CpSc 418 **Final** December 12, 2012

100 points

1. **DLS talk (5 points)**
   Judea Pearl's talk: "The Mathematics of Cause and Effect" made frequent use of the term "counterfactual." What word or phrase below best describes "counterfactual" as Pearl used it in his talk:

   (a) a false statement;

   (b) an ancient story or account such as the Garden of Eden or the destruction of Sodom;

   (c) a paradox;

   (d) a "what if?" scenario;

   (e) a rebuttal to an argument.

   Solution: d.

2. **Short definitions (20 points).** Give a **short** definition for each term below. Give a brief (one or two sentences) definition **and** a simple example (one or two sentences).

   (a) deadlock

   Solution:
   A cycle of processes or threads where each is blocked waiting for the next process in the process in the cycle. Example 1: thread 0 holds a lock that thread 1 is trying to acquire and vice-versa. Example 2: Process 1 is waiting to receive a message from process 2, and will then send something to process 2, and vice-versa.

   (b) livelock

   Solution: A situation where two or more processes are busy interfering with each other in a way such that none of them make any progress. Example: several processes could be attempting to acquire the same lock. Each tries to get the lock, sees that another process is trying to get the lock, backs off and tries again.

   (c) the zero-one principle

   Solution: If a sorting network sorts all inputs consisting only of 0's and 1's correctly, then it sorts all inputs (arbitrary values of any type with an ordering relation) correctly.

   (d) termination detection

   Solution: Determining that an ensemble of processes have all completed. Example, the method in Stern & Dill's distributed model checking algorithm that checks message-sent and message-received counts (and suppresses sending new messages) when checking to see if all of the worker processes are done.

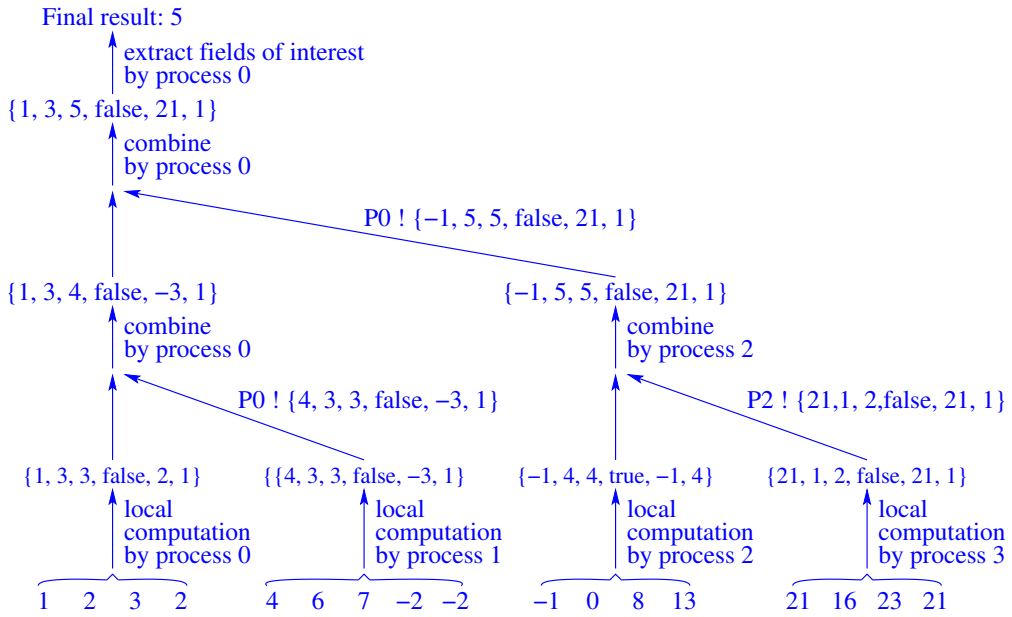

Figure A-Q3.a: Example of the reduce.

