CpSc 418

Midterm

October 22, 2013

100 points

Time for the exam: 80 minutes.

Open book: anything printed on paper may be brought to the exam and used during the exam. This includes the textbook, other books, printed copies of the lecture slides, lecture notes, homework and solutions, and any other material that a student chooses to bring.

Calculators are allowed: no restriction on programmability or graphing. There are a few simple calculations needed in the exam, a calculator will be handy, but the fancy features will not make a difference.

No communication devices: That's right. You may not use your cell phone for voice, text, web-surfing, or any other purpose. Likewise, the use of computers, iPods, etc. is not permitted during the exam.

Test books: I have included space with each question for you to write your answers. You may use a test booklet if you need more space.

Good luck!

CpSc 418

Midterm

October 22, 2013

Graded out of **100 points** (102 points are possible)

0. (**2** points)

   (a) Your name: Mark Greenstreet

   (b) Your student number: 00000000

1. **Erlang** (20 points) Consider the following code for computing the Fibonacci numbers:

```
fib(1) -> 0;
fib(2) -> 1;
fib(N) when is_integer(N), N > 2 -> fib(N-1) + fib(N-2).
```

I tried it, and got the results shown below:

N	fib(N)	time
1	0	0.28μs
2	1	0.28μs
3	1	0.34μs
4	2	0.41μs
5	3	1.7μs
10	34	3.4μs
15	377	35.2μs
20	4181	389.3μs
25	46368	4.3ms
30	514229	47.6ms
35	5702887	528.9ms
40	63245986	5.8s

(a) (**5 points**) Show that for $N > 10$, the runtime is roughly proportional to `fib(N)`. Give a one sentence explanation for why this is the case.

Answer: By dividing, the time to compute `fib(N)` seems to be about `fib(N)` $cdot 10\mu$s. There are `fib(N)` calls of the form `fib(2)` to get enough `1`s to add up to `fib(N)` – that gives a lower bound of `fib(N)` time.

(b) (**5 points**) Fill-in the blanks to complete the more efficient calculation of the Fibonacci numbers below:

```
fib2(1) -> 0;
fib2(N) when is_integer(N), N > 1 ->
   hd(fib2(N, [1,0])).  fib2(2, AB) -> AB;
fib2(N, [A, B]) ->
   fib2(N-1, [A+B, A])
```

(c) (**10 points**) Consider the function `score` defined below:

```
score(PPid, N) ->
   case N of
      0 -> receive V when V >= 0 -> score(PPid, 4-V)       end;
      1 -> receive V when V >= 0 -> 10                     end;
      3 -> receive V when V >= 0 -> score(PPid, V*V)       end;
      5 -> receive V when V >= 0 -> score(PPid, V*V+2)     end;
      7 -> score(PPid, 10);
      9 -> receive V when V >= 0 -> score(PPid, N-V)       end;
      _ when N =< 10 -> PPid !  N
   end.
```

Fill-in values for the three send expressions in the code below to complete a function, `grader`, that when executed will print your score for this problem:

```
grader() ->
   MyPid = self(),
   CPid = spawn(fun() -> score(MyPid, 0) end),
   CPid ! 1 % next consider score(PPid, 3)
   CPid ! 3 % next consider score(PPid, 9)
   CPid ! 2 % next consider score(PPid, 7)
   io:format("your score on this problem is ~b~n",
                  [receive S -> S after 1000 -> 0 end]).
```

2. **Performance** (30 points) Short answer. Each answer should be one to three sentences. Points may be taken off for answers longer than 100 words.

   (a) (**5 points**) What is *super-linear speed-up*?

   Answer: If a parallel application running with P processors has a speed-up relative to a sequential application greather than P, the parallel version is said to have *"super-linear speed-up"*.

   Name one common cause/explanation for super-linear speed-up.

   Answer: The most common cause is that the parallel version has more total fast memory (e.g. registers, or cache). Thus, it may be able to execute with fewer cache misses than the sequential code.

