CpSc 418 **Final Exam** December 5 2013

Solution set

1. Reduce and Scan (**24 points**) One of the problems below can be solved using reduce, the other can be solved using scan. Identify which is which. For the problem that can be solved using reduce, describe the `Leaf` and `Combine`, and `Root` functions – the `Root` function may (or may not) be the identity function. For the problem that can be solved using scan, describe the `Leaf1` and `Leaf2`, and `Combine` functions.

   (a) (**12 points**) Given a list of numbers that is distributed across the worker processes, compute a new list whose i^{th} element is the maximum element up to and including the current one minus the minimum element up to and including this one. I'll call this function `gap` and here are a few examples:

```
gap([5,2,6,18,-3,4,8,-1,20]) -> [0,3,4,16,21,21,21,21,23];
gap([]) -> [];
gap([42]) -> [0];
```

   Solution The problem is an instance of scan. In my solution, the `Leaf1` function reads the list and returns a tuple of the form `{Min, Max}` where `Min` is the minimum element of the list and `Max` is the maximum element. If the list is empty, `Leaf1` returns the atom `'empty'`. The `Combine` function takes two such tuples (or `'empty'` atoms) and returns the maximum and minimum for the combined subtrees (or `'empty'` if both are empty). The `Leaf2` function takes an `AccIn` value that is the tuple with the maximum and minimum for everything to the left of this leaf. It computes the gap for each element and stores is with with the given key. I wrote a helper function, `gap(List, Accin)` that returns a tuple of the form `{GapList, {Min, Max}}`, where `GapList` is the gap for each element of `List`, and the `{Min, Max}` tuple is the tuple produced by `Leaf1` or `Combine` (or the atom `'empty'`).

```
gap(L, AccIn) -> mapfoldl(
   fun
      (E, empty) -> {0, {E, E}};
      (E, {Min, Max}) ->
         X = min(Min,E),
         Y = max(Max, E),
         {Y-X, {X, Y}}
   end, AccIn, L).

gap(W, Src, Dst) ->
   wtree:scan(W,
      fun(S) -> % Leaf1
         element(2, gap(workers:get(S, Src), empty))
      end,
      fun(S, AccIn) -> % Leaf2
         workers:put(S, element(1, gap(workers:get(S, Src), AccIn)))
      end,
      fun % Combine
         (empty, Right) -> Right;
         (Left, empty) -> Left;
         ({Lmin, Lmax}, {Rmin, Rmax}) ->
            {min(Lmin, Rmin), max(Lmax, Rmax)}
      end
   ).
```

   Note: I haven't tried compiling and running this code yet – you didn't have that opportunity on the exam. When I do, there will almost certainly be some minor error. I'll fix them and might add

that version. You can think of the difference between the code written here and the final version as indicating the range of what I will consider to be an acceptable solution. Of course, there are many other valid implementations other than the one shown here. They will get full credit as well.

(b) (**12 points**) Given a list of numbers, report the number of consecutive Pythagorean triples. If `x`, `y`, `z` are consecutive elements of the list, then they are a Pythagorean triple iff $x^2 + y^2 = z^2$. Here are few examples:

```
nPythag3([1,2,4,6,8,10,11,3,4,5,12,13,14]) -> 3; % [6,8,10], [3,4,5], [5,12,13]
nPythag3([1,-2,3,-4,5]) -> 1; nPythag3([0,1,0,1,0,1,0,1]) -> 3;
```

