CpSc 448B **Final Exam** December 16, 2011

Solution Set

100 points

Time for the exam: 2.5 hours.

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 won't 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, i-pods, etc. is not permitted during the exam.

Good luck! ☺

CpSc 448B **Final Exam** December 16, 2011

100 points

Solve any **five** of the six problems below. If you write solutions (or partial solutions) to all six, please clearly write on the cover of your exam book which five you want to have graded.

Remark: The process of grading tends to reveal that there can be many, slightly different ways of interpreting the details of the problem statement. Any reasonable assumptions are acceptable as long as they are clearly stated or obvious from the solution itself. This solution set gives fairly detailed descriptions of the solutions that I had in mind for the problems. I have also noted other legitimate ways to solve some of the problems.
The solutions here are much more detailed that required to get full-credit. I tried to write self-contained solutions so that someone who wasn't able to solve the problem could figure out how to do it based on the solutions here. Perhaps, I should post a shorter version that would be representative of what I would expect for a full-credit solution – that will have to wait for another term.

1. **Reduce and Scan** (**20 points**) Consider a bank account with an initial balance of `b0`, and two arrays, `delta` and `interest` that describe daily activity in the account:

   `delta[i]` is the net amount deposited in the account on day `i` (i.e. `delta[i]` is the sum of the deposits on day `i` minus the sum of the withdrawals.).

   `interest[i]` is the daily interest on day `i`.

   If we have entries for `n` days, then a sequential algorithm for computing the daily account balance would be:

```
balance[0] = b0;
for(i = 0; i < n; i++) {
   balance[i+1] = (balance[i] + delta[i])*(1.0 + interest[i]);
}
```

   Sketch a parallel computation using reduce or scan for each of the three problems below. Grading will be based primarily on clarity and correctness. Your solution should not be extremely long.

   (a) (**6 points**) Find the third largest entry in `delta`. You may assume that you have a sequential function `seq_top3(v)` that takes as an argument an array, and returns an array consisting of the three largest elements of that array.

   Solution: My *leaf* function takes the uses *seq_top3(...)* to find the three largest elements of each subarray. I assume that the array returned by *seq_top3(...)* has the three largest elements in descending order. My *combine* function takes two arrays of three values (the largest three in each subarray), and produces an array consisting of the three largest of those six. My *root* function takes the third largest element from its array. To handle corner cases where a subarray has fewer than three elements, I assume that I can use a special value, *NEG_INFINITY*; *seq_top3(...)* will set elements of its result to this if needed; and that *NEG_INFINITY* tests to be less than any other integer. Here are the functions:

```
int[] leaf() {
   return(seq_top3(delta)); // I'm assuming that delta, etc. are visible in this
}                           // process/thread's methods.

int[] combine(int[] left, int[] right) {
   int i_left = 0, i_right = 0;
   int[] merge = new int[3];
   for(int i = 0; i < 3; i++) {
      if(left(i_left) >= right(i_right)) merge[i] = left(i_left++);
      else merge[i] = right(i_right++);
   }
   return(merge);
}

int root(int[] top3) {
   return(top3[2]);
}
```

To compute the third largest element of *delta* evaluate:

```
reduce(leaf, combine, root)
```

(b) (**7 points**) Determine the final balance of the bank account described above. This is the same as `balance[n]` described above. For this problem, you just need to compute the final balance. You don't need to compute all of the elements of the `balance` array.

Solution: This time, I'll use reduce in pidgin erlang – it makes tuples simpler. My *Leaf* function computes the amortized (accumulated with interest) value for its interval of the dates and the compounded interest for that interval. My *Combine* function multiplies the amortized value from the left by the compounded interest rate from the right, and adds this to the amortized value from the right to get the amortized value for the entire interval. The compounded interest for the combined interval is the product of the compounded interest from the left and the compounded interest from the right. The *Root* function just returns the amortized value for the entire tree. I'm assuming that processes have a *get* function that does the obvious thing (Erlang actually provides such a *get*). Here's the code:

```
value([], [], V) -> V;
value([H_delta | T_delta], [H_rate | T_rate], {Balance, Interest}) ->
   value( T_delta, T_rate,
          {Balance*(1.0 + H_rate) + H_delta, Interest*T_rate}).

Leaf = fun() -> value(get(delta), get(interest).  {0.0, 1.0}) end.
Combine = fun({L_bal, L_rate}, {R_bal, R_rate}) ->
   {L_bal * R_rate + R_bal, L_rate * R_rate}.
Root = fun({Balance, Rate}) -> Balance.
```

To compute the final balance, evaluate the expression

```
reduce(Leaf, Combine, Root)
```

(c) (**7 points**) Determine the daily balance of the bank account described above. In other words, produce an array, `balance[i]` for $0 \leq i \leq n$.

