Name & Student Number

CpSc 418

Midterm Solution Set

October 25, 2017

Graded out of **100 points**

You were asked to answer question 0 and any **four** out of questions 1–5. If you wrote solutions to all questions and did not specify which should be graded, we chose which one to discard.

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.

   Because this is the solution set, we'll provide answers to all of the questions.

1. **Erlang** (25 points) Erlang provides bitwise operation on integers. For example, `X bor Y` computes the bitwise-OR of integers `X` and `Y`; the result is an integer. Here are a few examples – in Erlang `2#0101` is the way we can write an integer in binary, and `2#0101 == 5`.

```
2#0101 bor 2#1100 -> 2#1101.
2#0001 bor 2#1000 -> 2#1001.
```

   The following function computes the bitwise-OR of all of the elements of a list:

```
bit_or([]) -> 0;
bit_or([Hd | Tl]) -> Hd bor bit_or(Tl).
```

   (a) **(5 points)** Is `bit_or` head-recursive or tail-recursive? Briefly justify your answer.

   Head recursive. The recursive call is made while there is still a pending operation, `bor`, for the current call of the function.

   (b) **(5 points)** Write a second implementation that uses the other style – i.e. if `bit_or` is head-recursive, write a tail-recursive version. If it is tail-recursive, write a head-recursive version.

```
bit_or(List) -> bit_or(List, 0).
bit_or([], Acc) -> Acc;
bit_or([Hd | Tl], Acc) -> bit_or(Tl, Hd bor Acc).
```

   (c) **(10 points)** Write an implementation of `bit_or` using `lists:foldl`.

```
bit_or(List) ->
  lists:foldl(fun(X,Y) -> X bor Y end, 0, List).
```

   (d) **(5 points)** The *cummulative-OR* of a list replaces each element of a list with the bitwise-OR of everything up-to-and including that element. For example,

```
cummulative_bor([2#0001, 2#1000, 2#1100, 2#0101]) ->
                [2#0001, 2#1001, 2#1101, 2#1101].
```

   Write an implementation of `cummulative_bor` using `lists:mapfoldl`.

element number i	element bits	element integer	parity $0 \dots i$ bits	parity $0 \dots i$ integer
1	10101010	170	10101010	170
2	00001111	15	10100101	165
3	11111111	255	01011010	90
4	00000000	0	01011010	90
5	10000000	128	11011010	218
6	00111100	60	11100110	230

Table 1: Example of computing the parity of a sequence of bytes.

```
cummulative_bor(List) ->
  {Ans, _} = lists:mapfoldl(
    fun(X, Y) -> Z = X bor Y, {Z, Z} end,
    0, List),
  Ans.
```

.

2. **Reduce / Scan** (25 points)

The *parity* of a sequence of bits is 1 if the number of ones in the sequence is odd, otherwise it is 0. Put another way, if the parity is treated as an extra bit at the end of a sequence, the augmented sequence will always have an even number of ones. The notion of parity can be extended to sequences of bytes (8 bits) by separately computing the parity of all of the first bits, all of the second bits, ..., all of the eighth bits (the parity of a sequence of bytes will be a byte). Table 1 shows an example sequence of bytes $i = 1 \dots 6$ (in both their bitwise and integer representation) and the parity bytes computed over elements $0 \dots i$ (in both their bitwise and integer representation).

The parity of a sequence of bits can be computed with the *exclusive or* (often called "xor") operator. In Erlang, we can do this to bytes or words using the `bxor` operator. For example, `170 bxor 15 = 165` and `165 bxor 255 = 90`. Compare these results to the integer columns of the first three rows of the sequence in table 1. Exclusive or and Erlang's `bxor` are both associative and commutative.

You will complete the helper functions for `compute_parity(W, ListKeyIn, ListKeyOut)`, where

- `W` is a list of worker nodes arranged in a tree.
- `ListKeyIn` is a Key with which each worker node can retrieve its local portion of the input list from its `ProcState`. The input list will be bytes of data represented as integers in the range $[0, 255]$.
- `ListKeyOut` is a Key under which each worker node should store its local portion of the output list in its `ProcState`. The output list will be parity bytes represented as integers in the range $[0, 255]$.

