CpSc 448B — Final Exam — December 10, 2009

Questions

1. (**20 points**)
   Give short definitions for each of the terms below.

   (a) Block cyclic allocation
   A method for partitioning matrix elements between processors where the matrix is divided into more blocks than there are processors. These blocks are "indexed" in lexigraphical order (i.e. left-to-right, top-to-bottom), and block i is assigned to processor $i \bmod P$ where P is the number of processors. This is useful for computations such as LU factorization where rows and columns are eliminated from the matrix at each step, and an allocation with P blocks would leave some processors idle after a small amount of computation.

   (b) Condition variable
   Condition variables are used in shared-memory programs so that one thread can notify another when some "condition" holds. In particular, one thread can wait for a condition variable to receive a signal, and another thread can send a signal for that condition variable. We used condition variables in class in our implementation of a shared buffer for producer/consumer computations.

   (c) Deadlock
   A situation where there is a cycle of processes or threads, where each process is waiting for its predecessor in the cycle. For example, this could be waiting to receive a message, or waiting for a lock to be released. Because all processes in the cycle are blocked, none of them will be able to resume execution. Generally, deadlock is an error.

   (d) Sequential consistency
   A shared memory multiprocessor provides sequential consistency if all memory operations (loads and stores) performed by all processors have the same effect as if they were all performed in some sequential order that is consistent with the program order for each processor.

   (e) Speed-up
   The ratio of the time to execute a program sequentially divided by the time to execute it in parallel. Actually, I'll accept two versions. If T_s is the sequential time, and T_p is the parallel time then you can say that there is a speed-up of a factor of T_s/T_p or a speed-up of $((T_s/T_p) - 1) * 100\%$.

2. (**20 points**)
   Consider a one-dimensional array of $N * P$ elements distributed across P processors. In particular, processor 0 holds elements $0 \dots N-1$, and processor i holds elements $i * N$ through $(i+1) * N - 1$. Describe how to use reduce and or scan operations to find the longest sequence of consecutive 3's in the array, the length of this sequence, and the index of the first element of this sequence.

   Of course, you may define generalized reduce and scan operators in Erlang, use the versions from homework 0, or the versions from the book. Just state what you assume.

   I'll assume that there's a list Pids of P process identifiers such that lists:nth(i+1, Pids) is the process for "processor i" as described above. I'll assume a generalized scan function like in homework 0 with an parameter for the identity element of the combine function:

```
reduce(CombineFn, Ident, LeafFn, LeafArgs) ->
   % CombineFn: is an associative function that "combines" its two arguments
   % LeafFn:    a function that takes an argument list from LeafArgs and
                produces a value for CombineFn
   % LeafArgs = a list of argument lists – each argument list
                is a list of arguments for LeafFn.
```

My solution computes a tuple for each block in the array

{Left, Longest, Index, Length, Right}

where

Left: *The number of consecutive 3's starting at the left (lowest index or head of list) edge of the block;*

Longest: *The length of the longest run of 3's in the block;*

Index: *The position within the block of the leftmost*

Length: *The total length of this block; 3 in the longest block of 3's.*

Right: *The number of consecutive 3's ending at the right (highest index or last element of the list) edge of the block;*

The arguments to the scan are

LeafArgs: Pids.

LeafFn: *Send a message to the process. It replies with the a tuple of the form described above, assuming that it's first element has index 0.*

CombineFn: *Let* L *and* R *be the tuples for the left and right subtrees.* CombineFn *creates a new tuple* Z *as shown below:*

```
combineFn(L, R) -¿
    L_Left, L_Longest, L_Index, L_Length, L_Right = L,
    R_Left, R_Longest, R_Index, R_Length, R_Right = R,
    Z_Left = L_Left + if (L_Longest == L_Length) -> R_Left;
                        true -> 0
                      end,
    Z_Right = R_Right + if (R_Longest == R_Length) -> L_Right;
                          true -> 0
                        end,
    Z_Longest = lists:max([Z_Left, Z_Right, L_Longest, R_Longest, L_Right + R_Left]),
    Z_Index = case Z_Longest of
        Z_Left -> 0;
        Z_Right -> Z_Length - 1 - Z_Longest;
        L_Longest -> L_Index;
        R_Longest -> L_Length + R_Index;
        _ -> L_Length - 1 - L_Right
    end,
    Z_Left, Z_Longest, Z_Index, Z_Length, Z_Right.
```