Solution: This is a simple variant of the solution to part b. We can use *Leaf* from part b as *Leaf1* and *Combine* from part b as our *Combine*. We just need *Leaf2* and *Acc0*. The *Leaf2* function will take the balance from the its *AccIn* parameter and use this as the starting balance for updating the daily balance for its array. *Acc0* should describe an initial balance of 0. It doesn't matter what the compounded interest is; I'll set it to 1 as that is the identity element for these rates. Here's the code:

```
balance([], [], BalIn, BList) -> BList; % BList out?
balance([H_delta | T_delta], [H_rate, T_rate], BalIn, BList) ->
   NewBalance = BalIn*H_rate + H_delta, % a running shoe?
   balance(T_delta, T_rate, NewBalance, [NewBalance | BList]).

Acc0 = {0, 1}.
Leaf1 = Leaf.
Leaf2 = fun({BalIn, _}) ->
   putbalance,
      lists:reverse(balance(get(delta), get(interest), BalIn))).
```

To compute the daily balances, evaluate the expression

```
scan(Leaf1, Leaf2, Combine, Acc0)
```

You should use reduce or scan functions that are equivalent to those defined in the `wtree` module (the erldoc for those functions is included at the end of this document). You don't have to write your solution in Erlang – you can use Erlang, Peril-L, or pidgin versions of C, C++, or Java, as long as your use of reduce and/or scan is clear, and consistent with the `wtree` versions. If you use a distributed approach, you can assume that the arrays (or lists) for `delta` and `interest` are distributed across the processors before your functions are called, and your `balance` array (or list) should be likewise distributed at the end of your solution to part **c**. You can assume that a process can remember its state and thus omit the "process state" stuff from `wtree`.

2. **Data Transposition** (**20 points**) Consider a computation involving P processes with each process running on a different processor. For $0 \leq i < P$, let p_i denote the i^{th} process. Each process, p_i has an array, X_i, of P objects, where each object consists of B bytes. Let $X_{i,j}$ denote the j^{th} element of the array X_i. We want to "deal" each process's data amongst all of the processes. At the end of this operation, each process, p_i will have an array, Y_i of P objects where each object consists of B bytes such that for all $0 \leq i, j \leq P$, $Y_{i,j} = X_{j,i}$. Thus, we can think of Y as a kind of transpose of X.

   We'll perform the transpose described above by sending messages between processors. Assume that sending a message of B bytes between two processors takes time

$$t_0 + t_1 * B$$

   for some constants t_0 and t_1.

   (a) (**7 points**) A simple implementation has each processor send $P-1$ messages of B bytes each to each of the $P-1$ other processors. Likewise, each processor will receive $P-1$ messages of B bytes each from each of the $P-1$ other processors. Write an expression for the elapsed time (i.e. wall-clock time from starting the operation until it finishes) to perform the transpose, $T_{elapsed}$ in terms of B, P, t_0, and t_1.

Solution: $T_{elapsed} = (P-1) * (t_0 + t_1 * B)$

   (b) (**7 points**) Another approach works by handling one bit of the indices at a time. For example, if i is even, then the "bit-0 partner" of p_i is p_{i+1}. Likewise, if i is odd, then the bit-0 partner of p_i is p_{i-1}. At the first step of this, every processor, p_i with i even sends all of its odd-indexed blocks to its bit-0 partner, and every odd-indexed processor sends all of its even indexed blocks to its bit-0 partner. We then do a similar swap based on the bit 1 of the processor index (the bit in the 2's place), then bit 2 (the bit in the 4's place) and so on. Here is pseudo code for the algorithm:

```
G = log2(P); // assume P is a power of 2
forall i in 0..P-1 { // all processors in parallel
   M = array of P/2 elements of size B
   for(k = 0; k < G; k++) {
      mask = 1 << k; // k has a 1 in the k^th bit
      partner = i ∧ mask; // our bit-k partner
      // set M to have all blocks of X where the k^th bit of the block index differs from
      // the k^th bit of the process index.
      jj = 0;
      for(j = 0; j < P; j++)
         if((j & mask) != (i & mask))
            M[jj++] = X[j];
      send(partner, M);
      receive(partner, M);
      // update our blocks
      jj = 0;
      for(j = 0; j < P; j++)
         if((j & mask) != (i & mask))
            X[j] = M[jj++];
   }
}
```

   Write an expression for the total time elapsed time to perform the transpose, $T_{elapsed}$ in terms of $B, P, t_0, and t_1$.

Solution: $T_{elapsed} = \log_2(P) * (t_0 + t_1 * P * B/2)$

   (c) (**6 points**) Assume $t_0 = 100$, $t_1 = 1$, and $P = 1024$. Which approach is faster if $B = 1$? Which approach is faster if $B = 1024$?

Solution: If $B = 1$ then the first approach requires time $(1024 - 1) * (100 + 1 * 1) = 103,323$, and the second requires time $\log_2(1024) * (100 + 1 * 1024 * 1/2) = 6120$. In this case, the second method is about 17 times faster than the first.