Solution The problem is an instance of reduce. In my solution, each leaf worker process produces a tuple of the form `{First2, N_triple, Last2}` where `First2` is a list of the first two elements of the worker's piece of the lists; `Last2` is a list of the last two elements of the worker's piece of the lists; and `N_triple` is the number of Pythagorean triples in the worker's piece of the list. If the worker's piece of the list has fewer than two elements, then `Leaf` just returns the list. The `Combine` function combines these in the "obvious" way. Have a different pattern for leaves with short lists makes handling the special cases fairly straightforward. The `Root` function extracts the count from the triple, or if the top-value is a list, returns 0. Finally, I wrote `nPythag3(List)`, a sequential implementation which is used by `Leaf` and `Combine`.

```
nPythag3(L=[A, B | _]) ->
{N, Last2} = nPythag3(L, 0),
{[A,B], N, Last2};
nPythag3(L) -> L.
nPythag3(W, Src) ->
   wtree:reduce(W,
      fun(S) -> nPythag3(workers:get(S, Src)) end, % Leaf
      fun % Combine
         ({LL, LN, LR}, RL, RN, RR) ->
            {LL, LN + element(2, nPythag3(LR++RL)) + RN, RR};
         (L, {RL, RN, RR}) ->
            {First2, MidN, RL} = nPythag3(L++RL),
            {First2, MidN+RN, RR};
         ({LL, LN, LR}, R) ->
            {LR, MidN, Last2} = nPythag3(LR++R),
            {LL, LN+MidN, Last2};
         (L,R) -> nPythag3(L++R)
      end,
      fun % Root
         ({_, N, _}) -> N;
         (L) -> 0
      end
   ).
```

2. **Filter Locks (16 points)** I've mentioned several times that Peterson's mutual exclusion algorithm can be extended to apply to any number of clients. The algorithm is called a "filter lock". Here's the code:

```
00: int level[N], victim[N]; // initially all 0

01: lock(i) {
02:   for(int j = 1; j < N; j++) {
03:     level[i] = j;
04:     victim[j] = i;
```

```
05:     while((victim[j] == i) && ∃ 0 ≤ k < N. (k ≠ i) && level[k] ≥ j);
06:   }
07: }

08: unlock(i) {
09:   level[i] = 0;
10: }
```

The key idea in the lock is that each thread must pass through $N-1$ "levels" of locking (the `for(j...)` loop) before entering its critical section. To advance from level `j` to level `j+1`, there must be no other threads at level `j` or higher, **or** there must be some other thread that is the "victim" at level `j`. A consequence is that if there is another thread at this level or higher, then there must be at least one thread "left behind" at each of the lower levels. In particular, if thread x is in the critical section, and thread y is at level $N-1$, there are threads at every level from 1 to $N-1$; thread y is the victim at level $N-1$; and thread y can't advance to the critical section.

(a) (**8 points**) Show that the filter lock is **not** starvation free. For example, consider a filter lock with three clients: 0, 1, and 2. Show that one of the clients can spin forever while the other two alternately acquire and release the lock.

Solution: Proof that filter locks are starvation free. If a thread is stuck spinning at line `05`, then this thread must be the victim, and there must be another thread at or above this level. If there is another thread at this level, then that thread can advance because it is not the victim. This shows that the filter lock is deadlock free. Once the threads that are ahead of this one make it to the critical section and back to idle, this one must will be able to advance.

The concern is that some other threads may come reach this level in the meantime – for example, threads that were above this one could reach their critical section, come back around and then pass this one again. The key observation (and this is where I blew it) is that if thread i_1 is at level j, and thread i_2 enters level k then thread i_2 sets the `victim[j]` variable to i_2. It will not get set back to i_2 (until after i_1 has reached its critical section and is trying again for the lock), because only thread i_1 can set a victim variable to i_1. Thus, once some other thread enters level j, thread i_1 can progress at least to level $j+1$. Eventually, thread i_1 gets the lock.

(b) (**8 points**) Show that if the statements at lines `03` and `04` are exchanged (i.e. `victim[j] = i;` is performed *before* `level[i] = j;`, then mutual exclusion can be violated.

Solution: The following execution shows a violation:

0. All threads are at level 0.
1. Thread 1 sets `victim[1] = 1;`
2. Thread 2 sets `victim[1] = 2;`
3. Thread 2 sets `level[2] = 1;`
4. Thread 2 checks the condition at line `05`. It is the victim, but there is no other thread with `level` at or above 1.
5. Thread 2 enters it's critical section.
6. Thread 1 sets `level[1] = 1;`
7. Thread 1 checks the condition at line `05`. It is the not victim.
8. Thread 1 enters it's critical section.

