Name & Student Number

CpSc 418 **Midterm** March 1, 2017

Graded out of **86 points**

This exam has 10 pages (plus the cover sheet). Please check that you have a complete exam.

Answer question 0, and any **four** out of questions 1–5. If you write solutions for all questions, please indicate which you want to have graded; otherwise the graders will select which one to discard. Note that questions 1–3 are worth 20 points each and questions 4–5 are worth 24 points each. Depending on which questions you choose to answer, your maximum score will be either 86 or 90; however, no matter which questions you choose to answer the exam will be graded on a scale of 86 points.

0. (2 points)

   Sign below to confirm that you have read and understood the exam instructions.

   I, Mark Greenstreet, confirm that I have read and understood the exam instructions and that I only need to attempt **four** questions out of questions 1–5 below. If I attempt all questions, I understand that I should clearly indicate which ones to grade. If I attempt all questions and do not indicate which ones to grade, then I accept that the graders will choose at their own convenience which to grade.

1. **Erlang** (20 points)

   The final two pages of the exam contain Erlang definitions of the functions `last`, `lastv2` and `biggest_product`, along with associated functions that are used to define these functions. You may find it convenient to tear off the last sheet of the exam so you can refer to these functions while answering this question.

   For each of these functions, state its asymptotic worst-case run time (using big-O notation) and briefly justify your claim (you do not need to provide a formal proof or recurrence).

   For your run time analysis you may use the variable N to refer to the length of the list, the variable P to refer to the size of the worker pool `W`, and variable λ to refer to the ratio between the time taken to complete a global action (such as communicating a message between workers) and the time taken to complete a local action (such as a basic arithmetic operation). You may assume that the following operations are all unit time local actions:

   - `hd`, `tl`, pattern matching
   - `+`, `-`, `*`, `/`, `abs`, `div`, `max`, `rem`
   - `<`, `=<`, `=:=`, `==`, `/=`, `=/=`, `>=`, `>`
   - `and`, `andalso`, `or`, `orelse`, `not`.

   You may also assume that `lists:foldl(Fun, Acc0, List)` takes time linear in the length of `List` plus the time for the (linear number of) calls to `Fun`.

   The possible run-time complexities are: $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(N^2)$, $\mathcal{O}(N+P)$, $\mathcal{O}(N+\lambda)$, $\mathcal{O}(NP)$, $\mathcal{O}(\lambda N)$, $\mathcal{O}((N/P)+\log P)$, $\mathcal{O}((N/P)+\lambda \log P)$, $\mathcal{O}((N/P)+\lambda)$, or $\mathcal{O}((N/P)+\lambda P)$. Not all of these complexities occur in this question.

   (a) (5 points) `last`: return the last element of a list. Asymptotic complexity and brief justification:

   $\mathcal{O}(N)$. Each recursive call to `last` takes $\mathcal{O}(1)$ time. There are N such recursive calls.
   Note: from the review problems.

   (b) (5 points) `lastv2`: another implementation of `last`. Asymptotic complexity and brief justification:

   $\mathcal{O}(N^2)$. Each recursive call to `lastv2` takes $\mathcal{O}($`length(List)`$)$ to check the "`when length(List) > 1`" part of the guard. This is done for `length(List)` $=$ N, $N-1$, ..., 1. The total time is

   $$\sum_{i=1}^{N} i \;=\; \frac{N^2+N}{2} \;\in\; \mathcal{O}(N^2)$$

   Note: from the review problems.

(c) (10 points) `greatest_product`: return the greatest product of pairs of adjacent elements in a list (parallel implementation). Asymptotic complexity and brief justification:

$O\left(\frac{N}{P} + \lambda \log P\right)$.
Each worker calls `bp_leaf` with a list of $\frac{N}{P}$ elements. `bp_leaf` does $\mathcal{O}(1)$ work plus the work of `gp(List)`. `gp(List)` calls a tail recursive version of itself. The tail-recursive `gp(List, Acc)` does $\mathcal{O}(1)$ work per recursive call – note that `my_max` takes $\mathcal{O}(1)$ time. Thus `gp(List, Acc)` and therefore `gp(List)` and `bp_leaf(List)` take $\mathcal{O}($`length(List)`$) = \mathcal{O}(N/P)$ time.
Each level of the combine tree involves one send-and-receive pair at a cost of λ and $\mathcal{O}(1)$ sequential computation. There are $\lceil \log_2 P \rceil$ levels to the tree. The total cost for the combine is $\mathcal{O}(\lambda \log P)$.
`pb_root` extracts the second element from a three element tuple. This takes $\mathcal{O}(1)$ time.
Combining the results for `bp_leaf`, `bp_combine`, and `bp_root` yields the total time of $O\left(\frac{N}{P} + \lambda \log P\right)$.
Note: We've covered this in class many times. Just writing the answer with a reference to the lecture slides is a sufficient justification. For example, "See Jan. 13 lecture slides, slide 3.".