If $B = 1024$ then the first approach requires time $(1024 - 1) * (100 + 1 * 1024) = 1,149,852$, and the second requires time $\log_2(1024) * (100 + 1 * 1024 * 1024/2) = 5,243,880$. In this case, the first method is about 5 times faster than the second.

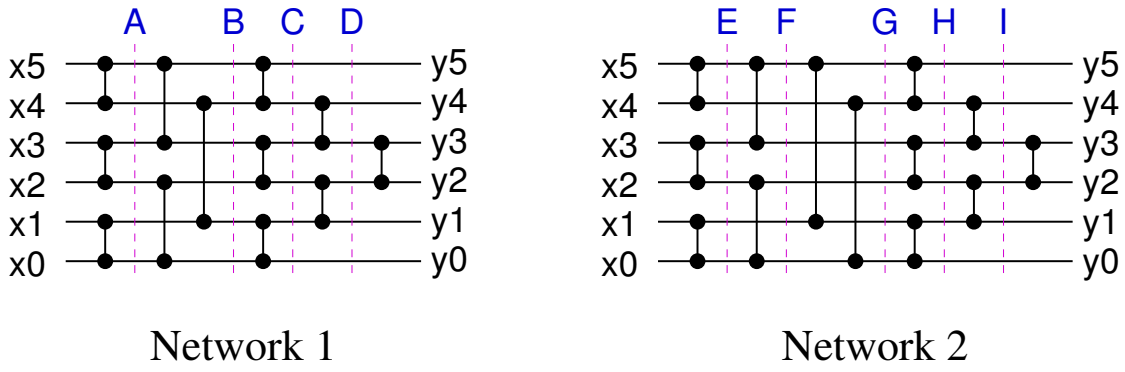

Figure 1: Two sorting networks

3. **Sorting Networks** Consider the two sorting networks shown in figure 1. One of them sorts all inputs correctly, and the other does not.

   (a) (**10 points**) Determine which network is the correct network. Hint, the incorrect one fails for one of the two inputs listed below:

   ```
   x = [2, 3, 0, 1, 5, 4]
   x = [3, 1, 0, 5, 2, 4]
   ```

   Note: these vectors are listed with `x5` on the left and `x0` on the right – this was clarified during the exam.

Solution: The second vector is a counter-example to the second sorting network:

		Network 1						Network 2						
line	input	A	B	C	D	output	line	input	E	F	G	H	I	output
5	3	3	5	5	5	5	5	3	3	5	5	5	5	5
4	1	1	4	4	4	4	4	1	1	1	1	1	3	3
3	0	5	3	3	3	3	3	0	5	3	3	3	1	4
2	5	0	2	2	2	2	2	5	0	2	2	2	4	1
1	2	4	1	1	1	1	1	2	4	4	4	4	2	2
0	4	2	0	1	1	1	0	4	2	0	0	0	0	0

For good measure, I'll show what the other vectors do. Both networks sort the first vector correctly:

		Network 1						Network 2						
line	input	A	B	C	D	output	line	input	E	F	G	H	I	output
5	2	3	3	5	5	5	5	2	3	3	5	5	5	5
4	3	2	5	3	4	4	4	3	2	2	2	2	4	4
3	0	1	1	4	3	3	3	0	1	1	1	4	2	3
2	1	0	4	1	2	2	2	1	0	4	4	1	3	2
1	5	5	2	2	1	1	1	5	5	5	3	3	1	1
0	4	4	0	0	0	0	0	4	4	0	0	0	0	0

Now, I'll try the two test vectors in reverse order. Both networks sort the reversed version of the first vector correctly.

	Network 1						Network 2							
line	input	A	B	C	D	output	line	input	E	F	G	H	I	output
5	4	5	5	5	5	5	5	4	5	5	5	5	5	5
4	5	4	4	4	4	4	4	5	4	4	4	4	4	4
3	1	1	1	2	2	3	3	1	1	1	1	2	2	3
2	0	0	2	1	3	2	2	0	0	2	2	1	3	2
1	3	3	3	3	1	1	1	3	3	3	3	3	1	1
0	2	2	0	0	0	0	0	2	2	0	0	0	0	0

Both networks sort the reversed version of the second vector correctly as well.

	Network 1						Network 2							
line	input	A	B	C	D	output	line	input	E	F	G	H	I	output
5	4	4	5	5	5	5	5	4	4	5	5	5	5	5
4	2	2	3	4	4	4	4	2	2	2	2	4	4	4
3	5	5	4	2	2	3	3	5	5	4	4	2	2	3
2	0	0	1	1	3	2	2	0	0	1	1	1	3	2
1	1	3	2	3	1	1	1	1	3	3	3	3	1	1
0	3	1	0	0	0	0	0	3	1	0	0	0	0	0

It's sorted correctly by both networks

(b) (**10 points**) Give an input consisting only of zeros and ones that the incorrect network fails to sort correctly.