3. **Sorting (25 points)**

The question finishes showing that the compare-and-swap elements of sorting networks can be replaced by "merge-and-split" operations. Homework 4, question 2, also looked at this. This can be used to obtain practical parallel sorting algorithms for common parallel architectures.

Let M be a positive integer. We will consider merge-and-split operations on lists of length M. In particular,

```
merge_and_split({In0, In1}) -> {Out0, Out1}
```
where:

`In0` and `In1` are lists of M elements in ascending order.

`Out0` is the first M elements of `merge(In0, In1)`, sorted into ascending order.

`Out1` is the last M elements of `merge(In0, In1)`, sorted into ascending order.

For example:

```
merge_and_split({[1, 4, 9, 16, 25, 36], [3, 5, 7, 9, 11, 13]}) ->
   {[1, 3, 4, 5, 7, 9], [9, 11, 13, 16, 25, 36]}.
```

(a) (**5 points**) What is the result of

```
merge_and_split({[0, 0, 0, 0, 1, 1], [0, 0, 0, 1, 1, 1]})
```

?

Solution: `{[0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 1]}`.

(b) (**5 points**) Show that if `In0` and `In1` are lists of 0s and 1s in ascending order, and

```
{Out0, Out1} = merge_and_split({In0, In1})
```

then at least one of `Out0` or `Out1` must be clean (i.e. all 0s or all 1s).

Solution: There are a total of $2M$ elements in `In0` and `In1`. If at least M of these are 0s, then `Out0` is all 0s and therefore clean. Otherwise, at least $M + 1$ of the input elements are 1s, and `Out1` is all 1s and therefore clean.

(c) (**10 points**) Let S be a sorting network with N inputs and N outputs. Let S' be the network obtained by replacing every compare-and-swap in S with a merge-and-split operation. Now, let A_0, A_1, ..., A_{N-1} be any N sorted lists of 0s and 1s of length M. Apply these lists as the inputs to S', and let X_0, X_1, ..., X_{N-1} be the resulting outputs.

Show that there is an input to S', B_0, B_1, ..., B_{N-1} such that B is a permutation of X and when B is applied as the input to S', X is the output.

Solution: The main idea is that we can derive B by working backwards through the sorting network from X. We use the forward computation that started with A to see where the clean lists are in the network. If A produces a clean list at the output of some merge-and-split operation, we'll keep that list in our backwards pass for deriving B. We note that if A produces a dirty list at the output of some merge-and-split operation, we can change the number of 1s or 0s in that list, and still be able to find a valid input to that merge-and-split that produces the outputs that we are using in our backwards sweep. This lets us keep the lists as a permutation of the lists from X, and we never have to break a clean lists from X into the dirty lists of A that went into it. Quite a bit of partial credit will be given for making observations like those above.

The rest of my solution gives the details for the proof. I give lots more details than I require in a full-credit solution. I'm trying to write a solution that will be clear to someone who has already thought about the problem but may not have solved it.

We can describe a sorting network as a list of tuples. The list `[{I,J} | Tl]` describes a network that first does a merge-and-split of lists `I` and `J`, and applies the resulting lists to the sorting network described by `Tl`. In particular, Let

```
derive_b(A, []) -> A;
derive_b(A, S1 = [{I, J} | Tl]) ->
   {AI, AJ} = {nth(I, A), nth(J, A)},
   {RI, RJ} = merge_and_split({AI, AJ}),
   R = replace(I, RI, replace(J, RJ, A)),
   Y = derive_b(R, Tl),
```

```
{YI, YJ} = {nth(I, Y), nth(J, Y)}
{BI, BJ} = if
   clean(AI) and (YJ == AI) -> YJ, YI;
   clean(AJ) and (YI == AJ) -> YJ, YI;
   true -> YI, YJ
end,
B = replace(I, BI, replace(J, BJ, Y)).
```

Where `nth(I, List)` is the I^{th} element of `List`; `replace(I, X, List)` is the list obtained by replacing the I^{th} element of `List` with `E`; and `clean(List)` is true if every element of `List` is 0 or every element of `List` is 1. If `S` is a sorting network (using merge-and-split operations) and `LL` is a list of lists, then I'll write `S(LL)` to denote the list of lists obtained by applying sorting network `S` with input `LL`.