3. **Reduce (25 points)** Given a list of numbers, report the length of the longest non-descending sequence, in other words the length of the longest sequence of the form $x_i, x_{i+1}, \ldots x_{i+k}$ such that $x_i \leq x_{i+1} \leq \cdots \leq x_{i+k}$. For example,

```
longest_ascending([1, 2, 3, 2, 4, 6, 7,-2, -3, -1, 0, 8, 13, 21, 16, 23, 21])
```

is 6.

(a) (**5 points**) Draw a figure showing how this computation can be performed with four processes using a reduce. Assume that the list is distributed over the processes as shown below:

Process 0: [1, 2, 3, 2]
Process 1: [4, 6, 7, -2, -3]
Process 2: [-1, 0, 8, 13]
Process 3: [21, 16, 23, 21]

Solution: See Figure A-Q3.a.

(b) (**5 points**) Describe the type for the value that you will pass up the tree to do the reduce. For example, "a tuple of three elements where the first element is…", or "an integer", or "a list of …", etc.

Solution:

If the local portion of the list is non-empty, then a triple of the form:

`{LeftVal, LeftLen, MaxLen,Span, RightVal, RightLen}`

where:

`LeftVal` is the leftmost value under this subtree;

`LeftLen` is the length of longest ascending sequence starting at the leftmost element (`LeftLen` = 1 if the leftmost value is greater than the next value.);

`MaxLen` is the length of longest ascending sequence for this subtree;

`Span` is `true` iff the entire subtree is one ascending sequence;

`RightVal` is the rightmost value under this subtree; and

`RightLen` is the length of longest ascending sequence ending at the rightmost element of the subtree.

Otherwise, an empty tuple.

(c) (**5 points**) Sketch a function to use at the leaves of the tree to perform this reduce. You can write it in erlang, C, Java, or any reasonable resemblance of any of those languages.

Solution:

```
leaf([]) -> {};
leaf([LeftVal | T]) -> Leaf({LeftVal, 1, 1, true, LeftVal, 1}, T).
leaf(V, []) -> V;
leaf({LV, LL, ML, Sp, RV, RL}, [H | T]) when (H >= RV) and Sp ->
   leaf({LV, LL+1, ML+1, Sp, H, RL+1}, T);
leaf({LV, LL, ML, Sp, RV, RL}, [H | T]) when (H >= RV) and (RL == ML) ->
   leaf({LV, LL, ML+1, Sp, H, RL+1}, T);
leaf({LV, LL, ML, Sp, RV, RL}, [H | T]) when (H >= RV) ->
   leaf({LV, LL, ML, Sp, H, RL+1}, T);
leaf({LV, LL, ML, Sp, RV, RL}, [H | T]) ->
   leaf({LV, LL, ML, false, H, 1}).
```

The rules for `leaf/2` say:

If the current value of the list, `H`, is greater than or equal to the previous value, `RV`, and the entire list as been ascending, then increment `LL` (i.e. `LeftLen`), `ML` (i.e. `MaxLen`), and `RL` (i.e. `RightLen`);

Otherwise, if the current value of the list is greater than or equal to the previous value and the ascending sequence at the right is the longest one so far, then increment `ML` and `RL`; `RL` (i.e. `RightLen`);

Otherwise, if the current value of the list is greater than or equal to the previous value, increment `RL`.

If the list is empty, return the tuple that we've generated in the previous steps.

(d) (**5 points**) Sketch a function to use to combine values from subtrees. Use the same notation as you chose for part c.

Solution:

```
combine({}, Right) -> Right;
combine(Left, {}) -> Left;
combine({LV1, LL1, ML1, Sp1, RV1, RL1}, {LV2, LL2, ML2, Sp2, RV2, RL2}) ->
   Extend = RV1 =< LV2,
   { LV1,
     LL1 + if (Sp1 and Extend) -> LL2; true -> 0 end,
     max(max(ML1, ML2, if (Extend) -> RL1 + LL2; true -> 0)),
     Sp1 and Sp2,
     RV2,
     RL2 + if (Sp2 and Extend) -> RL1; true -> 0 end
   }
```

(e) (**5 points**) Sketch a function to generate the final answer given the result of the combine at the root.

