CpSc 418 **Homework 4** Due: Nov. 29, 2012, 11:59pm

5% extra credit if solution submitted by 11:59pm on Nov. 25.

Please submit your solution using the handin program as:
cs418 hw4

1. **Scan (20 points)**
   Consider the decaying average operation:

$$y_i = \sum_{j=1}^{i} \alpha^{i-j} x_j$$

   In Erlang:

```
average_d(List, Alpha) -> average_help(List, Alpha, 0).
average_help([], _, _) -> [];
average_help([X | Tl], Alpha, V) ->
   V2 = Alpha*V + X,
   [V2 | average_help(Tl, Alpha, V2)].
```

   (a) **(5 points)** Let `x = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]`. Draw a picture showing how decaying average can be computed using scan with four processors and $\alpha = 0.1$.
   You can draw your diagram neatly by hand, scan it, and include it in `hw4.pdf`; you can draw it using a drawing program of your choice, export it as a PDF file, and include it in `hw4.pdf`; or you can draw it as ASCII-art an include it in `hw4.pdf` or `hw4.txt`.

   (b) **(15 points)** Write a parallel version using the `wtree:scan` function. Here's the stub:

```
average_d(W, SrcKey, DstKey, Alpha) -> ok.
```

   where
   `W` is a pool of worker processes that form a tree (i.e. `W` was created by `wtree:create`).
   `SrcKey` is the key for accessing the distributed list of numbers to be averaged.
   `DstKey` is the key for saving the distributed result list. Note: your implementation should work correctly even in the case that `SrcKey` = `DstKey`.
   `Alpha` is the weighting factor for the decaying average.
   When the scan completes, each worker should have its piece of the result list in its process state with each sublist associated with the key `DstKey`.

2. 0-1 Principle – extended (**20 points**)
   In class, we defined a sorting network by starting with the identity network and adding compare-and-swap modules. In Erlang, we can define the compare-and-swap function as:

```
compare_and_swap(A, B) when A =< B -> {A, B};
compare_and_swap(A, B) -> {B, A}.
```

   Let

```
apply_to_tuple(Fun, {A, B}) -> {Fun(A), Fun(B)}
```

   The key to the proof was that for any monotonic function, `F`:

```
compare_and_swap(F(A), F(B))  =  apply_to_tuple(F, compare_and_swap(A, B))
```

If applying `F` to the operands of some operation `Op` produces the same result as applying `Op` to the original operands and then apply `F` to the results of `Op`, then we say that `F` *commutes* with `Op`. In particular, monotonic functions commute with compare-and-swap.

By showing that other operations commute with compare-and-swap, we can extend the 0-1 principle wider classes of networks:

(a) (**5** points) Let

```
swap(A, B) -> {B, A}
```

For any monotonic function, `F`, show that `F` commutes with `swap`.

This shows that the inputs and/or outputs of a sorting network can be permuted in an arbitrary way and the resulting network is still a sorting network. This is handy as it allows us to decompose a sorting network into simpler sub-networks.

(b) (**15** points) For sorted lists, `A` and `B`, let

```
my_merge(A, B) -> lists:split(length(A), lists:merge(A, B)).
```

For any monotonic function (i.e. non-decreasing), `F`, show that `F` commutes with `my_merge`.

This shows that if some sorting network, SN, sorts correctly, then we can replace the compare-and-swap modules of SN with `my_merge` modules. The values passed between `my_merge` values are sorted lists (or arrays). The sorting an merging operations performed on these lists or arrays can be done by any algorithm that we like. The sorting network shows us how to combine such operations into a parallel algorithm for sorting.

3. Parallel sorting (**40 points**)
Consider the following algorithm for sorting an array, A, of N elements using P processors. For simplicity, assume that N is a perfect square and N is a multiple of P^4, and P is even. A is arranged as a $\sqrt{N} \times \sqrt{N}$ array. At the end, we produce an array Z that has the same elements as A, and the elements of Z are sorted in lexicographical order:

$$(i_1 < i_2) \vee ((i_1 = i_2) \wedge (j_1 < j_2)) \quad \Rightarrow \quad Z(i_1, j_1) \leq Z(i_2, j_2)$$

The algorithm proceeds in seven "phases" described below and illustrated in Figures 1 and 2.