   (b) (**5 points**) What is *communication overhead*? Define the term *and* give a one or two sentence example.

   Answer: Communication overhead is time spent by a parallel program for communication between processes. For example, in our many examples using reduce, messages must be sent and received up the tree to produce the final result.

   (c) (**5 points**) What is *computation overhead*? Define the term *and* give a one or two sentence example.

   Answer: Computational overhead occurs when the parallel program performs computation that the sequential version does not. For example, when computing the primes $\leq$ N, we had a solution where each of the P processors would first compute the primes in $2 \dots \sqrt{N}$. The sequential version would only need to compute this set of primes once. The parallel version replicated the computation to avoid communication.

(d) (**5 points**) What is *Amdahl's Law*? Give the mathematical formula *and* a short, English explanation of what it suggests about parallel computing.

Formula:

$$SpeedUp \quad = \quad \frac{P}{1+(P-1)s}$$

where P is the number of processors, and s is the fraction of inherently sequential code. Note that if $P \ll 1/s$, then $SpeedUp \approx P$, and if $P \gg 1/s$ then $SpeedUp \approx 1/s$.

In English: Amdahl's law says that speed-up is limited by the the fraction of the total work that must be performed sequentially. If s were fixed, this would limit the maximum amount of achievable parallelism. For many problems, s decreases with increasing problem size. In this cae, Amdahl's law tells us that for those problems, parallel computing is most effective when used with large problem sizes.

(e) (**5 points**) Write a short (less than five lines) code fragment, and point out a RAW dependency. Explain why it is a dependency.

Code Fragment:

```
[1]   for(k = 0; k < n; k++)
[2]     sum = sum + a[i,k] * b[k,j];
```

Dependencies: The reads of `k` on line 2 (`a[i,k]` and `b[i,k]`) depend on the writes on line 1 (`k=0`, and `k++`).

Another example: the write of `sum` on line 2 depends on the reads of `sum`, `a[i,k]` and `b[i,k]` on line 2 as well.

(f) (**5 points**) Write a short (less than five lines) code fragment – you may use the same code as for question 2e – and point out a WAR hazard. Give a short explanation of how the hazard can be removed by renaming.

Code Fragment: Same as above.

Dependency: The write of `k` on line 1 for `k++` follows the read of `k` on line 2 for `a[i,k]` and `b[i,k]`.

Renaming: With register renaming, a different physical register can be used for `k` for each iteration of the loop. This means that the read of `k` for `a[i,k]` when $\mathtt{k} = 2$ will access a different register than the one that is set when `k` is incremented to have the value 3.

3. Network Topologies. (25 points) Consider a messaging passing machine with 64 processors. This problem examines the routing latency for various topologies. We will assume a latency of one time unit for each network link that the message traverses. All network links are bi-directional.

Example: if the processors are arranged as an 8×8 mesh, then processor i (for $0 \leq i < 64$) will be at the mesh location $(i \div 8, i \bmod 8)$ where $i \div j$ denotes integer division (truncating), and $i \bmod j$ indicates the remainder when i is dividd by j (like `%` in C/Java or `rem` in Erlang. For example, processor 17 will be at location $(2, 1)$ and processor 42 will be at location $(5, 2)$. The minimum latency to send a message from processor 17 to processor 42 is 4 time units (three hops along one dimension, and one hop along the other).

(a) (**5 points**) If the processors are arranged as a ring, we can number the processors 0…63, so that processor i has a bi-directional link to processors $(i+1) \bmod 64$ and $(i-1) \bmod 64$. What is the minimum number of routing hops to send a message from processor 17 to processor 42?

Solution: $\min(42-17, 17+(64-42)) = \min(25, 39) = 25$.

(b) If the processors are arranged as a $4 \times 4 \times 4$ mesh, then processor i will be at location $(i \div 16, (i \div 4) \bmod 4, i \bmod 4)$.

i. (**5 points**) What are the coordinates for processors 17 and 42?

