CpSc 418

Midterm

February 10, 2015

100 points

Time for the exam: 50 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, iPads, 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.

Bug bounties are in effect and given in midterm points. Only report a suspected error if it affects your ability to complete the exam. To report an error, raise your hand. Due to my hearing difficulties, I may need to step in the hall to hear you – I don't understand whispers. I will post any corrections to the whiteboard.

Good luck!

CpSc 418 **Midterm** February 10, 2015

Graded out of **100 points** (101 points are possible)

Answer question 0, and any **three** out of questions 1–4. If you write solutions for all four, please indicate which three you want to have graded. Otherwise, three will be chose arbitrarily.

0. (**2** points)

   (a) Your name: ______________________________

   (b) Your student number: ______________________________

1. **Reduce** (33 points) Let's say we have a list of N numbers that is stored on P worker processes. We want to find the largest perfect square in the list. For example,

   ```
   largest_square([5622, 64, 4214, 4624, 2150, 5583, 1599, 6889, 2095])
   ```

   is 6889 (i.e. 83*83). Of course, we want to do this is parallel, and we'll use reduce.

   (a) (**25 points**): Fill-in the blanks to complete the computation of `largest_square` in Figure 1.

   (b) (**8 points**): Consider execution where the four worker processes have the specified lists: stored under key `rawdata`:

   - Worker 0: `[0, 83, 64, 5, 101]`.
   - Worker 1: `[17, 23, 164, 125, 111]`.
   - Worker 2: `[168, 169, 81, 25, 3]`.
   - Worker 3: `[0, 0, 0, 0, 0]`.

   Fill-in the blanks below to describe what happens while computing `largest_square(W, rawdata)` using the code from Figure 1, and assuming that `W` is a worker tree consisting of the four processes mentioned above. Hint: here's a list of all the squares less than 200: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196].

   - Each worker performs its leaf function:

     Worker 0: `leaf(MyList)` returns __________ ;

     Worker 1: `leaf(MyList)` returns __________ ;

     Worker 2: `leaf(MyList)` returns __________ ;

     Worker 3: `leaf(MyList)` returns __________ ;

   - The first round of combines:

     Worker ______: computes `combine(________, ________)` to produce ________ ;

     Worker ______: computes `combine(________, ________)` to produce ________ ;

   - The final result:

     Worker ______: computes `combine(________, ________)` to produce ________ ;
     This is the final result and is returned by `largest_square`.

```
largest_square(W, Key) ->
   wtree:reduce(W,
      fun(ProcState) ->
         leaf(wtree:get(ProcState, _______))
      end, fun(Left, Right) ->
         combine(Left, Right)
      end
      ).

   leaf(MyList) ->
      Squares = [
         X || X <- _____________,
                   X == square(round(math:sqrt(X)))
      ],
      case Squares of
        [] -> none;
        _  -> lists:max(_____________)
      end.


   combine(none, Right) -> ________________ ;

   combine(Left, none)  -> ________________ ;

   combine(Left, Right) -> ________________ .

   square(X) -> __________ .
```

Figure 1: Code for `largest_square` — fill it in

2. **Performance** (33 points) Consider an algorithm that takes time $t_0 N \log_2 N$ for the best sequential algorithm. Assume that a parallel version can be done in time

$$\left(t_0 \frac{N}{P} \log_2 N\right) + \left(\lambda + t_0 \frac{N}{P}\right) \log_2 P$$

Assume $t_0 = 10\text{ns}$ and $\lambda = 10\mu\text{s}$, where $1\text{ns} = 10^{-9}$seconds, and $1\mu\text{s} = 10^{-6}$seconds.

(a) **(5 points)**: What is the speed-up if $N = 2^{16} = 65536$, and $P = 256$?
Show your work on the blank pages at the end or on the back side of a test page, and write your answer here.

(b) **(6 points)**: If $P = 256$, how large must N be to get a speed-up of at least $P/2$?