To understand the code, note that `{I,J}` are the indices of the lists for the next merge-and-split. Thus `AI` and `AJ` are the inputs to that merge-and-split, and `RI` and `RJ` are the outputs. The goal now is to derive `B` so that `S1(A) = S1(B)` and the lists of `B` are a permutation of the lists of `S1(A)`. The code does this recursively. It constructs `Y` so that `Tl(Y) = S1(A)` and the lists of `Y` are a permutation of the lists of `S1(A)`. Then, it derives `B` from `A` and `Y`.

Let `B = derive_b(A, S')`. I'll show that `B` has the properties described above. My proof is by induction on `S1`, and my induction hypothesis is

`S1(B) == S1(A)`,

and The lists of `B` are the same (but possibly permuted) as the lists of `S1(A)`,

and If the I^{th} list of `A` is clean, then the I^{th} list of `B` is the same as the I^{th} list of `A`.

The base case is when `S1 = []`, then `B = A = X`, and the claims are immediate. Otherwise, let `S1 = [{I,J} | Tl]`. From the induction hypothesis, we have that `Tl(R) = S1(A)`. There are two cases to consider:

`clean(AI) and clean(AJ)`: then `RI` and `RJ` are both clean. From the induction hypothesis, `YI=RI` and `YJ=RJ` are both clean. From the declarations for `BI` and `BJ`, `BI=AI` and `BJ=AJ`, therefore `B=A` and `S1(B) = S1(A)`, and all clean lists of `A` are equal to their counterparts in `B`. From the declarations for `BI` and `BJ`, `B` is a permutation of `Y`. From the induction hypothesis, `Y` is a permutation of `S1(A)`. Therefore, `B` is a permutation of `S1(S)`.

Otherwise: On of `AI` or `AJ` (or both) are dirty. From question 3b, one of `RI` or `RJ` is clean. If `AI` is clean and all 0s, then `RI` will be clean and all 0s (def. `merge_and_split`); `YI` will be clean and all 0s (induction hypothesis); and `BI` will be clean and all 0s (declaration of BI). Similar arguments apply in the other three cases where `AI` or `AJ` are clean. If `AI` and `AJ` are both dirty, then one of `RI` or `RJ` will be clean (question 3b); and the corresponding `YI` or `YJ` will be clean. The other one may be either clean or dirty. In this case `BI=YI` and `BJ=YJ`. By construction,

`{YI, YJ} = merge_and_split({BI, BJ})`

Therefore, `S1(B) = T1(Y) = S1(A)`. Furthermore, the lists of `B` are a permutation of the lists of `S1(A)` by the same argument as used in the previous case.

This completes the induction proof. Therefore, `B = derive_b(A, S')` satisfies the induction hypothesis, which means that it is a permutation of `X = S'(A)` and `S'(B) = X` as required.

(d) (**5 points**) The induction argument from question 3c shows that if the lists $B_1, \ldots, B_{N-1}$ are applied as the inputs to S', then for each `merge_and_split` operation in S', either `{Out0, Out1} = {In0, In1}`, or `{Out0, Out1} = {In1, In0}`. For $0 \leq i < N$, let C_i be the number of 1s in list B_i. Show that if S' sorts the B lists incorrectly, S must sort C incorrectly as well.

Solution: If `B` is applied as an input to `S'` then each merge-and-split operation has at least one clean input – otherwise, the number of 1s in the lists wouldn't be preserved by the merge-and-split operations. If one input to a merge-and-split is clean, then it must have a clean output, and the other output must match the other input. In other words, the merge and split either copies `{In0, In1}` to `{Out0,`

`Out1`} or it swaps them. In the problem statement, I told you you could assume this; so, a solution can get full credit without the argument of this paragraph.
The lists consist of 0s and 1s and are sorted into ascending order; so, there are only $M+1$ possible lists, depending on how many 1s are in the list. Thus, each input to a merge-and-split can be described using integers in $0, \ldots, M$, and the outputs of the merge-and-split are the same of the inputs if `In0` has fewer 1s than `In1` and swapped if `In0` has more 1s than `In1`. Thus, a compare-and-swap of the integers is equivalent to merge-and-split of the lists. This can be applied to the entire sorting network. Therefore, if the merge-and-split sorting network produces an unsorted result with the B list-of-lists, then the compare-and-swap network will produce an unsorted result with the C list-of-integers.