Solution:

```
root({}) -> 0;
root({_LV, _LL, ML, _Sp,, _RV, _RL} -> ML.
```

Each of your functions descriptions should be short.

4. **Peril-L and Pthreads (25 points)** Peril-L has full/empty variables. There isn't a direct equivalent in POSIX threads. Figure 2 shows a one way that full/empty variables could be implemented using POSIX threads.

   (a) (**10 points**) Write the body for function `FE_int_get`.

   Solution:

```
// FE_int_get: Get the value of *ef.
// If *ef is empty, block until it is full.
int FE_int_get(struct FE_int *ef) {
   int val;
   pthread_mutex_lock(&(ef->lock));
   while(!ef->full)
      pthread_cond_wait(&(ef->cond), &(ef->lock));
   val = ef->value;
   ef->full = FALSE;
   pthread_cond_signal(&(ef->cond));
   pthread_mutex_unlock(&(ef->lock));
   return(val);
}
```

   Likewise, Peril-L doesn't provide mutexes or condition variables. Show that you can implement a mutex using Peril-L full/empty variables. Figure 1 shows an outline of the code that you should write. Note that I'm not asking you to write the functions to allocate, initialize, or free your mutex structs.

   (b) (**5 points**) Define a Peril-L `struct FE_mutex` for a mutex type that is implemented using one or more full/empty variables. In Peril-L, a variable whose name ends with an apostrophe is a full/empty variable, for example, `int v';`. Your struct can also use regular variables that are, presumably, global because this struct is used for synchronization. Note that Peril-L uses C-syntax plus the Peril-L extensions. Any reasonable approximation to Peril-L's syntax will be accepted.

   Solution:

```
struct FE_mutex {
   bool fe';
   bool is_locked;
}
```

   (c) (**5 points**) Write the function body for `FE_mutex_lock`. The textbook points out that it is a usage error if a thread attempts to acquire a lock that it already has. POSIX-threads doesn't check for this; so, you don't have to either.

   Solution:

```
void FE_mutex_lock(struct FE_mutex *m) {
   m->fe' = TRUE; // will block until the mutex is free.
   m->is_locked = TRUE; // see FE_mutex_unlock
}
```

   (d) (**5 points**) Write the function body for `FE_mutex_unlock`. It is an error to attempt to unlock a mutex that is not presently locked. Your code should call `error("bad unlock");` in this situation.

   Solution:

```
void FE_mutex_unlock(struct FE_mutex *m) {
   if(!m->is_locked) error("bad unlock");
   m->is_locked = FALSE;
   int toss = m->fe'; // empty fe to release the lock
}
```

```
struct FE_mutex {
    // You declare the fields.
};

// FE_mutex_lock(m): acquire mutex m.
void FE_mutex_lock(struct FE_mutex *m) {
    // You write the function body.
}

// FE_mutex_unlock(m): release mutex m.
void FE_mutex_unlock(struct FE_mutex *m) {
    // You write the function body.
}
```

Figure 1: Implementing a mutex using full/empty variables.

```
struct FE_int {
    int value;
    boolean full;
    pthread_mutex_t lock;
    pthread_cond_t cond;
};

// FE_int_set: Set the value of *ef to val.
// If *ef is full, block until it is empty.
void FE_int_set(struct FE_int *ef; int val) {
    pthread_mutex_lock(&(ef->lock));
    while(ef->full)
        pthread_cond_wait(&(ef->cond), &(ef->lock));
    ef->value = val;
    ef->full = TRUE;
    pthread_cond_signal(&(ef->cond));
    pthread_mutex_unlock(&(ef->lock));
}

// FE_int_get: Get the value of *ef to val.
// If *ef is empty, block until it is full.
int FE_int_get(struct FE_int *ef) {
    // You get to write get!
}
```