2. `largest_gap(W, Key)` (20 points)

`largest_gap` should return the largest gap between adjacent numbers of the list of numbers stored on the worker processes of `W` and associated with `Key`. Here is the sequential version:

```
largest_gap([]) -> undef;
largest_gap([H | Tail]) ->
   {Gap, _} = lists:foldl(fun(X, {G, PrevX}) -> {my_max(abs(X-PrevX), G), X} end,
                          {undef, H}, Tail),
   Gap.
```

where `my_max` is the function defined in question 1. For example,

```
largest_gap([]) -> undef.
largest_gap([1,4,9,16,20]) -> 7.   % |16 − 9| = 7
largest_gap([1,4,9,16,2]) -> 14.   % |2 − 16| = 14
```

The parallel version of `largest_gap` is a typical reduce pattern, and you will write the functions `lg_leaf`, `lg_combine` and `lg_root` in the following implementation:

```
largest_gap(W, Key) ->
   wtree:reduce(W,
      fun(ProcState) -> lg_leaf(wtree:get(ProcState, Key)) end,
      fun(Left, Right) -> lg_combine(Left, Right) end,
      fun(Top) -> lg_root(Top) end
   ).
```

Assuming that there are N elements in the list spread evenly across P workers in the worker pool `W` (where $P \ll N$), your implementation should achieve a speedup close to P. Your implementation should fail if there is no value associated with `Key` or if the value associated with `Key` is not a list of numbers.

You may call any of the functions defined in question 1, the sequential `largest_gap(List)` function defined above, any Erlang built-in functions, and any functions from the `lists`, `misc`, `workers`, or `wtree` modules.
Hint: Some functions from question 1 may prove inspirational even if they cannot be called directly.

(a) (8 points) Your code for `lg_combine(Left, Right)`:

```
% All three parts of this problem can be solved by copying the code
% for biggest_product from Q1 with a few small changes.
lg_combine(empty, Right) -> Right; % adapted from bp_combine
lg_combine(Left, Empy) -> Left;
lg_combine({LL, LBig, LR}, {RL, RBig, RR}) ->
   {LL, my_max([LBig, abs(LR-RL), RBig]), RR}.  % my_max from Q1.
```

(b) (8 points) Your code for `lg_leaf(Data)`:

```
% adapted from bp_leaf.
% largest_gap(List) is from the problem statement.
lg_leaf([]) -> empty;
lg_leaf(List) -> {hd(List), largest_gap(List), tl(List)}.
```

(c) (4 points) Your code for `lg_root(Top)`:

```
lg_root(V) -> bp_root(V).
```

3. **Message Passing Network Topologies** (20 points)

We are building a message passing machine with 64 nodes, and we need to decide how to connect its network. For each of the following topologies, specify the bisection bandwidth, diameter and maximum number of network ports needed by a node.

(a) (3 points) Ring.

bisection bandwidth: 2

diameter: 32

network ports: 2

(b) (3 points) 2D mesh.

bisection bandwidth: 8

Note: $\sqrt{P}$ assuming a simple mesh, not toroidal because the problem didn't use the word "toroidal". Draw a line that separates the left and right halves of the mesh.

diameter: 14

Note: 7 hops in each dimension.

network ports: 4

Note: connections to north, south, east, and west neighbours.

(c) (3 points) 3D mesh.

bisection bandwidth: 16

Note: $\sqrt[3]{P}$ assuming a simple mesh, not toroidal because the problem didn't use the word "toroidal". Draw a plan that separates the top half of the mesh from the bottom half.

diameter: 9

Note: 3 hops in each dimension.

network ports: 6

Note: same connections as above, plus up and down.

(d) (3 points) 6D hypercube.

bisection bandwidth: 32

diameter: 6

Note: example, node 0 to node 63.

network ports: 6

Note: one for each bit of the node index.

(e) (4 points) State **two** advantages of the 6D hypercube over the 2D mesh? Answers which give more than two will receive zero.

- Small diameter → low routing latency.
- High bisection → fewer communication bottlenecks.
- Note: for the solution set, I'll include another response that gets full credit:
  - No special cases at edges.