3. (**20 points**)
Consider multipling a pair of $n \times n$ matrices on a machine with p processors. For simplicity, assume that n is a multiple of p and that p is a perfect square. Assume that we implement a parallel algorithm with p processes and that the time to send m matrix elements between two processes can be done in time $t_0 + a * m$. Assume that computing the product of an $m_1 \times m_2$ matrix with a $m_2 \times m_3$ matrix takes time $bm_1m_2m_3$. Now, consider two ways of computing $Z = X * Y$ on p processors where X and Y are both $n \times n$ matrices:

   (a) (**8 points**)
   Each process holds n/p rows of each matrix. Each process sends all of its rows for Y to every other process. Based on the rows that the processes receives, and its own rows for X, the process computes its rows for Z.

Using the model described above, how long should it take to compute Z?

The parallel computation time is

$$t(n) \quad = \quad (p-1)\left(t_0 + \tfrac{an^2}{p}\right) + \tfrac{bn^3}{p}$$

where the left term is the communication time and the right term is the time for computation. In more detail, the communication time comes from each process sending its block to $p-1$ other processes. The message consists of its block of y which is of size n^2/p. The computation time is because the total effort is bn^3, and each processor performs $1/p$ of the total computation.

(b) (**8 points**)
Each process holds a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block for X and Y, and will compute the corresponding block for Z. To do so, it sends its blocks for X to the $\sqrt{p}-1$ processes that need it, and likewise for Y. After receiving the blocks that it needs, the process computes its block of Z. Using the model described above, how long should it take to compute Z?

The parallel computation time is

$$u(n) \quad = \quad 2(\sqrt{p}-1)\left(t_0 + \tfrac{an^2}{p}\right) + \tfrac{bn^3}{p}$$

The formula is basically the same as for the previous case, but this time, the processor sends its block to each of the $\sqrt{p}-1$ processors in the same block-row as itself, and to the processors in the same block-column as itself.

(c) (**4 points**)
Is one method always better than the other, or does the choice depend on n, p, t_0, a, and/or b? Justify your answer either by showing that one is always clearly faster, or by giving the critical values for the parameters to decide the faster algorithm.

The second method is always faster (in the trivial case that $p = 1$, both are the standard sequential algorithm and run at the same speed). Both methods perform the same amount of computation on each processor. The difference is in the commnication. Note that in both cases, each processor holds n^2/p elements. With the first method, every processor sends all of its elements to all of the other processors. With the second method, a processor only sends its block to a subset of the other processors. Thus, the second method performs less communication and is faster than the first.

If you prefer formulas to prose:

$$\begin{array}{lrcll}
 & 2(\sqrt{p}-1)\left(t_0 + \frac{an^2}{p}\right) + \frac{bn^3}{p} & \leq & (p-1)\left(t_0 + \frac{an^2}{p}\right) + \frac{bn^3}{p} & \\
\Leftrightarrow & 2\sqrt{p} & \leq & p, & \textit{assuming } t_0, a, b, n, p > 0 \\
\Leftrightarrow & 4 & \leq & p &
\end{array}$$

By the assumption that p is a perfect square, this condition holds for all cases except the trivial ones that $p = 1$ or $p = 0$.

4. (**20 points**)
Let $P0$ and $P1$ be two Erlang processes, each of which has an array, A of N integers. Each process has a variable, OtherPid which is the PID for the other process. Now, consider the following Erlang code fragment:

```
A1 = lists:sort(A),
{Even, Odd} = unshuffle(A1),
OtherPid ! {self(), 0, Odd}, receive {OtherPid, 0, OtherOdd} -> ok,
A2 = lists:merge(Even, OtherOdd),
{FirstHalf, LastHalf} = lists:split(N div 2, A2),
OtherPid ! { self(), 1,
    if (MyPid < OtherPid) -> LastHalf; true -> FirstHalf end
},
receive {OtherPid, 1, OtherHalf} -> ok,
S = if
    (MyPid < OtherPid) -> lists:merge(FirstHalf, OtherHalf);
    true -> lists:merge(LastHalf, OtherHalf)
end,
```

The unshuffle function returns two lists: Even has the even-indexed elements of lists (starting from 0, very un-Erlang), and Odd has the odd-indexed elements.