Solution: We start with the counter example described above for the second sorting network:

$$\begin{bmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 5 \\ 2 \\ 4 \end{bmatrix} \xrightarrow{\textit{Network 2}} \begin{bmatrix} 5 \\ 3 \\ 4 \\ 1 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix}$$

Sorting networks commute with monotonic functions; so, I'll pick a threshold for the elements of these vectors that will cause the network to fail. For example, the elements with the values of 1 and 2 come out in the wrong order. Noting that $1 < 1.5 < 2$, I'll choose $x' = (x > 1.5)$ as my solution:

	Network 2						
line	input	E	F	G	H	I	output
5	1	1	1	1	1	1	1
4	0	0	0	0	0	1	1
3	0	1	1	1	1	0	1
2	1	0	1	1	1	1	0
1	1	1	1	1	1	1	1
0	1	1	0	0	0	0	0

Thus,

$$\begin{bmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

is a vector consisting only of zeros and ones that is sorted incorrectly by the second network.

4. **Bitonic Merge** (**20 points**) Let $x_0, \ldots x_{n-1}$ be a bitonic sequence. Let y be the sequence with:

$$\begin{aligned} y_i &= \min(x_i, x_{i+\frac{n}{2}}), \quad \text{if } i < \tfrac{n}{2} \\ &= \max(x_i, x_{i-\frac{n}{2}}), \quad \text{if } i \geq \tfrac{n}{2} \end{aligned}$$

(a) (**10 points**) Prove $\forall 0 \leq i < \frac{n}{2}.\ y_i \leq y_{i+\frac{n}{2}}$.

Solution: For all $0 \leq i < \frac{n}{2}$, $y_i = \min(x_i, x_{i+\frac{n}{2}}) \leq \max(x_i, x_{i+\frac{n}{2}}) = x_{y+\frac{n}{2}} = y$. Thus, $y_i \leq y_{i+\frac{n}{2}}$ as required.

I had really meant to ask you to show that for all $0 \leq i < \frac{n}{2}$ and all $\frac{n}{2} \leq j < n$

$$y_i \quad \leq \quad y_j\ .$$

This is a nice warm-up for part b; so, I'll prove this stronger claim as well. The main impact of this change is that I put a few more points on the "easy" half of the problem than I otherwise would have. The combination of parts a and b remain what I had originally planned.

As noted in the hints, we only need to consider x vectors for which each element is either zero or one. Graphically, we've got:

$x_{\frac{n}{2}}$	$x_{\frac{n}{2}+1}$	$\ldots$	x_{n-1}
x_0	$x1$	$\ldots$	$x_{\frac{n}{2}-1}$

c&s →

$y_{\frac{n}{2}}$	$y_{\frac{n}{2}+1}$	$\ldots$	y_{n-1}
y_0	$x1$	$\ldots$	$y_{\frac{n}{2}-1}$

where "c&s" denotes the compare-and-swap operations described in the problem statement. In particular, the lower element in each column for y is the smaller of the two elements in the corresponding column for x, and the upper element in each column for y is the larger of the two elements in the corresponding column for x.

I will show that for any choice of x that is bitonic, either $y_0, \ldots y_{\frac{n}{2}-1}$ are all 0, or $y_{\frac{n}{2}}, \ldots y_{n-1}$ are all 1, and the claim follows immediately.

Using the depiction of the x and y vectors above, I'll refer to $x_0, \ldots x_{\frac{n}{2}-1}$ as the lower row of x and $x_{\frac{n}{2}}, \ldots x_{n-1}$ as the upper row, and likewise for y. If either row of x consists entirely of zeros, then the lower row of y will consist entirely of zeros, and the claim holds. Likewise, if either row of x consists entirely of ones, then the upper row of y will consist entirely of ones and the claim holds. Thus, we only need to consider the case when both rows of x are mixed.

I'll consider the case where $x_0 = x_{n-1} = 0$. The case when $x_0 = x_{n-1} = 1$ is nearly the same. This means that there is an i and j with $0 < i < j < n-1$ such that:

$$\begin{aligned} x_k &= 0, \quad \textit{for all } k \textit{ with } 0 \leq k < i \\ &= 1, \quad \textit{for all } k \textit{ with } i \leq k < j \\ &= 0, \quad \textit{for all } k \textit{ with } j \leq k < n \end{aligned}$$

Graphically, this looks like:

$x_{\frac{n}{2}}$	$x_{\frac{n}{2}+1}$	$\ldots$	x_{j-1}	x_j	$\ldots$	$x_{i+\frac{n}{2}-1}$	$x_{i+\frac{n}{2}}$	$\ldots$	x_{n-1}
x_0	x_1	$\ldots$	$x_{j-\frac{n}{2}-1}$	$x_{j-\frac{n}{2}}$	$\ldots$	x_{i-1}	x_i	$\ldots$	$x_{\frac{n}{2}-1}$

=

1	1	$\ldots$	1	0	$\ldots$	0	0	$\ldots$	0
0	0	$\ldots$	0	0	$\ldots$	0	1	$\ldots$	1

This is the same arrangement as we had with mesh sort from homework 4, and I'll use the same argument to show that after the compare-and-swap operations are performed, either the lower row will be all zeros, or the upper row will be all ones. In both cases the claim holds.