The return value of `compute_parity()` should be the parity byte of the entire list represented as an integer in the range $[0, 255]$.

We implement `compute_parity()` with the following call to `wtree:scan()`:

```
compute_parity(W, ListKeyIn, ListKeyOut) ->
  wtree:scan(W,
    fun(ProcState) -> cp_leaf1(wtree:get(ProcState, ListKeyIn)) end,
    fun(ProcState, AccIn) ->
      ListIn = wtree:get(ProcState, ListKeyIn),
      ListOut = cp_leaf2(ListIn, AccIn),
      wtree:put(ProcState, ListKeyOut, ListOut)
    end,
    fun(Left, Right) -> cp_combine(Left, Right) end,
    0
    ).
```

As an example, imagine that `W` contains two workers with the following data:

- On worker 1, `ProcState` contains the Key/Value pair `{listIn, [ 170, 15, 255, 0 ]}`.
- On worker 2, `ProcState` contains the Key/Value pair `{listIn, [ 128, 60 ]}`.

Then the call to `compute_parity(W, listIn, listOut)` will return the value `230` and the following additional data will be stored on the workers after the scan completes:

- On worker 1, `ProcState` will now contain the Key/Value pair `{listOut, [ 170, 165, 90, 90 ]}`.
- On worker 2, `ProcState` will now contain the Key/Value pair `{listOut, [ 218, 230 ]}`.

Assuming that there are N elements in the input 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 `ListKeyIn` or if the value associated with `ListKeyIn` is not a list of integers.

You may call any Erlang built-in functions, and any functions from the `lists`, `misc`, `workers`, or `wtree` modules.

(a) (4 points) Briefly describe the segment summary that you plan to use.
The bitwise-xor of all the elements for the segment.

(b) (4 points) Your code for `cp_combine(Left, Right)`:

```
cp_combine(Left, Right) -> Left bxor Right.
```

(c) (8 points) Your code for `cp_leaf1(ListIn)`:

```
cp_leaf1(ListIn) -> lists:foldl(fun(X,Y) -> X bxor Y end, 0, ListIn).
```

(d) (8 points) Your code for `cp_leaf2(ListIn, AccIn)`:

```
cp_leaf2(ListIn, AccIn) ->
  {Ans, _} = lists:mapfoldl(
    fun(X, Y) -> Z = X bxor Y, {Z, Z} end,
    AccIn, List),
  Ans.
```

3. **Architecture and other stuff** (25 points)

(a) **(5 points)** The statements below describe the contents of caches for a shared memory multiprocessor using the MESI protocol. Mark each statement below T for true or F for false.

F Only one cache can have valid data for a particular memory location at any given time.

T Multiple caches can have read-only copies of the data for the same memory location at the same time.

F Multiple caches can have write-only copies of the data for the same memory location at the same time.

F Multiple caches can have read-and-writable copies of the data for the same memory location at the same time.

F If a value in a cache is valid (i.e. can be read or written by the cache's processor), then it matches the value in main memory.

(b) **(7 points)** Mark each statement below T for true or F for false.

F A parallel implementation of an algorithm can be faster than the sequential version or use less energy, but not both as the same time.

T A parallel implementation of an algorithm can be faster than the sequential version and use less energy.

F Parallel algorithms are always faster than their sequential counterparts.

F Parallel algorithms are always slower than their sequential counterparts.

T A CPU can consume less energy per operation when operating at a lower frequency.

T Power is energy per unit time.

we'll accept either T or F Knowledge is power.

(c) **(5 points)** For each message-passing network below, write whether the cross-section bandwidth grows as 1, $\log N$, N, N^2, or N^3. We're ignoring constant factors.

N A $N \times N$ mesh (i.e. two dimensional)

N^2 A $N \times N \times N$ torus (i.e. three dimensional)

1 A N processor ring.

1 A binary tree with N processors at the leaves – the non-leaf nodes

N A d-dimensional hypercube with $N = 2^d$ processors.