Prove that at the end of executing the code above, the process with the smaller PID has a list of the smallest N elements of the two lists in ascending order, and the process with the larger PID has the largest N elements in ascending order.

Use the zero-one principle. Let $\mathsf{A1}_0$ denote the list $\mathsf{A1}$ for process $P0$, and likewise for $\mathsf{A1}_1$ and for other variables. Assume that $\mathsf{A1}_0$ has Z_0 zeros, and $\mathsf{A1}_1$ has Z_1 zeros. I'll write $zeros(X)$ to denote the number of zeros in list X. We now get:

$$\begin{array}{rcl} zeros(\mathsf{Even}_0) & = & \lceil \frac{Z_0}{2} \rceil \\ zeros(\mathsf{Odd}_0) & = & \lfloor \frac{Z_0}{2} \rfloor \\ zeros(\mathsf{Even}_1) & = & \lceil \frac{Z_1}{2} \rceil \\ zeros(\mathsf{Odd}_1) & = & \lfloor \frac{Z_1}{2} \rfloor \\ zeros(\mathsf{A2}_0) & = & \lceil \frac{Z_0}{2} \rceil + \lfloor \frac{Z_1}{2} \rfloor \\ zeros(\mathsf{A2}_1) & = & \lfloor \frac{Z_0}{2} \rfloor + \lceil \frac{Z_1}{2} \rceil \end{array}$$

We now consider two cases: $Z_0 + Z_1 \leq N$ and $Z_0 + Z_1 > N$.

$Z_0 + Z_1 \leq N$: *We now have*

$$\begin{array}{rcll} zeros(\mathsf{A2}_0) & = & \lceil \frac{Z_0}{2} \rceil + \lfloor \frac{Z_1}{2} \rfloor & \\ & \leq & \frac{Z_0+1}{2} + \frac{Z_1}{2}, & \lceil x/2 \rceil \leq (x+1)/2,\ \lfloor x/2 \rfloor \leq x/2 \\ & \leq & \frac{N+1}{2}, & Z_0 + Z_1 \leq N \\ & \leq & \frac{N}{2}, & zeros(\mathsf{A2}_0) \textit{ is an integer, } N \textit{ is even} \end{array}$$

From which we conclude that all the zeros of $\mathsf{A2}_0$ *go into* $\mathsf{FirstHalf}_0$*:*

$$zeros(\mathsf{FirstHalf}_0) \quad = \quad \lceil \tfrac{Z_0}{2} \rceil + \lfloor \tfrac{Z_1}{2} \rfloor$$

By a similar argument:

$$zeros(\mathsf{FirstHalf}_1) \quad = \quad \lfloor \tfrac{Z_0}{2} \rfloor + \lceil \tfrac{Z_1}{2} \rceil$$

Therefore,

$$\begin{array}{rcl} zeros(\mathsf{S}_0) & = & \lceil \frac{Z_0}{2} \rceil + \lfloor \frac{Z_1}{2} \rfloor + \lfloor \frac{Z_0}{2} \rfloor + \lceil \frac{Z_1}{2} \rceil \\ & = & Z_0 + Z_1 \end{array}$$

This means that all of the zeros end up in S_0 *which gets sorted, and* S_1 *is all ones.*

$Z_0 + Z_1 > N$: *We can apply the same argument as above, just counting ones instead of zeros and showing that all of the ones end up in* S_1.

5. (**20 points** + 10 extra credit)
The module `bank.erl` included in this packet is the Erlang source code for a simple model of bank accounts. Each account has an initial balance, and then a bunch of processes perform transfers between the accounts. Each transfer has a transfer Amount which is deducted from the balance of the FromAccount and added to the balance of the ToAccount. Obviously, the *total* amount of money summed over all accounts should be constant.