If $j - i < \frac{n}{2}$ (the case depicted above), then for all k with $0 \leq < \frac{n}{2}$ either $k < i$ in which case $x_k = 0$ or $k + \frac{n}{2} > j$ in which case $x_{k+\frac{n}{2}} = 0$. In either case, $\min(x_k, x_{k+\frac{n}{2}}) = 0$, and $y_0, \ldots y_{\frac{n}{2}-1}$ are all 0. Likewise, if $j - i \geq \frac{n}{2}$, then for all k with $0 \leq < \frac{n}{2}$, either $k < i$ in which case $k + \frac{n}{2} > j$ and $x_{k+\frac{n}{2}} = 1$, or $i \leq k < \frac{n}{2}$ in which case $x_k = 1$. In either case, $\max(x_k, x_{k+\frac{n}{2}}) = 1$, and $y_{\frac{n}{2}}, \ldots y_{n-1}$ are all 1.

This completes the proof.

(b) (**10 points**) Prove that the sequence $y_0 \ldots y_{\frac{n}{2}-1}$ is bitonic.

Solution: Divide x and y each into two rows as in my solution to the stronger version of part a. If either row of x is all zeros, then the bottom row of y is all zeros, and is trivially bitonic. If either row of x is all ones, then the bottom row of y is the same as the other row of x. Because any subsequence of a bitonic sequence is bitonic, the bottom row of y is bitonic in this case as well. Thus, as in part a, the main case is when both rows of x are mixed.

If both rows of x are mixed, then let i and j be defined as in part a. If $j - i < \frac{n}{2}$,

$x_{\frac{n}{2}} \ldots x_{n-1}$:	1	1	…	1	0	…	0	0	…	0
$x_0 \ldots x_{\frac{n}{2}-1}$:	0	0	…	0	0	…	0	1	…	1

then (as shown in part a), $y_0, \ldots y_{\frac{n}{2}-1}$ are all 0, in which case $y_0, \ldots y_{\frac{n}{2}-1}$ is trivially bitonic.

If $j - i \geq \frac{n}{2}$,

$x_{\frac{n}{2}} \ldots x_{n-1}$:	1	1	…	1	1	…	1	0	…	0
$x_0 \ldots x_{\frac{n}{2}-1}$:	0	0	…	0	1	…	1	1	…	1

then

$$\begin{aligned} y_k &= 0, \quad \textit{for all } k \textit{ with } 0 \leq k < i \\ &= 1, \quad \textit{for all } k \textit{ with } i \leq k < n - j \\ &= 0, \quad \textit{for all } k \textit{ with } n - j \leq k < \tfrac{n}{2} \end{aligned}$$

Thus, $y_0, \ldots y_{\frac{n}{2}-1}$ is bitonic.

Note that $\frac{n}{2} < j < n$, thus $0 < n - j < \frac{n}{2}$.

Hints: Because y is computed from x using compare-and-swap operations, this is a sorting network. Thus, it is sufficient to prove the claims for inputs consisting only of zeros and ones.

It is also the case that the sequence $y_{\frac{n}{2}}, \ldots y_{n-1}$ is bitonic. The proof is, of course, essentially a copy of the proof for part **b**. To keep the problem short, I didn't ask you to show that.

5. **Energy Trade-Offs (20 points)** We have often noted that processor performance is limited by power dissipation. Consider the naïve implementation of dynamic programming from the lecture. Using this algorithm, the elapsed time to compute the editing distance between two strings of length N using P processors is:

$$\begin{aligned} T_{elapsed} &= t_{update} N^2, && \text{if } P = 1 \\ T_{elapsed} &= t_0 P + t_1 N + t_{update} \frac{N^2}{P}, && \text{if } P > 1 \end{aligned}$$

Note: I've made the simplification that $P - 1 \approx P$ for any $P > 1$, and absorbed some factors of 2 into the other constants.

Now, I'm going to consider the time **and** energy required for communication and computation. I'll assume that the time and energy required for computation can be scaled such that $E_{compute} T_{compute}$ is constant. In other words, if the processor runs at half the speed, it uses one half the energy per operation. This is the same as saying that the power consumption of the processor is proportional to the square of the speed at which it is running. I'll assume that the energy for communication is proportional to the time for communication, and that these values can't be scaled. With this model, we get:

$$\begin{aligned} T_{elapsed} &= T_{communicate} + T_{compute} \\ T_{communicate} &= 0, && \text{if } P = 1 \\ &= t_0 P + t_1 N, && \text{if } P > 1 \\ T_{compute} &= t_{update} \left(\frac{N^2}{\alpha P} \right) \\ E_{total} &= E_{communicate} + E_{compute} \\ E_{communicate} &= T_{communicate} \\ E_{compute} &= \alpha^2 T_{compute} \end{aligned}$$

In these equations, α says how fast the processor is run: if $\alpha = 1$, the processor is run at its "nominal" speed; if $\alpha = 2$, the processor is run at twice its nominal speed; if $\alpha = 0.5$, the processor is run at one half its nominal speed; and so on.