(d) **(8 points)** Answer each of the questions below with a single sentence or short phrase.

- **(2 points)** What does acronym "RAM" mean in the RAM model?

  Random Access Machine

- **(2 points)** What does acronym "PRAM" mean in the PRAM model?

  Parallel Random Access Machine

- **(2 points)** What does λ represent in the CTA model. Note: the CTA model is the model we use most often in class.

  Communication delay and overhead

- **(2 points)** Should chip manufacturers pay a fine if they don't comply with Moore's Law?.
  YES: I want a computer with a 1000GHz clock.
  NO: Things would get too weird if they break that many laws of physics.

4. **Performance** (25 points)

A common operation in iterative numerical computing (such as image processing, solving differential equations, etc.) is to take an array of data and update each element of the array with a function that depends on the values of that element and its neighbors. For this question we will consider the simplest form of such an update. We will work on a 2D square array of size $N \times N$, so there are N^2 elements. Each element will be updated using the values of that element and the four neighbours above, below, left and right of that element (the value of any neighbor beyond the edge of the array can be treated as zero). If we define a unit of time to be the time necessary to perform an update to a single element (assuming that the values of all four neighbours are known locally), then a serial implementation takes N^2 time to update all of the elements of the array.

Now we will consider creating a parallel version on P processors. We will assume a fully connected network, so each processor can communicate directly with every other processor. To complete an update, each processor will need a copy of the data for any elements neighboring its own elements. Assume that sending or receiving K elements takes time $\lambda + K$, but that messages between different pairs of processors can be overlapped and each link can carry messages in both directions at once; for example, if processor 1 sends a message of length $K_{1\to2}$ to processor 2 and a message of length $K_{1\to3}$ to processor 3 while receiving a message of length $K_{2\to1}$ from processor 2 and a message of length $K_{3\to1}$ from processor 3, the total time spent by processor 1 will be $\lambda + \max(K_{1\to2}, K_{1\to3}, K_{2\to1}, K_{3\to1})$.

(a) (5 points) Our first implementation will divide the array up by columns, so each processor will get N/P complete and contiguous columns (each with N elements).

i. For a processor working on some interior columns of the array (in other words, it has neighbors on both sides), how much data must it send and receive before all the processors can perform their updates?
There are two reasonable answers depending on how you interpreted the question. Interpretation 1: $2N$ is the amount of data sent and also the amount of data received (since the same

amount is outgoing and incoming). Interpretation 2: $4N$ is the total amount of data either sent or received.
Note that we did not understand the reasoning behind the second interpretation until after the in-class midterm was graded, so students who were given 1 out of 2 points for answering $4N$ can submit for a regrade.

ii. How much time will be spent for this communication?
$\lambda + N$. Note that this answer is the same regardless of the interpretation you adopted in the previous part.

iii. How much time will be spent on computation to complete the update?
N^2/P. This is the time taken for a single processor and also the time taken to complete the whole update since the processors can work independently on this step. We also accepted an answer that included the communication time from the previous part.

(b) (5 points) Our second implementation will divide the array up into square blocks so that each processor gets $B = N^2/P$ elements arranged in a square (you may assume that B is a perfect square).

i. For a processor working on some interior block of the array (in other words, it has neighbors on all four sides), how much data must it send and receive before all the processors can perform their updates?
$4\sqrt{B}$ or $8\sqrt{B}$ depending on your interpretation of "send and receive" as described above.

ii. How much time will be spent for this communication?
$\lambda + \sqrt{B}$

iii. How much time will be spent on computation to complete the update?
N^2/P

(c) (3 points) Are there any conditions on N, P and/or λ under which the column partition would outperform the block partition? If yes, give the condition. If no, briefly explain why not.
No, the block partition is always faster. Both have the same computation time. The communication time for the column parition is $\lambda + N$ whereas the communication time for the block partition is $\lambda + N/\sqrt{P}$. We are interested in *parallel* implementations whch means $P > 1$. Therefore $\sqrt{P} > 1$ and $N/\sqrt{P} < N$. The communication cost for the block partition is always smaller than that for the column partition.