Phase 1: Divide A into P blocks.
The k^{th} block is composed of rows $\frac{k}{P}\sqrt{N} \ldots \frac{k+1}{P}\sqrt{N} - 1$ of A, and will be denoted as A_k. Each of these blocks has $\frac{N}{P}$ elements and can be sorted by sequential processor k in

$$O\left(\tfrac{N}{P}(\log N - \log P)\right) \quad = \quad O\left(\tfrac{N}{P}\log N\right)$$

time to create an array B_k. For k even, B_k is in ascending lexicographical order. For k odd, B_k is in descending lexicographical order.

Phase 2: For each m in $0 \ldots P-1$, each processor k sends columns $\frac{m}{P}\sqrt{N} \ldots \frac{m+1}{P}\sqrt{N} - 1$ of B_k to processor m. Processor m receives these blocks and assembles them into a $\sqrt{N} \times \frac{\sqrt{N}}{P}$ array. We'll write C_m to denote the array received by processor m. For example, the block received by processor m from processor k will form rows $\frac{k}{P}\sqrt{N} \ldots \frac{k+1}{P}\sqrt{N} - 1$ of C_m.

Phase 3: Processor k sorts C_k by columns to produce D_k. In other words, the elements of $D_k(\cdot, j)$ are the same as those of $C_k(\cdot, j)$ and the elements of $D_k(\cdot, j)$ are in ascending order. Each processor can do this in

$$O\left(\tfrac{\sqrt{N}}{P}\sqrt{N}\log(\sqrt{N})\right) \quad = \quad O\left(\tfrac{N}{P}\log(N)\right)$$

time.

Phase 4a: For each m in $0 \ldots P-1$, each processor k sends rows $\frac{m}{P}\sqrt{N} \ldots \frac{m+1}{P}\sqrt{N} - 1$ of D_k to processor m. Processor m receives these blocks and assembles them into a $\frac{\sqrt{N}}{P} \times \sqrt{N}$ array. We'll write E_m to denote the array received by processor m. For example, the block received by processor m from processor k will form columns $\frac{k}{P}\sqrt{N} \ldots \frac{k+1}{P}\sqrt{N} - 1$ of E_m.

Phase 4b: Cyclically shift all rows down by $\frac{\sqrt{N}}{2P}$ rows. We'll now say that processor k holds block F_k. Note that phases 4a and 4b could be combined into a single round of communication – I separated them to make them easier to describe.

Phase 5: For $k \in 1 \ldots (P-1)$, processor k sorts its block, F_k, into lexicographical order to produce G_k. This takes time $O\left(\frac{N}{P}\log(N)\right)$.

Phase 6: Cyclically shift all rows up by $\frac{\sqrt{N}}{2P}$ rows. We'll now say that processor k holds block H_k.

Phase 7: Processor k sorts its block, H_k, into lexicographical order to produce Z_k. This takes time $O\left(\frac{N}{P}\log(N)\right)$.

In this problem, you will show that the array Z is sorted into lexicographical order.

(a) (**10 points**) Show that at the end of phase 1, each block B_k has at most one dirty row.

(b) (**10 points**) Show that at the end of phase 3, all dirty rows of D are contained in a band of height at most $P/2$. Hint: each block C_k consists of P sub-blocks received from the P processors. As shown in part 3a each of these blocks has at most one dirty row. Now, recall our analysis of Shear sort from class.

(c) (**10 points**) Show that the operations in phases 5 and 7 will not make a clean row dirty.

(d) (**10 points**) Show that the band of dirty rows of D is contained within a block that is sorted in phase 5 or phase 7.

A few remarks:

- The total processing time (not counting communication) is $O(\frac{N}{P}\log N)$.
- We might look at the time for communication on HW5.
- In the figures, the small squares of the arrays correspond to data elements.
- I drew solid arrows to indicate the direction of increasing data values. A solid arrow includes the values of all of the elements that it crosses.
- Dashed arrows are "wrap-around" jumps to continue the sequence of ascending values.
- Which data is held by which processors is indicated by the color of the squares. Thus, changing the color scheme corresponds to sending and receiving data blocks. In particular, the cyclic shifting of rows to produce array F and H is indicated by shifting the colors for the processors (in the opposite direction).