Now, assume that $N = 10,000$, $t_0 = 1000$, $t_1 = 10$, and $t_{update} = 1$.

(a) (**6** points) What are the time and energy if the computation is performed on one processor running with $\alpha = 1$?

Solution: $T_{communicate} = 0$, and $T_{compute} = 100,000,000$. Thus, $T_{elapsed} = 100,000,000$, and $E_{total} = 100,000,000$.

(b) (**7** points) What are the time and energy if the computation is performed on 100 processors with $\alpha = 0.1$?

Solution: $T_{communicate} = 200,000$; $T_{compute} = 10,000,000$; $E_{compute} = \alpha^2 T_{compute} = 100,000$. Thus, $T_{elapsed} = 10,200,000$, and $E_{total} = 300,000$.

(c) (**7** points) Let's say that I have a performance requirement of computing the editing distance between the two strings with an elapsed time of 10^7 time units. What values of α and P minimize the energy consumption?

Hint: I found that a mix of derivation and trying a few values works pretty well. I'll give significant partial marks for finding reasonably good approximations of α and P.

Solution: First, I'll figure out what the value of α is as a function of P:

$$\begin{aligned} T_{elapsed} &= t_0 P + t_1 N + t_{update} \frac{N^2}{\alpha P} \\ 10^7 &= 10^3 P + 10^5 + \frac{10^8}{\alpha P} \\ \alpha &= \frac{10^8}{(9.9 * 10^6 - 10^3 P) P} \end{aligned}$$

The total energy is

$$\begin{aligned} E_{total} &= t_0 P + t_1 N + \alpha t_{update} \frac{N^2}{P} \\ &= 10^3 P + 10^5 + \frac{10^{16}}{(9.9 * 10^6 - 10^3 P) P^2} \end{aligned}$$

I could try differentiating with respect to P, but the prospects look painful. A more practical approach is to try a few values of P, for each value of P, calculate the corresponding α, and then E. Here's the sequence that I tried:

P	α	E_{total}	
100	0.1020	302,041	
200	0.0515	325,773	
150	0.0684	295,584	
125	0.0818	290,473	
135	0.0759	291,190	
130	0.0787	290,583	
127	0.0806	290,440	← *optimum*
126	0.0812	290,445	
128	0.0799	290,459	

I conclude that the minimum energy is achieved with $P = 127$ and $\alpha = 0.0806$. I'll note that the optimum is quite flat. Furthermore, the model is very rough approximation of any plausibly real system. So, I'll accept for full credit any answer that comes to within 5% of optimal energy as long as the method for getting it was reasonable. That means that any P with $98 \leq P \leq 168$ is acceptable along with the corresponding α. So, you can get full credit even if you didn't spend a bunch of time punching different guesses for P into your calculator.
To entertain those who want to see how far you can get by differentiating and simplifying, I differentiated the expression for E_{total} with respect to P and got:

$$\frac{d}{dP}E_{total} \quad = \quad 10^3 - \frac{10^{16}(1.98*10^7 P - 3*10^3 P^2)}{(9.9*10^6 - 10^3 P)^2 P^4}$$

I'll assume that $0 < P < 9.9 * 10^3$, in which case the derivative is non-singular. It's equal to zero if

$$\begin{aligned} & 10^3 * (9.801 * 10^{13} - 1.98 * 10^{10} P + 10^6 P^2) P^3 - 10^{16}(1.98 * 10^7 - 3 * 10^3 P) &= 0 \\ \Leftrightarrow \quad & (9.801 * 10^7 - 1.98 * 10^4 P + P^2) P^3 - 10^7 (1.98 * 10^7 - 3 * 10^3 P) &= 0 \\ \Leftrightarrow \quad & P^5 - 1.98 * 10^4 P^4 + 9.801 * 10^7 * P^3 + 3 * 10^{10} P - 1.98 * 10^{14} &= 0 \end{aligned}$$

I don't see any obvious way to factor this polynomial. Numerically, it's easy to show that this polynomial has a root near 126.6839, which confirms the choice of $P = 127$ as optimal. Of course, if anyone has a way to make further progress with an analytical approach, I'll be happy to see what you come up with.

Remark: After the exam, a few students pointed out to me that it would have been more realistic if I had modeled the energy for computation as:

$$E_{compute} \quad = \quad \alpha^2 P T\_compute$$

– this includes a factor of P that wasn't included in the problem statement. I agree. Of course, I graded based on the stated problem, but I would have happily given extra credit if anyone had solved this "improved" problem and stated why it is better. Oh, right. It's better because the $\alpha^2 T\_compute$ model just takes into account the energy reduction on a single processor. However, the parallel version has P processors running, and we should account for the total power of all P processors combined.
With this revised model for $E\_compute$ the solution to part b has $E\_communicate = 200,000$ as before, but $E\_compute$ becomes 10,200,000, which yields $E\_total = 10,400,000$.