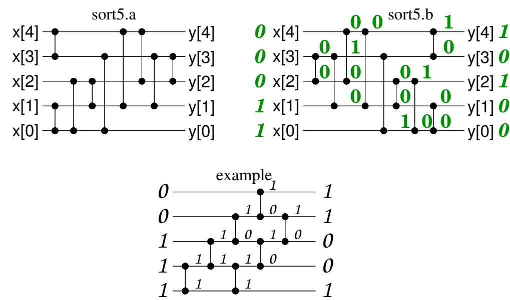

Figure 1: Sorting Networks

(f) (4 points) State **two** advantages of the 2D mesh over the 6D hypercube? Answers which give more than two will receive zero.

- Easily mapped to 2D technologies such as chips or circuit boards.
- Bounded degree nodes make the design more modular.
- Note: for the solution set, I'll include another response that gets full credit:
  - Doesn't become "all wire" for networks with a large number of processors.

4. **Sorting Networks** (24 points)

Figure 1 shows two sorting networks with five inputs: sort5.a, and sort5.b. One of these networks sorts correctly and one does not.

(a) (10 points) Identify which of the networks **does not** sort correctly. Label the inputs with values of **0** or **1** that the network does not sort correctly. Label the output of each compare-and-swap with its value for this input, and label the outputs of the sorting network with their values. The "example" network provides an example of such a labeling (and also an incorrect attempt at sorting).

Which network do you claim is incorrect (circle one): sort5.a (sort5.b)

Note: The counter-example has 1s for `x[0]` and `x[1]` and 0s for `x[2]`, `x[3]` and `x[4]`. The compare-and-swap (1,3) moves the 1 for `x[1]` into row 3. Note that the only path for the 1 for `x[0]` to rows 3 or 4 is through the compare-and-swap (0,3). However, it is blocked by the 1 from `x[1]` that has already moved to row 3. This shows that the network cannot sort this input correctly.

Remember to label that network with a counter-example in the figure above.

(b) (14 points) For the other network, explain specifically how it sorts any combination of three **0**s and two **1**s correctly.

We observe that the three compare-and-swap modules in the bottom left, i.e. (0,1), (0,2), and (1,2) sort rows 0, 1, and 2 into ascending order. Likewise, the compare-and-swap module in the top-left, i.e. (3,4), sorts rows 3 and 4 into ascending order. Call these four compare-and-swap operations "the initial compare and swaps"

Now, consider the number of 1s in $\{$`x[3]`, `x[4]`$\}$.

zero 1s: In this case, the values of rows 0 to 4 after the initial compare and swaps must be (0,1,1,0,0). The 1 in row 2 moves to row 4 with the compare-and-swap (2,4), and the 1 in row 1 moves to row 3 with the compare-and-swap (1, 3). Thus, the two ones make it to the top two rows.

one 1: In this case, the values of rows 0 to 4 after the initial compare and swaps must be (0,0,1,0,1). The 1 in row 2 moves to row 3 with the final compare-and-swap (2,3).

two 1s: In this case, the values of rows 0 to 4 after the initial compare and swaps must be (0,0,0,1,1). The data are already sorted, and all of the compare-and-swaps preserve this order.

Compare-and-swap operations preserve the total number of 1s in the network, and the total number of 0s. Showing that the two 1s arrive at `y[3]` and `y[4]` implies that `y[0]` = `y[1]` = `y[2]` = 0.

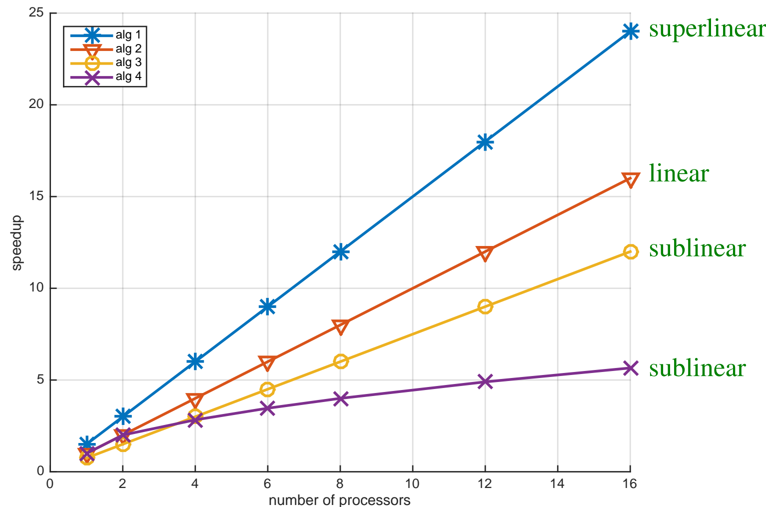

Figure 2: Speedup for some parallel algorithms.