4. Mutual Exclusion

In class, we have discussed Dekker's mutual exclusion algorithm which guarantees mutual exclusion for two clients. Lamport's "Bakery Algorithm" works with any number of clients. Figure 3 shows the algorithm for `N` clients. The algorithm models entering a bakery and taking a ticket with a number on it. When a number is called, the person with that ticket gets service. In Lamport's version, "taking a ticket" is done by checking the numbers on the tickets for every thread that is currently waiting or in its critical section. Let `v` be the maximum such ticket value. The thread trying to enter the lock gives itself a ticket number of `v+1`. The thread then spins, checking to see when all threads with lower ticket numbers have completed. When the thread has the lowest ticket number among those waiting, it is granted the lock.

The `for` loop at lines `7...9` can check the tickets for the other processors in any order, and other threads can change their ticket numbers while a thread is executing the loop. For each `j`, `v` is compared with a value that `ticket[j]` had *sometime* during the execution of the loop.

A remarkable feature of the algorithm is that it allows multiple threads to get the same ticket number! This is because there is no mutual exclusion for the process of checking tickets and choosing a number. If two threads

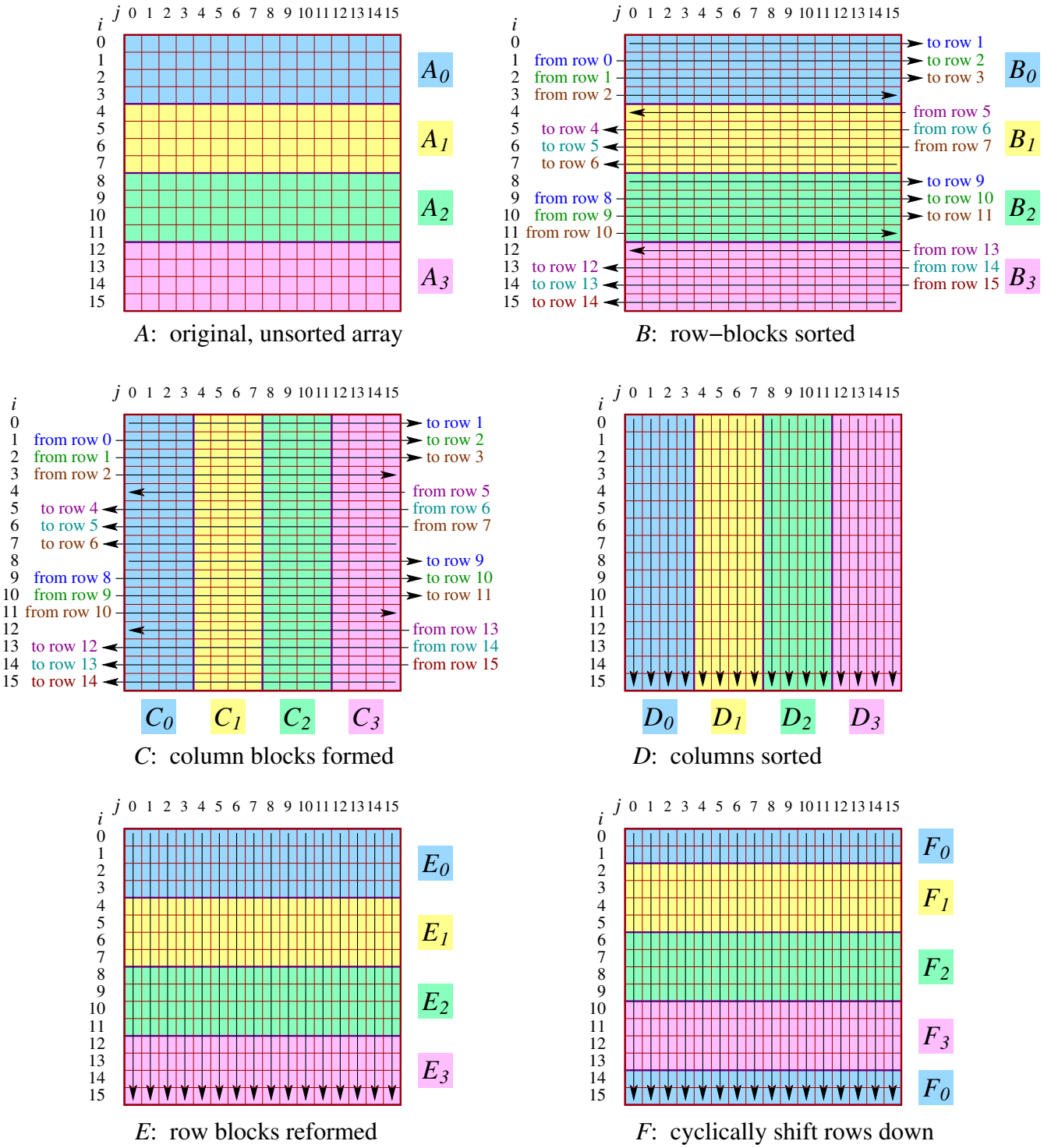
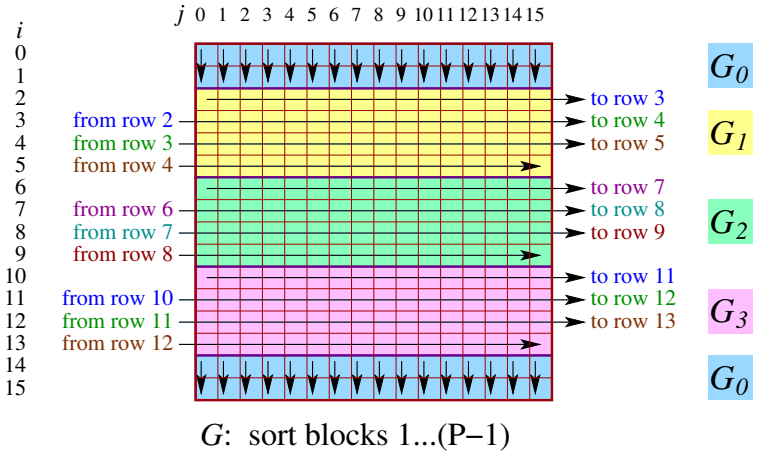
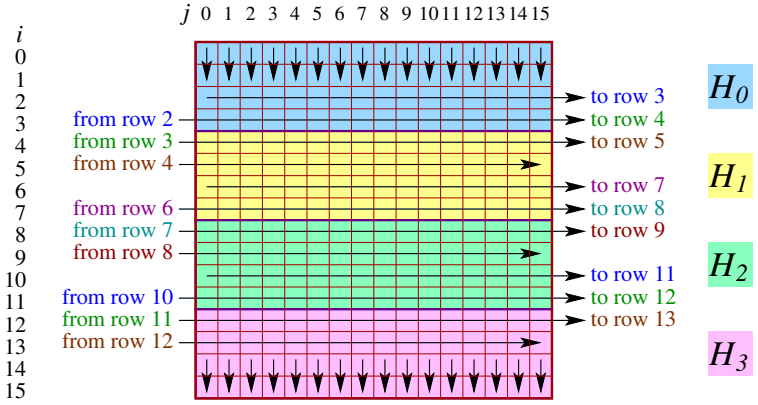


