:ws) = go crc' ws
           where crc' = (crc `shiftR` 8) `xor`
                        (crc32_tab !! fI (fI crc `xor` w))
                 fI x = fromIntegral x
```

It's basically a loop over the bytes of the input string, performing a few bitwise operations at each stage. It uses a 256-entry lookup table `crc32_tab :: [Word32]`, which can also be found in the file `CRC32.hs`, you can just import this module to gain access to it.

Our aim is to compute the CRC32 in parallel on all the words in the dictionary, using the GPU. The rest of this section will lead you to the solution in several stages and give lots of hints along the way; if you want a challenge then stop reading now and try to solve the problem on your own!

Exercise 6.1. write the following function:

```
crc32_one :: Acc (Vector Word32) -> Exp Word32 -> Exp Word8
          -> Exp Word32
```

This performs one iteration of the CRC32 calculation. The first parameter is the lookup table, the second parameter is the current CRC value, the third parameter is the byte of the input string, and the output should be the new CRC32 value.

There is one extra criterion: *if the input byte is zero, then the output CRC must be the same as the input.* This will enable us to run `crc32_one` in parallel on multiple strings even though the strings vary in length: we'll just pad the shorter strings with zeros.

Exercise 6.2. Using `crc32_one`, write the following function to compute the CRC32 values for a list of strings:

```
crcAll :: [String] -> Acc (Vector Word32)
```

We're going to do this by mapping `crc32_one` over a *slice* of the input strings. That is, first we map over all the first characters, then over all the second characters, and so on until we have reached the maximum string length.

Start by bringing the lookup table `crc32_tab` into the `Acc` world:

```
    table :: Acc (Vector Word32)
    table = ...
```

Next, you should calculate both the number of strings (call this `n`), and their maximum length (call this `width`).

```
    n = ...
    width = ...
```

Then create a vector of length `n` containing the initial CRC values, which are all `0xffffffff`. Hint: use `fill`.

```
    init_crcs :: Acc (Vector Word32)
    init_crcs = ...
```

Write a function `one_iter`, which performs one iteration over a vector of input characters. In here you will call `crc32_one`:

```
    one_iter :: Acc (Vector Word32) -> Acc (Vector Word8)
             -> Acc (Vector Word32)
```

Finally, we want to write a function to perform all the iterations:

```
    all :: Int -> Acc (Array Word32) -> [String]
        -> Acc (Array Word32)
    all 0 crcs words = crcs
    all x crcs words = ...
```

The first argument is a counter, which starts at `width` and counts down to zero. The second argument is the vector of current CRC values, and the third argument is the current list of strings, where the first character of each is the next to process (at each iteration we will remove one character from the head of each string).

In order to complete this function, you will need to create an `Acc (Vector Word8)` consisting of the first character from each string (or zero if the string is already empty). Then pass this to `one_iter` to calculate the new CRC values, and finally recursively call `all` with the new CRC values and the remainder of each string.

Test this out by calling it from GHCi with a few sample strings, and test that you get the same results as calling the pure Haskell version `crc32String`.

Exercise 6.3. Find the index of the element in the array that has the correct CRC32 value.

We want to write this function:

```
find :: Acc (Vector Word32) -> Acc (Scalar Int)
```

which takes the array produced by `crcAll`, and returns the index of the element that has the value we are looking for.

You could do this in two stages: first map every element to either (a) its index if it has the correct value or (b) zero otherwise, and then fold the `max` function over the resulting array. NB. there appears to be a bug in Accelerate such that `max` doesn't work on the GPU with `Int` arguments, so we have to use `Int32`.

Hint: an array of the same shape as an input array `arr`, in which every element contains its index as an `Int32` is given by:

```
    generate (shape arr) (A.fromIntegral . unindex1)
```

Exercise 6.4. Having done all this, you should be able to use this `main` wrapper to find the answer:

```
main = do
  s <- readFile "/usr/share/dict/american-english"
  let ls = lines s
  let [r] = toList $ run $ find $ crcAll ls
  print (ls !! r)
```

You can run it with the interpreter, and it will take a few seconds. To actually run it on the GPU, make sure you replace

```
import Data.Array.Accelerate.Interpreter
```

with

```
import Data.Array.Accelerate.CUDA
```

at the top of your program. Does it go faster on the GPU?

Sample solution: `code/crc32/crc32_acc.hs`