5. **Short Answer** (24 points)

(a) (7 points) Figure 2 shows the speedup measured for four algorithms: alg 1, alg 2, alg 3 and alg 4. In the space to the right of the plot, label each curve as either *sublinear*, *linear* or *superlinear* speedup. Not all speedups may be present in the plot.

Give **two** general reasons why parallel programs sometimes display sublinear speedup *other than computational or communication overhead*. Answers which give more than two reasons will receive zero.

- Non-parallelizable code.
- Extra computation.
- Note: for the solution set, I'll include other responses that get full credit:
  - memory overhead
  - synchronization overhead
  - resource contention
  - idle processors

Give **one** general reason why parallel programs sometimes display superlinear speedup. Answers which give more than one reason will receive zero.

- more fast memory (in total)
- Note: the "fast memory" answer is the right explanation 99% of the time. occasionally, a problem is so naturally parallel that the sequential algorithm has control overhead to serialize the operations that the parallel code avoids. A common explanation is "invalid comparison" because there's something wrong with the way the times were measured, but I don't think that's an intended answer.

(b) (6 points) For sequential computing there is the successful and near ubiquitous Von Neumann architecture; in the early days of computing there were other sequential architectures, but they have essentially disappeared from use. In contrast, we have discussed three very different approaches to achieving parallel computation that are still widely used, often in combination with one another: superscalar, shared memory and message passing. For **each** of the three, give **one** reason why it has **not** become dominant over the others; in other words, one significant disadvantage. Note that your reasons need not be unique. Answers which give more than one reason for any approach will have marks deducted.

superscalar: Limited instruction level parallelism.

shared memory: Poor scaling of coherence models to large numbers of processors.

message passing: Network delays.

(c) (6 points) You should remember from your algorithms course that sequential merge sort requires $\mathcal{O}(N \log N)$ time and $\mathcal{O}(N \log N)$ comparisons for N elements. We saw that under reasonable assumptions a bitonic sorting network requires $\mathcal{O}((\frac{N}{P})(\log N + \log^2 P))$ time and $\mathcal{O}(N \log N \log^2 P)$ comparisons for N elements with P processors. Assume now that we have M (where $M \gg P$) independent sorting problems each of size N. In this question we consider two methods of solving these M problems: either connecting the P processors together into a single bitonic sorting network and running the problems through the network one at a time, or running up to P independent sequential merge sorts at a time. We will assume in each case that the data is already distributed in a manner which is optimal for that case. Is the *latency* of a bitonic sorting network better, worse or about the same as independent sequential merge sorts? Is the *throughput* of a bitonic sorting network better, worse or about the same as independent sequential merge sorts? Briefly explain your answers.

Bitonic latency (circle one and briefly explain): (BETTER) WORSE SAME

The latency for parallel bitonic sort is $\mathcal{O}(\frac{N}{P} \log N \log^2 P)$ which is less than the latency for sequential merge sort, $\mathcal{O}(N \log N)$ when $N \gg P$.

Note: I found this one a little confusing. The problem statement says

> "Assume now that we have M (where $M \gg P$) independent sorting problems each of size N."

Is latency the time to solve one of these problems are all M of them? I can suggest rephrasings of the problem that would make this unambiguous, but it's a bit late for that.

Bitonic throughput (circle one and briefly explain): BETTER (WORSE) SAME

With $M \gg P$, we can keep P processors busy working on independent merge sort problems. Thus, the bottleneck is the execution time of the processors. Sequential merge sort requires fewer total processor cycles than bitonic sort *and* it avoids the communication costs of the parallel algorithm. Thus, sequential merge sort will achieve higher throughput.

(d) (5 points) We developed a tree-based reduce operation in Erlang—a language which assumes a message passing architecture, although it can be run on either message passing or shared memory hardware—to speed up parallel computations that produce a small amount of summary information at a single process from a large amount of data spread across many processes. If we were to use a language and hardware which supported shared memory, is a tree-based reduce still useful to speed up parallel computation? Briefly explain your answer. (If it is relevant to your answer, you may assume an idealized memory where access to any item costs the same amount for any process.)

The tree based algorithm is still faster. We need some kind of synchronization between the threads, and that comes at a cost of λ per synchronization. A tree has a cost of $\lambda \log P$ for synchronization. If a single master processor synchronized with each worker, the synchronization cost would be λP.

Note: I'm a bit concerned about "idealized memory". If I have atomic reads and writes that take $\mathcal{O}(1)$ time, then I can implement a mutual-exclusion algorithm (such as Peterson's) that provides lock acquistion or transfer in $\mathcal{O}(1)$ time. This removes the λs from the analysis. In this case, a

tree-algorithm takes time $\mathcal{O}(\log P)$ and a singel master process takes tiem $\mathcal{O}(P)$. Again, the tree is faster, but it is likely that P is much less than the time for the worker processes to do their tasks before meeting at a reduce.