I tried running the program with eight accounts, and eight activity processes where each process performed 10,000 transfers. Initially, each account had a balance of $5,000, for a total of $40,000. After performing the transfers, the total was $26,181.

(a) (**5 points**)

Explain why this program does not work as intended.

There is a race condition when multiple processes are reading and writing the balance of the same account. For example, two processes could read the balance of account A, compute a new balance, and then both write their result to account A. The second write will nullify the effect of the first one creating an incorrect result.

(b) (**15 points**)

A solution to this problem is to add locking behaviour to the accounts. One way to do this is to modify the account process to perform the following interactions:

receive {read, Pid, Tag}: Send {Tag, Balance} to Pid (the current behavior).

receive {write, Amount}: Set the account Balance to Amount (the current behavior).

receive exit: Terminate the account process (the current behavior).

receive {lock, Pid, Tag}: Generate a "unique" Key (my solution calls random:uniform(1000000000)) and send {Tag, Key} to Pid. Subsequent requests will block unless they include the value of Key in the request (until an unlock) is performed.

receive {read, Pid, Tag, Key}: If Key matches the current key value for this account, send {Tag, Balance} to Pid. Messages of the form {read Pid, Tag} (without a key) should be blocked until the account is unlocked.

receive {write, Amount, Key}: If Key matches the current key value for this account, set the account Balance to Amount. Messages of the form {write Amount} (without a key) should be blocked until the account is unlocked.

receive {unlock, Key}: If Key matches the current key value for this account, go back to the original (unlocked) behaviour.

Write an account function that implements these behaviours.

Hint: My solution has an account function that comes in both a one-argument and a two-argument version.

```
account(Balance) ->                      % This version called when the
    receive                              % account is not locked.
        {read, Pid, Tag} ->
            Pid ! {Tag, Balance},
            account(Balance);
        {lock, Pid, Tag} ->
            Key = random:uniform(1000000000),
            Pid ! {Tag, Key},
            account(Balance, Key);
        {write, Amount} -> account(Amount);
        exit -> ok
    end.
account(Balance, Key) ->                 % This version is called when the
    receive                              % account is locked.
        {read, Pid, Tag, Key} ->
            Pid ! {Tag, Balance},
            account(Balance, Key);
        {write, Amount, Key} -> account(Amount, Key);
        {unlock, Key} -> account(Balance)
    end.
```

To simplify the revisions to transfer, I modified get to include a version with the Key argument. That was shown in `bank2.erl` that was included with the questions.

(c) (**5 points, extra credit**)
I implemented a solution to part (b), and modified the get and transfer functions as shown in module `bank2.erl` (included in this packet). The code deadlocked. Explain why?

Consider what happens if process $P0$ selects accounts $A1$ and $A2$ as its FromAccount and ToAccount respectively, while process $P1$ concurrently selects accounts $A1$ and $A2$ as as its ToAccount and FromAccount. Then $P0$ could lock $A1$ while $P1$ locks $A2$. After that, $P0$ tries to lock $A2$ but blocks. Likewise, $P1$ tries to lock $A1$ and blocks. Now, the processes are deadlocked. There are many other scenarios that lead to deadlock for this version of the code.

(d) (**5 points, extra credit**)
Modify the transfer function from module `bank2.erl` to correct the deadlock problem.

A solution is to order the locks. In my solution, I take advantage of the fact that Erlang has an ordering relation for PIDs. So, I modified transfer to lock the FromAccount and ToAccount by locking the lesser of the two PIDs first. Here's the code:

```
transfer(FromAccount, ToAccount, Amount) when FromAccount /= ToAccount ->
    if
        (FromAccount < ToAccount) ->
            FromKey = get(FromAccount, lock, from_acct),
            ToKey = get(ToAccount, lock, to_acct);
        true ->
            ToKey = get(ToAccount, lock, to_acct),
            FromKey = get(FromAccount, lock, from_acct)
    end,
    % continue as in the previous version
```