For part c, the formula for α as a function of P remains:

$$\alpha = \frac{10^8}{(9.9*10^6-10^3P)P}$$

Using revised model for *E_compute* yields:

$$\begin{aligned} E_{total} &= t_0P + t_1N + \frac{N^4}{(T\_elapsed-t_0P-t_1N)P}t_{update} \\ &= 10^3P + 10^5 + \frac{10^{16}}{(9.9*10^6-10^3P)P} \end{aligned}$$

Plugging in various guesses for P yields a minimum at $P = 999$ for which $\alpha = 0.0112$ and $E = 2,236,594$. It's not surprising that the minimum energy increases – the energy for computation for any give choice of P and α is larger. Also, it makes sense that the optimal value of P increases. We've increased the energy for computation; so there is a greater incentive to reduce the energy for computation at a cost of increasing the energy for communication.
For the amusement of those who prefer analytical approaches, differentiated the *E_total* with respect to P to get:

$$\frac{d}{dP}E_{total} = 10^3 - \frac{10^{16}(9.9*10^6-2*10^3P)}{(9.9*10^6-10^3P)^2P^2}$$

And $\frac{d}{dP}E_{total} = 0$ when

$$P^4 - 1.98*10^4P^3 + 9.801*10^7P^2 + 2*10^{10}P - 9.9*10^{13} = 0$$

I don't see an obvious way to factor this polynomial (sure, there's a closed form solution for factoring quartics,

http://en.wikipedia.org/wiki/Quartic_function,

but it's pretty messy, and I seriously doubt it would lead to an enlightening solution. A numerical solution shows that $P \approx 998.6921$ is a root of the polynomial above. This is consistent with the choice of $P = 999$ as the optimal number of processors.

6. **Potpourri** (20 points)

(a) (**4 points**) What is "idle processor overhead"? Give a short definition and a simple example.

Solution: This is a loss of performance relative to an ideal speed-up of P for a parallel comptuer with P processors due epochs when there are processors that are not performing any useful computation. Often, this happens at the beginning or end of a parallel computation when only one or a few processors are busy. Here are a few examples (one is sufficient):

- The MPI implementation of dynamic programming that we studied initially has only one processor busy. Each time, that processor completes a "block", it can dispatch work for another processor. Thus, the first processor must complete $P-1$ blocks before P processors can all be busy. A similar issue occurs at the end of the computation.
- The reduce operation uses a tree to combine results. This requires $\log_2 P$ phases of operation, and all processors are busy only for the first phase. In each successive round, half of the active processors become idle. Thus, on average $\frac{2}{\log P}P$ processors are busy. Similar scenarios arise for broadcast or scan operations.
- LU decomposition was described in a fair amount of detail in the book and in less detail in lecture as an algorithm that has idle processors at the end. As the computation progresses, a smaller and smaller fraction of the original matrix is being modified. A simple block allocation scheme results in $\frac{3}{4}$ of the processors being idle after half of the total compute time. Finer grained, block-cyclic work allocations can mitigate this problem.

(b) (**4 points**) What is "communication overhead"? Give a short definition and a simple example.

Solution: This is a loss of performance relative to an ideal speed-up of P for a parallel computer with P processors due to the time spent for communication. This time can includes CPU time for coordinating message transfers with a network interface (and context switches to run other tasks while waiting for message) and the network latency. Here are a few example:

- In the MPI implementation of dynamic programming that we studied, we saw that making the tableau blocks to small resulted in extra overhead to send and receive a large number of messages.
- In fact, pretty much every parallel algorithm involves some communication and the associated overhead. It's easy to find communication overhead in count-3s, matrix multiply, reduce and scan, etc. Any of these are acceptable examples.

(c) (**4 points**) What is "false sharing"? Give a short definition and a simple example.

Solution: False sharing occurs in a shared-memory multiprocessor when threads running on two different processors are repeatedly accessing different parts of the same cache block, and at least one of the threads is modifying the block. This causes cache block invalidations, transfers of ownership, and cache misses even though no data is actually being transfered between the threads.

The text book gave an example of false sharing with the count 3's program when the per-thread tallies were stored in a global array. Each thread updated a different element of this array. Because these elements could be on the same cache line, the extra cache misses arising from this false sharing made the parallel program slower than the sequential one. Many other examples are possible.