Functions for Q1a and Q1b

Do not write your answer on this page—it will not be graded. You may find it convenient to tear this page off when answering questions 1 and 2. If you tear it off, you need not submit it with the rest of your exam.

```
%%%%%%%%%   Q1.a
% last(List) -> LastElement.
%   Return the last element of List. If List is not
%   a list or if List is empty, then an error will be thrown.
%   Example:
%     last([1, 4, 9, 16]) -> 16.
%     For run-time analysis, N = length(List).
last([E]) -> E;
last([_ | T]) -> last(T).
```

```
%%%%%%%%%   Q1.b
% lastv2(List) -> LastElement.
%   Another implementation of last(List).
%     For run-time analysis, N = length(List).
lastv2(List) when length(List) > 1 -> lastv2(tl(List));
lastv2([X]) -> X.
```

Repeated from the question 1 problem statement for your convenience:

For your run time analysis you may use the variable N to refer to the length of the list, the variable P to refer to the size of the worker pool `W`, and variable λ to refer to the ratio between the time taken to complete a global action (such as communicating a message between workers) and the time taken to complete a local action (such as a basic arithmetic operation). You may assume that the following operations are all unit time local actions:

- `hd`, `tl`, pattern matching
- `+`, `-`, `*`, `/`, `abs`, `div`, `max`, `rem`
- `<`, `=<`, `=:=`, `==`, `/=`, `=/=`, `>=`, `>`
- `and`, `andalso`, `or`, `orelse`, `not`.

You may also assume that `lists:foldl(Fun, Acc0, List)` takes time linear in the length of `List` plus the time for the (linear number of) calls to `Fun`.

The possible run-time complexities are: $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$, $O(N+P)$, $O(N+\lambda)$, $O(NP)$, $O(\lambda N)$, $O((N/P) + \log P)$, $O((N/P) + \lambda \log P)$, $O((N/P) + \lambda)$, or $O((N/P) + \lambda P)$. Not all of these complexities occur in this question.

Function for Q1c

Do not write your answer on this page—it will not be graded. You may find it convenient to tear this page off when answering questions 1 and 2. If you tear it off, you need not submit it with the rest of your exam.

```
%%%%%%%%%    Q1.d
% greatest_product(W, Key) -> Number.
%   W is a worker-tree. Key is the key for a list of numbers distributed
%   across the workers We return the largest product of adjacent numbers.
%   If the list is empty or a singleton, then greatest_product returns 'undef'.
%   If there is no value associated with Key, or if the value associated with
%   Key is not list of numbers , then greatest_product fails.
%   Example:
%     If Key corresponds to the list [1, 2, 7, 3, 5, 4, 2, -8, 6, 3],
%     Then the products of adjacent numbers are
%       [2, 14, 21, 15, 20, 8, -16, -48, 18],
%     and the largest product of adjacent numbers is 21.
%   For run-time analysis: let N denote the total length of the list
%   corresponding ot Key; let P denote the number of processes.
%   Assume that the list is divided into equal sized segments.
%   Assume that there are enough processors so that all processes
%   can run at the same time.
greatest_product(W, Key) ->
   wtree:reduce(W,
      fun(ProcState) -> bp_leaf(wtree:get(ProcState, Key)) end,
      fun(Left, Right) -> bp_combine(Left, Right) end,
      fun(Top) -> bp_root(Top) end
   ).

bp_leaf([]) -> empty;
bp_leaf(List) -> {hd(List), gp(List), last(List)}.

bp_combine(empty, Right) -> Right;
bp_combine(Left, empty) -> Left;
bp_combine({LL, LBig, LR}, {RL, RBig, RR}) ->
   {LL, my_max([LBig, LR*RL, RBig]), RR}.

bp_root({_, Big, _}) when is_number(Big) -> Big;
bp_root(_) -> undef.

% gp(List) -> Number.
%   a sequential version of greatest_product.
gp(List) -> gp(List, undef).
gp([], BigProd) -> BigProd;
gp([_], BigProd) -> BigProd;
gp([A | Tail = [B | _]], BigProd) ->
   gp(Tail, my_max(A*B, BigProd)).

% my_max: calculate max where 'undef' is less thn any number
my_max(undef, Y) -> Y;
my_max(X, undef) -> X;
my_max(X, Y) -> max(X, Y).

% my_max for lists.
my_max(List) -> lists:foldl(fun(X, Acc) -> my_max(X, Acc) end, undef, List).
```