CpSc 448B

Final Exam

December 16, 2011

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.

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.

   (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.

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

   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 ith 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 jth 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.

   (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 $B, P, t_0, andt_1$.

   (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 $B = 1024$?

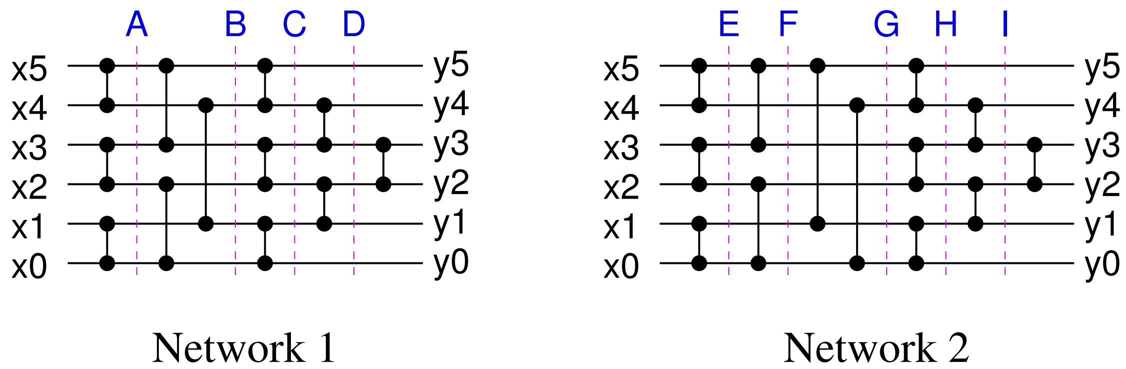

Figure 1: Two sorting networks

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.

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

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 $\forall 0 \leq i < \frac{n}{2}.\ y_i \leq y_{i+\frac{n}{2}}$.

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

   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:

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

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:

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

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 $\alpha = 1$?

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

(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.

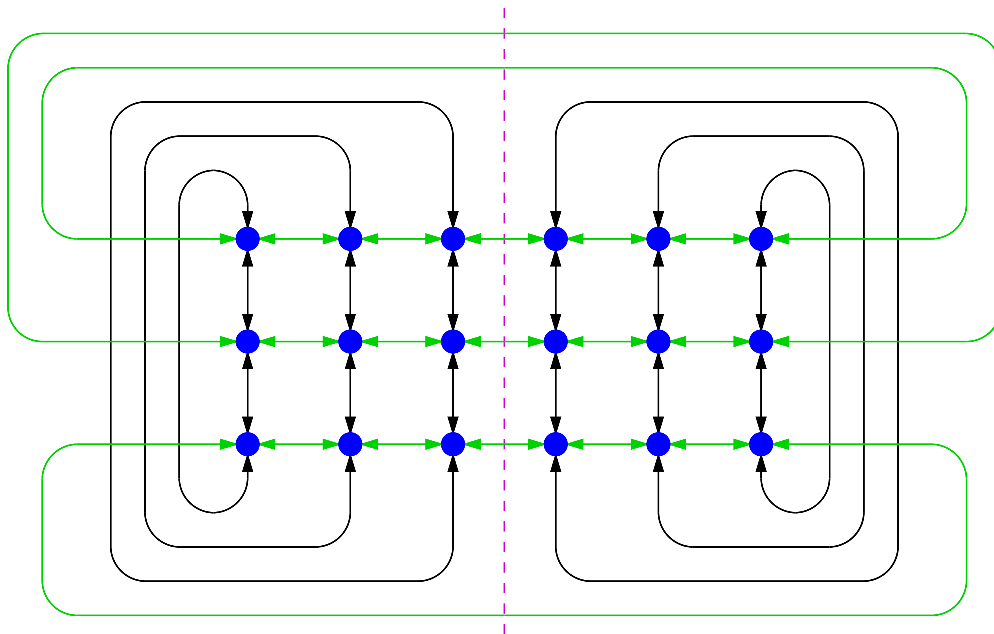

Figure 2: A 3×6 2-dimensional torus

6. **Potpourri** (**2**0 points)

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

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

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

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

   (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?

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`.