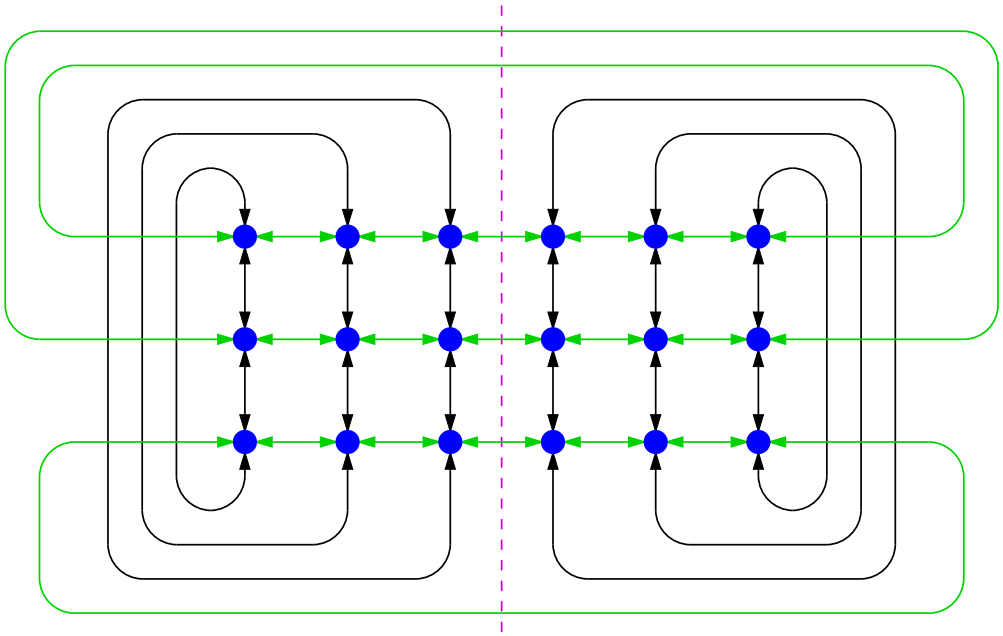

Figure 2: A 3×6 2-dimensional torus

(d) (**4 points**) What does SIMD stand for? Given a example of a processor architecture that exploits SIMD parallelism.

Solution: SIMD = Single Instruction Multiple Data.

A single instruction stream controls multiple functional units that each perform the same operation on different streams of data values. Example include:

- GPUs: for example nVidia calls it "SIMT" for "Single Instruction, Multiple Threads" because of some of the generalizations that they've made.
- MMX extensions: media extensions for the x86 and other processors exploit data parallelism in an SIMD fashion.
- The IBM/Sony/Toshiba Cell processor uses SIMD processing in its "synergistic processing units."
- and there are many other examples.

(e) (**4 points**) Draw a diagram illustrating a 3×6 2-dimensional torus. Assume that each link has a bandwidth of 1 Gbit per second in each direction. Divide the processors into two groups of nine such that each processor in the first group sends a 1 megabyte message to a processor in the second group. Draw a dashed line on your diagram to show how to split the processors into two groups that will maximize the time required to send these messages. If the time to send the messages is determined entirely by the bandwidth of the network (i.e., ignore all other overheads), how long does it take to send these nine messages from one group of processors to the other?

Solution: See Figure 2 for the figure. There are 6 links connecting the nine processors on the left from the nine on the right. These provide 9 $\mathit{Gbits/second}$ of total bandwidth. Each of the left processors sends 1 Mbyte to a processor on the right for a total of 72 Mbits of data transfered. Assuming that the time is entirely determined by the bandwidth constraint, it will take $7\ \mathit{Mbits}/(9\ \mathit{Gbits/sec}) = 12\ \mathit{ms}$ to send these messages.

Reduce and Scan Documentation

reduce(Leaf, Combine, Root)

A generalized reduce operation. The `Leaf` function is applied at each leaf of the process tree. The `Leaf1` function has no arguments – it operates on the local state the worker process in which it's applied. The results of these are combined, using a tree, using `Combine`. The `Combine` function has two arguments: the cumulative value for the left subtree of the current node, and the cumulative value for the right subtree. The `Combine` function produces the cumulative value for the subtree rooted at the current node. The `Root` function is applied to the final result from the combine tree to produce the result of this function.

scan(Leaf1, Leaf2, Combine, Acc0)

A generalized scan operation. The `Leaf1` function is applied at each leaf of the process tree. The `Leaf1` function has no arguments – it operates on the local state the worker process in which it's applied. The results of these are combined, using a tree, using `Combine`. The `Combine` function has two arguments: the cumulative value for the left subtree of the current node, and the cumulative value for the right subtree. The `Combine` function produces the cumulative value for the subtree rooted at the current node. The return value of the scan is the result of applying the `Combine` function at the root of the tree. Furthermore, the `Leaf2` function is applied in each worker process. The `Leaf2` function has one argument, `AccIn` which is the the result of `Combine` for everything to the left of this node in the tree. For the leftmost process, `Acc0` is used.

Often, `scan` is used for the local updates performed by `Leaf2`, and the return value (the result of applying `Combine` at the root of the process tree) is ignored. Of course, the results the applications of `Leaf1` and `Combine` are used to produce the arguments passed to `Leaf2`.