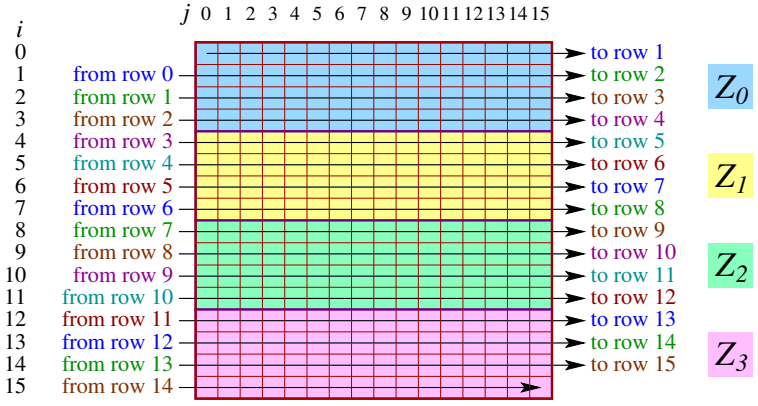
Figure 1: Sorting: phases 1...4



G: sort blocks 1...(P−1)



H: cyclically shift rows up



Z: sort row blocks

Figure 2: Sorting: phases 5...7

```
 0:    Initially:
 1:       ∀ i ∈ {0 ...(N-1)}:  flag[i] = false;
 2:       ∀ i ∈ {0 ...(N-1)}:  ticket[i] = 0;
 3:
 4:    lock(int i) { % thread i wants the lock
 5:       flag[i] = true; % let the world know my intention to acquire the lock
 6:       v = 0;
 7:       for j = 0..(N-1) in any order { % find the largest ticket issued so far
 8:          v = max(v, ticket[j]);
 9:       }
10:       ticket[i] = v+1; % take the next ticket number
11:       do {
12:          ok_to_enter = true;
13:          for j = 0..(N-1) in any order { % anyone with a smaller ticked waiting?
15:             if(flag[j] AND (j ≠ i)
16:                         AND (    (ticket[j] < ticket[i])
17:                                OR ((ticket[j] == ticket[i]) AND (j < i))))
18:                ok_to_enter = false; % we still need to wait
19:          }
20:       } while(NOT ok_to_enter);
21:    }
22:
23:    unlock(int i) { % thread i is releasing the lock
24:       flag[i] = false;
25:    }
```

Figure 3: Lamport's Bakery Algorithm

have the same ticket number, the one with the lower thread index goes first. Of course, that's a claim that needs a proof, and that's the point of this question.

For the questions below, assume that all threads use the lock correctly. This means that thread `i` calls `lock(i)` and `unlock(i)` in alternation zero or more times. The first such call is a call to `lock(i)`. Once the thread acquires the lock it must eventually call `unlock(i)`. We say that a thread is in its critical section when a call to `lock(i)` has returned and the thread has not yet called `unlock(i)`.

(a) (**10 points**) Show that the Bakery algorithm is deadlock free.
This means that if one or more threads are waiting to get the lock, some thread will eventually get it.
To show this, consider a state where no thread has the lock but one or more threads are waiting. This means that one or more threads are executing the `do...while` loop at lines 12...21 of the `lock` function. Show that there is one thread that will execute the body of this loop at most one more time before acquiring the lock.

(b) (**10 points**) Show that the Bakery algorithm issues grants in order.
This means that if thread `i1` has the lock with ticket value `t1`, and the next thread to acquire the lock is thread `i2` with ticket value `t2`, then either $t1 < t2$, or $t1 = t2$ and $i1 < i2$.
Note that deadlock freedom and in-order-granting ensures that all requests are eventually granted.

(c) (**10 points**) Show that the Bakery algorithm guarantees mutual exclusion.
This means that two threads cannot be in their critical regions at the same time. The grants-in-order property helps prove this one.

(d) (**10 points**) Show that if the statement

```
5:        flag[i] = true;
```

is moved until after the for loop at lines 7–9, then mutual exclusion is not guaranteed.

Note: for all of these, it's useful to note that each thread repeatedly cycles through the following four "states":

Idle: Starting at line 25 of `unlock` (having just set `flag[i]` to `false`), including any non-critical-section code of the application, and ending at line 5 of `lock` (just before setting `flag[i]` to `true`. Both ends are inclusive (i.e. the idle state includes lines 5 and 25). Note that a thread has no obligation to call `lock`.

Ticketing: Starting at line 6 of `lock` (having just set `flag[i]` to `true`) and continuing to line 10 (just before setting `ticket[i]`). A thread in the ticketing state must eventually complete these operation and enter the spinning state.

Spinning: Starting at line 11 of `lock` (having set `ticket[i]` and ending at line 20 if `ok_to_enter` is true. A thread in the spinning state must eventually try the loop body (i.e. it can't take forever to get to its next instruction).

Critical: Starting at line 21 of `lock`, including the thread's critical section, and ending at line 24 of `unlock`. A thread that is in its critical section must eventually finish the critical section and call `unlock`.

You can state that each thread cycles through these four "states" (or code regions) in the order described without any further proof. Now, some handy lemmas (you need to prove them if you want to use them):

- `flag[i]` is false when thread `i` is in the idle state and true otherwise.
- We will say that thread `i` *precedes* thread `j` if `ticket[i]` $<$ `ticket[j]`, or `ticket[i]` $=$ `ticket[j]` and $i < j$. If thread `i` is idle or critical and thread `j` is spinning, then it can be shown that thread `i` precedes thread `j`.