Hint: you can do this even if you didn't solve question 3c. In particular, you are allowed to assume what I said about the `merge_and_split` operations.

4. **Matrix Multiplication** (20 points)
Let's say you get a job writing code at a local company that needs to do some moderately intense number crunching. A typical problem is computing products of $N \times N$ matrices, where N ranges from 1000 to 20000.

(a) (**4 points**) Let's say you've got a laptop with a 2.5GHz, quad-core processor. From mini-assignment 4, we know that multiplying a $N_1 \times N_2$ by a $N_2 \times N_3$ matrix on a machine with a clock frequency of f takes time of roughly $3N_1N_2N_3/f$. What is the time to compute a matrix products for $N = 1000$, $N = 5000$, and $N = 20000$?

Solution: The time is $3N^3/f$. For $f = 2.5$ GHz, this is $(1.2 \text{ seconds})(N/1000)^3$.
$N = 1000$: $T_{seq} = 1.2$ seconds.
$N = 5000$: $T_{seq} = 1.2 * (5^3)$ seconds $= 1.2 * 125$ seconds $= 150$ seconds.
$N = 20000$: $T_{seq} = 1.2 * (20^3)$ seconds $= 1.2 * 8000$ seconds $= 9600$ seconds.

(b) (**4 points**) You try to convince your manager to buy a cluster of linux machines for your computation. Your proposal is for 32, 8-core processors running at 2.5GHz. The total number of processors is 256. For simplicity, we'll pretend that there isn't any multi-threading. Assume that the time to transfer a block m double-precision values between processors (over the network, it's a cluster) is:

$$T_{network} \quad = \quad 0.1\text{ms} + m * 50\text{ns}$$

where $1\text{ms} = 10^{-3}$seconds, and $1\text{ns} = 10^{-9}$seconds.
Using the algorithm we derived in class (Sept. 24), each processor performs P matrix products where it multiplies a $(N/P) \times (N/P)$ matrix by a $(N/P) \times N$ matrix. Furthermore, each processor sends (and receives) $P-1$ messages of size N^2/P double precision numbers. For simplicity, assume that the time for sending messages cannot be overlapped with computation time. What is the speed-up when computing matrix products for $N = 1000$, $N = 5000$, and $N = 20000$?

Solution: Each processor computes P multiplications of a $(N/P) \times (N/P)$ matrix by a $(N/P) \times N$ matrix. Each matrix multiplication takes $3N^3/(P^2 f)$ time. The total compute time for each processor is $P\frac{3N^3}{P^2 f} = 3N^3/(Pf)$ which is just T_{seq}/P. The communication time is $(P-1) * (0.1\,\text{ms} + 50(N^2/P)\,\text{ns})$. That gives us the following table:

N	T_{seq}	$T_{par,1}$	$T_{par,2}$	$T_{par,3}$	T_{par}	*speed_up*
1000	1.2s	4.6875ms	1.5ms	49.8ms	55.9875ms	21.4
5000	150s	586ms	1.5ms	1.245s	1.8326s	81.9
20000	9600s	37.5s	1.5ms	19.92s	57.42s	167.2

where $T_{par,1} = 3N^3/(Pf)$ is the compute time for the parallel version; $T_{par,2} = (P-1) * 0.1\,\text{ms}$ is the part of the communication time that is independent of matrix size; and

$$T_{par,3} \quad = \quad (P-1)50\tfrac{N^2}{P}\,\text{ns} \quad = \quad 50\tfrac{P-1}{P}\left(\tfrac{N}{1000}\right)^2\,\text{ms}$$

is the part of the communication time that depends on the matrix size. Of course, to save time, you didn't need to copy numbers from your solution to question 4a. I put them all into the table to make it easier to read.