Solution: Processor 17 has grid coordinates $(17 \div 16, (17 \div 4) \bmod 4, (17 \bmod 4)) = (1, 0, 1)$. Processor 42 has grid coordinates $(42 \div 16, (42 \div 4) \bmod 4, (42 \bmod 4)) = (2, 2, 2)$.

ii. (**5 points**) What is the minimum time (shortest path) to send a message from processor 17 to processor 42?

Solution: The distance for each dimension is $(1, 2, 1)$, and the total distance is 4.

(c) If the processors are arranged as a 6-dimensional hypercube, then processor i has location $((i \div 32),\ (i \div 16) \bmod 2,\ (i \div 8) \bmod 2,\ (i \div 4) \bmod 2,\ (i \div 2) \bmod 2,\ i \bmod 2)$.

i. (**5 points**) What are the coordinates for processors 17 and 42?

Solution: The coordinates for each processor are just its 6-bit binary encoding. Processor 17 has coordinates $(0, 1, 0, 0, 0, 1)$, and processor 42 has coordinates $(1, 0, 1, 0, 1, 0)$.

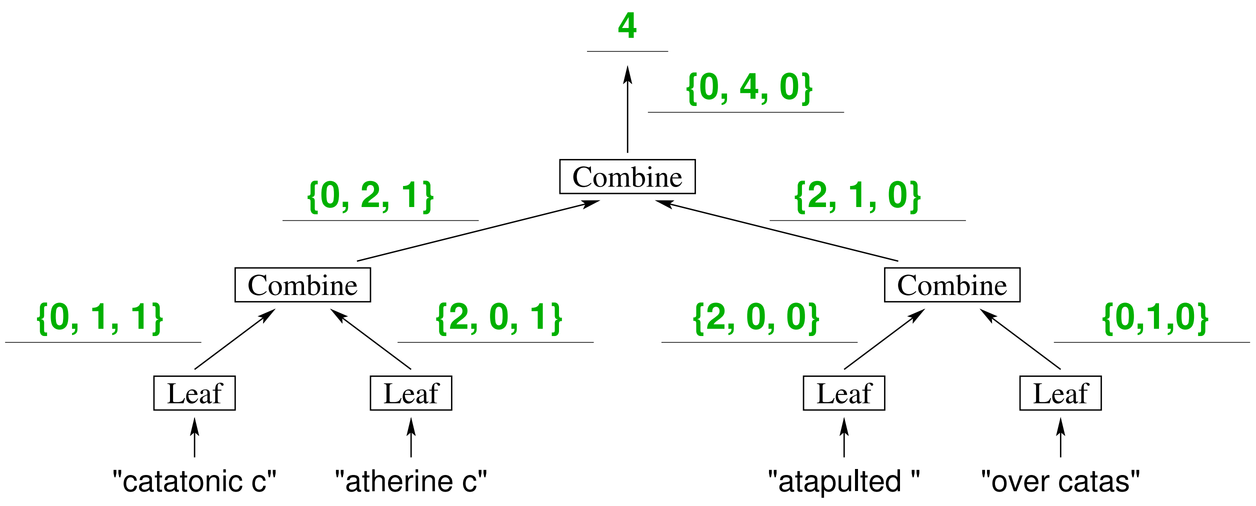
ii. (**5 points**) What is the minimum time (shortest path) to send a message from processor 17 to processor 42?

Solution: The routing distance is the number of coordinates in which the processor addresses differ: 5.

4. **Counting Cats** (25 points) Given a string, `S`, `cats(S)` is the number of occurrences of the substring `"cat"` in S. For example, if

```
S =    "catatonic catherine catapulted over catastrophic cataracts in"
    ++ "her catamaran while caterwauling scathing acatelectic scat"
    ++ "scattering cattle from catalpas and catering to categories of"
    ++ "caterpillars".
```

(a) (5 points) Complete the figure of the tree below to show how counting cats can be done in parallel. In other words, fill in the blank lines with the values that should be there.