(d) (4 points) Compute the speedup for the **block partitioned** version with:

i. $N = 200$, $P = 100$, $\lambda = 100$.

Sequential time: $T_{seq} \; = \; N^2 \; = \; 4 \cdot 10^4$.

Parallel time: $T_{par} \; = \; \frac{N^2}{P} + \lambda + \frac{N}{\sqrt{P}} \; = \; 520$

SpeedUp: $\frac{T_{seq}}{T_{par}} \; \approx \; 77$

ii. $N = 1000$, $P = 400$, $\lambda = 1000$.

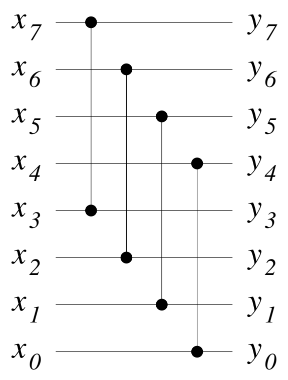
Sequential time: 10^6.

Parallel time: 3550.

SpeedUp: approximately 280.

(e) (4 points) Will the **block partitioned** algorithm display the behavior described by Amdahl's law? Briefly explain.
Yes. If we fix N, then the communication overhead is $\lambda + N/\sqrt{P}$. The term λ does not decrease with P and Amdahl could have called this the sequential code. For fixed N, Amdahl's Law applies.

(f) (4 points) Will the **block partitioned** algorithm display the behavior described by Gustafson's law? Briefly explain.
Yes. If we allow N to grow, then the communication overhead decreases as a fraction of the total time for the computation. This allows N and P to both grow and achieve speed-ups that are close to P for large enough N.



a. The Sorting Network

i. $j < i + (N/2)$		**ii.** $j = i + (N/2)$		**iii.** $j > i + (N/2)$	
i=3,	*j=6*	*i=1,*	*j=5*	*i=1,*	*j=6*
$x_7 = 0,$	$y_7 = 1$	$x_7 = 0,$	$y_7 = 1$	$x_7 = 0,$	$y_7 = 1$
$x_6 = 0,$	$y_6 = 0$	$x_6 = 0,$	$y_6 = 1$	$x_6 = 0,$	$y_6 = 1$
$x_5 = 1,$	$y_5 = 1$	$x_5 = 0,$	$y_5 = 1$	$x_5 = 1,$	$y_5 = 1$
$x_4 = 1,$	$y_4 = 1$	$x_4 = 1,$	$y_4 = 1$	$x_4 = 1,$	$y_4 = 1$
$x_3 = 1,$	$y_3 = 0$	$x_3 = 1,$	$y_3 = 0$	$x_3 = 1,$	$y_3 = 0$
$x_2 = 0,$	$y_2 = 0$	$x_2 = 1,$	$y_2 = 0$	$x_2 = 1,$	$y_2 = 0$
$x_1 = 0,$	$y_1 = 0$	$x_1 = 1,$	$y_1 = 0$	$x_1 = 1,$	$y_1 = 1$
$x_0 = 0,$	$y_0 = 0$	$x_0 = 0,$	$y_0 = 0$	$x_0 = 0,$	$y_0 = 0$

b. Examples (you need to fill in ii and iii)

Figure 1: Sorting Network for Question 5b

5. **Bitonic Sequences** (25 points)

(a) **(6 points)** Consider the six sequences below. Which of them are monotonic. Your answer can just list the numbers for the monotonic ones (e.g. *i*, *ii*, etc.).

i.	[0, 0, 1, 1, 1, 1, 1, 0]
ii.	[0, 1, 0, 1, 0, 1, 0, 1]
iii.	[1, 1, 1, 1, 1, 1, 0, 0]
iv.	[1, 1, 1, 1, 1, 1, 1, 1]
v.	[1, 2, 3, 4, 3, 2, 1, 2]
vi.	[0, 1, 1, 2, 3, 5, 8, 0]

Answer: Only iii and iv are monotonic (as asked during the in-class version). Only i, iii, iv and vi are bitonic (as asked during the take-home version).