Note: for all parts of this problem, I did most of the calculations in my head, and used a calculator for the multi-digit divisions. My answers may be slightly off by one or two in the least significant digits. That's indicative of the error-tolerance that I'll include in grading.

(c) (**4 points**) Your manager isn't convinced, but you really want to her to buy that linux cluster. You read up on algorithms for matrix multiplication and find an algorithm where each processor performs $\sqrt{P}$ matrix products where it multiplies a $(N/\sqrt{P}) \times (N/\sqrt{P})$ matrix by another $(N/\sqrt{P}) \times (N/\sqrt{P})$. Furthermore, each processors sends (and receive) $\sqrt{P}$ messages of size N^2/P double precision numbers. With the new algorithm, what is the speed-up when computing matrix products for $N = 1000$, $N = 5000$, and $N = 20000$?

Solution: The computation time stays the same. There are now $\sqrt{P}$ messages sent per processor instead of $P - 1$. I'll just scale the communication times ($T_{par,2} + T_{par,3}$) from the previous problem by $\sqrt{P}/(P-1) = 0.62745$. The new table is:

N	T_{seq}	$T_{par,1}$	$T_{par,2+3}$	T_{par}	$speed\_up$
1000	1.2s	4.6875ms	3.22ms	7.91ms	151.7
5000	150s	586ms	78.2ms	664.2ms	225.8
20000	9600s	37.5s	1.25s	38.75s	247.7

Reducing the number of messages makes a big difference!

(d) (**8 points**) A coworker installs an optimized BLAS (numerical library) which is faster than the simple code I showed in class. You find that the time to multiply a $N_1 \times N_2$ matrix by a $N_2 \times N_3$ matrix on a machine with a clock frequency of f now takes time of roughly $1.2N_1N_2N_3/f$. What is the sequential time (as in question 4a and parallel time (as in question 4c when the BLAS library is used?

i. What is the sequential time (as in question 4a and parallel time (as in question 4c when the BLAS library is used?

Solution: The sequential times get divided by $3/1.2 = 2.5$. For the parallel version, the compute time is divided by 2.5, but the communication time stays the same.

N	T_{seq}	$T_{par,1}$	$T_{par,2+3}$	T_{par}	$speed\_up$
1000	0.48s	1.875ms	3.22ms	4.1ms	117.5
5000	60s	234.4ms	78.2ms	312.6ms	191.9
20000	3840s	15s	1.25s	16.25s	236.3

ii. Write one sentence to summarize the impact of the faster sequential algorithm on the execution time.

Solution: Execution times dropped for both the sequential and parallel versions.

iii. Write one sentence to summarize the impact of the faster sequential algorithm on the speed-up.

Solution: Speed-ups decreased as well.

iv. Write one sentence to explain the relationship you observe between execution time and speed-up?

Solution: A faster sequential implementation results in lower speed-ups because the overheads (communication) become a larger percentage of the total execution time for the parallel program.

5. **Other Stuff (16 points)**

Answer each question below with 1-3 sentences. Points may be taken off for long answers.

(a) (**4 points**) What is instruction level parallelism?

Solution: Executing multiple instructions in parallel by identifying their dependencies and executing instructions when their dependencies are resolved. Super-scalar architectures are one way to exploit instruction level parallelism.

(b) (**4 points**) What is "single-instruction, multiple-thread" parallelism?

Solution: Fetching and decoding a single instruction stream and sending it to multiple execution pipelines. Each pipeline can resolve branch conditions independently. Instead of taking or not taking a branch, instructions are executed conditionally based on the branch outcome. This is called "predicated execution." GPUs are an example of an architecture that provides single-instruction multiple-thread parallelism.

(c) (**4 points**) How does map-reduce handle machine and network failures?

Solution: The map and reduce processes can be (re-)executed on a different machine if the original machine fails to respond (e.g. to a ping). The entire job can be re-executed with a different master process if the master fails to respond.

(d) (**4 points**) What is a memory fence?

Solution: An operation that forces pending memory operations to complete, thus ensuring that all cache coherence operations have finished as well.

Note, my answer for single-instruction, multiple-thread was five sentences. A three sentence answer is fine as long as it clearly gets two or three key points. The same applies for the other problems as well.