(c) **(5 points)**: What is the speed-up if $N = 2^{16}$, $P = 256$, and λ is reduced to $1\mu\text{s}$ (keeping $t_0 = 10\text{ns}$)?

(d) **(5 points)**: What is the speed-up if $N = 2^{20}$, $P = 256$, and t_0 is reduced to 1ns (keeping $\lambda = 10\mu\text{s}$)?

(e) **(6 points)**: Does speed-up increase or decrease with a decrease of λ? Why?

(f) **(6 points)**: Does speed-up increase or decrease with a decrease of t_0? Why?

3. **Erlang** (33 points)

   (a) **(24 points):** Let `double(List)` return the list obtained by doubling each element of `List` Consider the three implementations below (I won't worry about guards until part b):

```
double_1([]) -> [];
double_1([Hd | Tl]) -> [2*Hd | double_1(Tl)].

double_2(List) -> double_2(List, []);
double_2([Hd | Tl], Acc) -> double_2(Tl, Acc++[2*Hd]);
double_2([], Acc) -> Acc.

double_3([]) -> [];
double_3([A]) -> [2*A];
double_3(L) ->
   {L1, L2} = lists:split(lists:length(L) div 2, L);
   double_3(L1) ++ double_3(L2).
```

   Which of these runs in $O(N)$ time? Which in $O(N \log N)$ time? and which in $O(N^2)$ time? Here, N denotes the length of the list given as an argument to `double`. Note: `lists:split(N, List) -> {FirstN, Rest}`, where `FirstN` is the first `N` elements of `List`, and `List` is the rest. You can assume that `lists:split(N, List)` runs in time $O(N)$. With each answer, give a one or two sentence justification(maybe three for the $O(NlogN)$ case). Write your answers below:

   $O(N)$: ________________________________

   Why?

   $O(N \log N)$: ________________________________

   Why?

   $O(N^2)$: ________________________________

   Why?

(b) **(3 points):** Which of the three versions of `double` from part (a) is tail recursive?

(c) **(6 points):** Here's an Erlang quirk I encountered recently:

```
1> X = [a | b].
[a|b]
2> is_list(X).
true
3> tl(X).
b
4> is_list(tl(X)).
false
```

That's right – you can have a list whose tail is defined, but whose tail is not a list! We'll say that `L` is a "true list" iff `X` is a list, and if you take `tl(X)` enough times, you eventually get `[]`. Write an Erlang function, `is_true_list(X)` that returns `true` iff `X` is a true list. For example,

```
is_true_list([]) -> true.
is_true_list([a, b]) -> true.
is_true_list(lists:seq(1, 1000000000)) -> true.
is_true_list([a | b]) -> false.
```

Write your solution below:

4. **(33 points)** Pot Pourri

   (a) **(6 points)** The best sequential implementation of a program takes 4 hours to run. A parallel version runs in 20 minutes using 16 processors. What is the speed-up?

   (b) **(Example: 0 points)** Consider the code below for the partition step of quicksort:

```
partition(A, lo, hi) {
   pivot = A[hi-1];
   i = lo;
   for(int j = lo; j < hi-1; j++) {
      if(A[j] <= pivot) {
         tmp = A[i]; A[i] = A[j]; A[j] = tmp;
         i++;
      }
   }
   A[hi-1] = A[i];
   A[i] = pivot;
}
```

   Describe a write-after-read dependency in the `partition` function.
   Answer: the write to `A[i]` in the statement '`A[i] = A[j]`' must be performed after the read of `A[i]` in the preceding statement, '`tmp = A[i]`'. Otherwise, the read will get the wrong value.

   (c) **(6 points)** Describe a read-after-write dependency in the `partition` function.

   (d) **(6 points)** Describe a control-dependency in the `partition` function.

   (e) **(6 points)** What is "super-linear speed-up"? Describe one typical cause.

(f) **(6 points)** How does a super-scalar machine determine if the register-operands for an instruction are available so the instruction can execute?

(g) **(3 points)** Who invented “Amdahl’s Law”?