Homework 3

Homework 3 is about F#. It consists of 3 parts. Please email all answers to jdodds@princeton.edu, chris@monsan.to, cbell@cs.princeton.edu, dpw@princeton.edu and august@princeton.edu . Please put COS597c in the subject line.

This zip contains a folder sections 1 and 2. Submit a single zip/tar. The tar should have a single file at the root that contains any of your non-code results. All code results should remain in the folder for the project.

PROBLEM 1 consists of 3 parts. It is about using asyncs in F# to improve the performance of quicksort. The text of the problems also appears in the .fs file.

PROBLEM 1.1: Create a function "qsortPar1" that tries to do as many steps in parallel as possible. On a machine with infinite processors this program would be extremely fast, but on our machines it will probably be much slower than even the sequential program.

Report the performance change for sizes 1000, 10000, 100000, 1000000. Report anything that takes longer than a minute as a timeout. (If your interactive window is stuck you can right click it and reset session) Also describe the processor you run it on.

For this problem you may find the F# power pack useful. You can download it at
http://fsharppowerpack.codeplex.com/

If you would like more information about asyncs and the Async class these might be helpful:
http://msdn.microsoft.com/en-us/library/ee370232.aspx
http://msdn.microsoft.com/en-us/library/dd233250.aspx

The code

```
#if INTERACTIVE
  #r "FSharp.PowerPack.Parallel.Seq";;
  #time;;
#endif
```

allows the interactive window to use the powerpack.
You might still have errors in your visual studio window
but your program is acceptable if it runs in the interactive window.
The #time switch will cause the interactive window to print how long any operation it runs took.
For example if you type #time;; into your interactive window you will see:

--> Timing now on

From then on in that interactive session, any time you run a command in your interactive window it will look like:

Real: 00:00:01.417, CPU: 00:00:01.653, GC gen0: 110, gen1: 32, gen2: 5

val result : int [] = ...

The number we are interested in is Real

PROBLEM 1.2: Explain in a sentence what caused the performance decrease you saw. Compare this problem to the prime number example from class. Explain how the design is similar or different to the prime number example and relate this similarity or difference to the performance changes from each.

The examples from class can be found at
http://www.cs.princeton.edu/courses/archive/fall10/cos597C/docs/asynchclassdemo.fs

PROBLEM 1.3: Write a new function qsortPar2 that improves in performance over qsort and qsortPar1. Because qsortPar1 is completely parallelized you should only be decreasing the amount of parallelization in this step. Report the performance increase. Finally describe in a few sentences any other designs you tried leading up to your final implementation of qsortPar2

PROBLEM 2 asks you to develop a small application using FRP primitives. We will use CJ's F# polling implementation. Check Assignment.fs (FRP directory) for information.

PROBLEM 3 is about message passing in F#. It turns out that I (Chris Monsanto) am right: Erlang was inspired by the Actor Model (http://en.wikipedia.org/wiki/Actor_model). F# has actors too! Here is a quick tutorial from MSDN:
http://blogs.msdn.com/b/concurrently_speaking/archive/2009/05/12/actors-in-f.aspx. It implements the standard ping pong example that we went over in class. The code should look pretty familiar to the slides from Erlang: instead of "receive", we create a MailboxProcessor. The MailboxProcessor.start function takes a function which is passed an "inbox". You can use the Receive() method of the inbox to grab the next method. If you have someone elses Mailbox, you can use the Post() method on it to send it something. Also, note that MailboxProcessors use asyncs—makes sense, after all, actors are asynchronous computations!

So, what are you going to implement? Let's try a cache for web pages. Design a server actor that responds to "Get" messages for a specific web page. The cache will store web page links, their contents, and their last access time.If the page is isn't in the cache, we download it, put it in the cache, put the correct time, and tell the client the contents and when the contents will expire. If it is in the cache, just retrieve it from there and send it to the client (again, with when the contents will expire). Clear cache entries out if they haven't been accessed in a while; say, 1 minute. Be sure to implement this using recursion… no mutable state allowed!

Then create a function that, given a number N, creates N clients that Get random web pages (maybe randomly selected from a list of 20 or so links).

What considerations would there have been if you were writing this in barebones C using
1) shared memory or
2) interprocess I/O
that you didn't have to deal with in F#?