For the remainder of this problem, let i, j, and N be integers with $0 \leq i \leq j \leq N$. Let x_0, x_1, ..., x_{N-1} be the bitonic sequence with

$$\begin{aligned} x_k &= 1, \quad \text{if } i \leq k < j \\ &= 0, \quad \text{otherwise} \end{aligned} \tag{1}$$

In English, x is a sequence of the form zero or more 0s followed by zero or more 1s followed by zero or more 0s; i is the index of the first 1; and j is the index of the first 0 following the 1s.

Let y_0, y_1, ..., y_{N-1} be the sequence defined by:

$$\begin{aligned} y_k &= \min(x_k, x_{k+(N/2)}), \quad \text{if } 0 \leq k < (N/2) \\ &= \max(x_{k-(N/2)}, x_k), \quad \text{if } (N/2) \leq k < N \end{aligned} \tag{2}$$

This is the generalization of Figure 1.a for arbitrary, even N.

(b) **(6 points)** For parts ii and iii of Figure 1.b, choose values of i and j that satisfy the given constraint – for these examples $N = 8$. Then, fill in the values for x and k according to Equations 1 and 2 above. We've completed part i of the figure to give you a worked example. If you want to save time writing, you can just write tall, skinny $\Bigg0$s and $\Bigg1$s that span several elements of a sequence for x or y.

See Figure 1.

(c) **(8 points)** Show that if x is bitonic, then either $y_k = 0$ for $0 \leq k < (N/2)$ or $y_k = 1$ for $(N/2) < k < N$ (or both). It is sufficient to prove this for $j \leq i+(N/2)$ and $j > i+(N/2)$. We've provided the proof for the $j \leq (N/2)$ case as an example so you know that you don't need any more detail than what we wrote. The $j > i+(N/2)$ case is similar.

Case $j \leq i+(N/2)$: I'll show that $y_0 \ldots y_{(N/2)-1}$ are all 0. Consider k with $0 \leq k < (N/2)$. If $k < i$ then $x_k = 0$ and $y_k = \min(x_k, x_{k+(N/2)})$ is 0. Otherwise,

$$j-(N/2) \;\leq\; i \;\leq\; k < (N/2),$$

$x_{k+(N/2)} = 0$, and therefore $y_k = \min(x_k, x_{k+(N/2)}) = 0$.

Case $j > i+(N/2)$: I'll show that $y_{N/2} \ldots y_{N-1}$ are all 1. Consider k with $(N/2) \leq k < N$. If $k < j$ then $x_k = j$ and $y_k = \max(x_k, x_{k+(N/2)})$ is 1. Otherwise,

$$i < j-(N/2) \;\leq\; k-(N/2) < (N/2),$$

$x_{k-(N/2)} = 1$, and therefore $y_k = \max(x_k, x_{k+(N/2)}) = 1$.

(d) **(5 points)** Show that the other half of y is bitonic. In other words, if $y_0 \ldots y_{(N/2)-1}$ are all 0, then $y_{N/2} \ldots y_{N-1}$ form a bitonic sequence, and if $y_{N/2} \ldots y_{N-1}$ are all 1, then $y_0 \ldots y_{(N/2)-1}$ form a bitonic sequence. My proof uses the same two cases as in question 5c.

Case $j \leq i+(N/2)$:

$$\begin{array}{rcll} y_k &=& 1, & \text{if } (N/2) \leq k < j \\ &=& 0, & \text{if } j \leq k < i+(N/2) \\ &=& 1, & \text{if } i+(N/2) \leq k < N \end{array}$$

Therefore $y_{N/2} \ldots y_{N-1}$ is bitonic.

Case $j \geq i+(N/2)$:

$$\begin{array}{rcll} y_k &=& 0, & \text{if } 0 \leq k < i \\ &=& 1, & \text{if } i \leq k < j-(N/2) \\ &=& 0, & \text{if } j-(N/2) \leq k < (N/2) \end{array}$$

Therefore $y_0 \ldots y_{N/2}$ is bitonic.