CpSc 418 **Midterm** February 10, 2015

100 points

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: Mark Greenstreet

   (b) Your student number: 00000000

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 64 ;

     Worker 1: `leaf(MyList)` returns none ;

     Worker 2: `leaf(MyList)` returns 169 ;

     Worker 3: `leaf(MyList)` returns 0 ;
   - The first round of combines:

     Worker 0: computes `combine(64, none)` to produce 64 ;

     Worker 2: computes `combine(169, 0)` to produce 169 ;
   - The final result:

     Worker 0: computes `combine(64, 169)` to produce 169 ;
     This is the final result and is returned by `largest_square`.

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

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


   combine(none, Right) -> Right ;

   combine(Left, none)   -> Left ;

   combine(Left, Right) -> max(Left, Right) .

   square(X) -> X*X .
```

Figure 1: Code for `largest_square` — filled 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.

To avoid repeating calculations, I'll note that $t_p = (t_s/P) + (\lambda + t_0 N/P) \log_2 P$, where t_p is the exeuction time for the parallel program, and t_s is the sequential execution time. Note: $1\text{ms} = 10^{-3}$seconds.

$$\begin{aligned} t_s &= 10\text{ns} * 2^{16} \log_2 2^{16} \\ &= 10.048\text{ms} \\ t_p &= (10.048\text{ms}/2^8) + \left(10\mu\text{s} + 10\text{ns}\tfrac{2^{16}}{2^8}\right) \log_2 2^8 \\ &= 40.96\mu\text{s} + 100.48\mu\text{s} \\ &= 141.44\mu\text{s} \\ \mathit{SpeedUp} &= \tfrac{t_s}{t_p} \quad = \quad \tfrac{10.048\text{ms}}{141.44\mu\text{s}} \quad = \quad 74.1 \end{aligned}$$

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

Clearly bigger than 2^{16}. Let's try 2^{17}, and if that isn't enough, 2^{18} should do the job. Calculating as for part (a), I get:

N	t_s	t_p	$\mathit{SpeedUp}$
2^{17}	22.3ms	$208\mu\text{s}$	107
2^{18}	47.2ms	$346\mu\text{s}$	136

N should be $2^{18} = 262,144$ or larger.

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

$$t_s = 10.5\text{ms}, \quad t_p = 69.4\mu\text{s}, \quad \mathit{SpeedUp} = 151$$

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

$$t_s = 21.0\text{ms}, \quad t_p = 195\mu\text{s} \quad \mathit{SpeedUp} = 108$$

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

Speed-up *increases* when λ decreases because the λ term is overhead that only adds to the parallel execution time and does not affect the sequential time.

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

Speed-up *decreases* when t_0 decreases because the sequential time is proportional to t_0 but the parallel time also includes some overheads that are independent of t_0.

Note: While solving this one, I discovered that I set $N = 2^{20}$ here. Because the speed up is lower than that for $N = 2^{18}$ and $t_0 = 10\text{ns}$, you can still make the conclusion that decreasing t_0 decreases speed-up. The take-home message is that speeding up the sequential computation tends to improve performance (both t_s and t_p decrease) but *lower* speed-up.

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:

$O(N)$: `double_1`

Why?

`double_1` performs O(1) operations (a multiplication and prepending an element to a list) for each element of its argument list.

$O(N \log N)$: `double_3`

Why?

`double_3` is a divide-and-conquer approach. It spends $O(N)$ time to split the list and concatenate the results of the recursive calls. Each call reduces the length of the list by half; so, the calls go to a depth of $\lceil \log_2 N \rceil$. At each level, $O(N)$ work is done in total. This gives the $O(N \log N)$ runtime.

`double_3` is a pretty convoluted way of doubling the elements in a list – imho, there's no reason to ever do this. OTOH, its runtime is much better than `double_2`.

$O(N^2)$: `double_2`

Why?

`double_2` does $O(|\mathtt{Acc}|)$ work for the ++ operation. `double_2` is called with $|\mathtt{Acc}|$ taking on values of $0, 1, 2, \ldots, N-1$. The total time is $O(N^2)$.

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

`double_2`

(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:

```
is_true_list([]) -> true; % just like in the examples above
is_true_list([Hd | Tl]) -> is_true_list(Tl); % so far, so good
is_true_list(_) -> false.  % whatever _ is, it's not a list
```

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?

$$SpeedUp \;=\; \frac{t_s}{t_p} \;=\; \frac{4\,\texttt{hours}}{20\,\text{minutes}} \;=\; \frac{240\,\texttt{minutes}}{20\,\text{minutes}} \;=\; 12$$

(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.

The read of `j` in '`if(A[j] <- pivot)`' must be performed after write of `j` in the `j++` operation in the `for`-statement.

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

The branch for the `if`-statement must be performed before the operations that swap `A[i]` and `A[j]`.

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

Super-linear speed-up refers to a situation where a parallel program with P processes runs in less time than the time of the sequential version divided by P. This can occur because the parallel machine has more fast memory (e.g. registers, cache, DRAM) in total than a single processor, and can have a higher fraction of its data references going to faster memory. Another cause is multi-threading where several threads can make better use of the resource of a super-scalar processor than a single thread can.

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

The result register for an instruction is tagged as "busy" when it is allocated during the renaming process. It will be tagged as "ready" when the instruction has updated the register with its result. If subsequent instruction needs the value from that register, that subsequent instruction will be blocked from execution until all registers that it needs are ready (or "committed").

(g) **(3 points)** Who invented "Amdahl's Law"?

Gene Amdahl, in 1967.

Note: "Amdahl" is a sufficient answer.