Solution: In my solution, the value produces by `Leaf` and `Combine` is a tuple of the form {`Lcount`, `Ncats`, `Rcount`} where `Lcount` is the number of "useful" characters at the left end of the string:

`Lcount` = 1 if the string matches `[$t | Tail]`;

`Lcount` = 2 if the string matches `[$a, $t | Tail]`;

`Lcount` = 0 otherwise.

Likewise, `Rcount` is the number of "useful" characters at the right end of the string:

`Rcount` = 1 if the string ends with the character `$c`;

`Rcount` = 2 if the string ends with the characters `$c` followed by `$a`;

`Rcount` = 0 otherwise.

`Ncats` is the number of occurences of the string `"cats"` in the string for the leaf node or sub-tree. There are other reasonable solutions. For example, one could use strings such as `[]`, `"t"`, and `"at"` instead of `Lcount`, and likewise use prefixes of `"cat"` instead of `Rcount`. Any valid solution will receive full credit.

(b) (10 points) Write a stub for the `Leaf` function:

- What are the parameters to the function?

  Solution: I'll write assuming the use of the `wtree:reduce` function. Then, my `Leaf` function has the form:

  ```
  Leaf(S, Key) -> Summary
  ```

  Where `S` is the worker process state, and `Key` is the key associated with the distributed list for which we are counting cats.

- What does it return?

  Solution: The `Summary` in the code fragment above is a tuple with three elements, {`Lcount`, `Ncats`, `Rcount`} as described above.

- Briefly describe how it could be implemented.

  Solution: Get the list using `workers:get`. Use pattern matching to determine `Lcount`. Then, call a tail recursive helper function to determine `Ncats` and `Rcount`. The three sentences above are a sufficient answer. Here's the code – that would also be acceptable:

  ```
  cat_leaf(S, Key) ->
    Lcount = case workers:get(S, Key) of
      [$t | Tail] -> 1;
      [$a, $t | Tail] -> 2;
  ```

```
      Tail -> 0
    end,
    {Ncats, Rcount} = cat_leaf_help(Tail, {0, 0}),
    {Lcount, Ncats, Rcount}.
cat_leaf_help([], NR) -> NR;
cat_leaf_help([$c | Tail], {Ncats, _}) -> cat_leaf_help(Tail, {Ncats, 1});
cat_leaf_help([$a | Tail], {Ncats, 1}) -> cat_leaf_help(Tail, {Ncats, 2});
cat_leaf_help([$t | Tail], {Ncats, 2}) -> cat_leaf_help(Tail, {Ncats+1, 0})
cat_leaf_help([_ | Tail], {Ncats, _}) -> cat_leaf_help(Tail, {Ncats, 0}).
```

- Give **one** example of a call to `Leaf` and what it returns.

  Solution: Assume the string `"catatonic c"` is associate with the key `mylist` for worker state `S`. Then `cat_leaf(S, mylist) -> 0,1,1`.

(c) (10 points) Write a stub for the `Combine` function:

- What are the parameters to the function?

  Solution: `Combine(Left, Right) -> Summary` where `Left` and `Right` are tuples of the form `{Lcount, Ncats, Rcount}`.

- What does it return?

  Solution: a tuple of the form `{Lcount, Ncats, Rcount}` that summarizes the subtree below this node.

- Briefly describe how it could be implemented.

  Solution: The result `Lcount` is the `Lcount` from the `Left` tuple. The result `Rcount` is the `Rcount` from the `Right` tuple. The total number of cats is the sum of those in the left and right subtrees plus one more if the `Rcount` from `Left` plus the `Lcount` from `Right` is exactly 3. This condition means that the `Left` and `Right` strings meet with a matching prefix and suffix of `"cat"`.

- Give **one** example of a call to `Combine` and what it returns.

  Solution: `Combine({0,1,1}, {2,0,1}) -> {0,2,1}`.