Figure 2: Implementing full/empty variables using POSIX threads

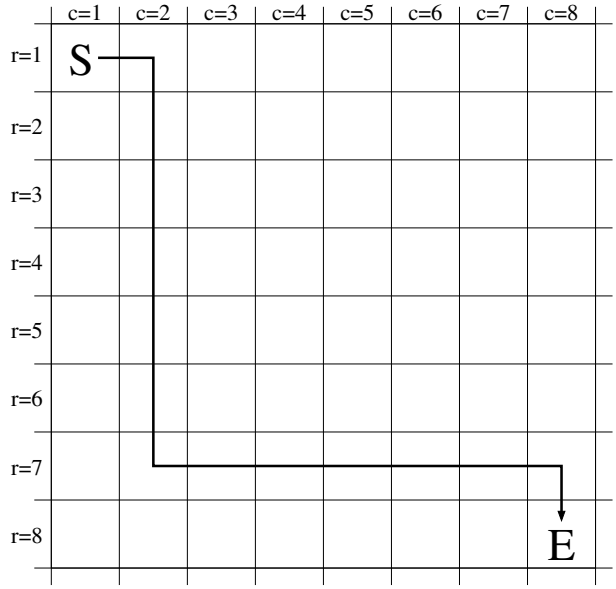

Figure 3: Critical path for parallel editing distance computation

5. (**27 points**) Consider the parallel implementation of dynamic programming for computing editing distance from the November 1 lecture. . . .

   In lecture, we modeled the time for this computation assuming that a processor takes time t_{up} to update one cell of the tableau, and that the total for the sender and receiver to send and receive a message of m tableau elements is $t_0 + t_1 m$. With this model, the time for the sequential computation is $t_{seq} = t_{up}N^2$ and the time for the parallel computation is

$$t_{par} = (2P-1)\left(\frac{N}{P}\right)^2 t_{up} + 2(P-1)\left(t_0 + \frac{N}{P}t_1\right)$$

   (a) (**5** points) Assume that $t_{up} = 20$ns, $t_0 = 200$ns and $t_1 = 10ns$ (reasonable values on a shared-memory implementation using POSIX threads or Java threads. Note: 1ns = 10^-9 seconds). What is the speed-up if $N = 1000$ and $P = 16$?

   Solution:

$$\begin{aligned} t_{seq} &= N^2 t_{up} \\ &= 1000^2 20\text{ns}, && N = 1000, t_{up} = 20\text{ns} \\ &= 20\text{ms} \end{aligned}$$

$$\begin{aligned} t_{par} &= (2P-1)\left(\frac{N}{P}\right)^2 t_{up} + 2(P-1)(t_0 + \frac{N}{P})t_1 \\ &= (2*16-1)\left(\tfrac{1000}{16}\right)^2 20\text{ns} + 2*(16-1)*(200\text{ns} + \tfrac{1000}{16})10\text{ns}, && P = 16, t_0 = 200\text{ns}, t_1 = 10\text{ns} \\ &= 2.45\text{ns} \end{aligned}$$

$$\begin{aligned} SpeedUp &= \frac{t_{seq}}{t_{par}} \\ &= 8.17 \end{aligned}$$

   Note: There was a right-parenthesis in the wrong place in the equation for t_{par} in the original version of the problem. If you used that formula, you would get (something whose units don't make sense, but for I'll ignore than to obtain)

- If 200ns is taken as (dimensionless) $200 \cdot 10^{-9}$, then $t_{par} = 2.44\text{ms}$, and $SpeedUp = 8.19$.
- If 200ns is taken as (dimensionless) 200, then then $t_{par} = 2.5\text{ms}$, and $SpeedUp = 7.998$.

(b) (**5** points) Show that in the limit that $N \to \infty$, the speed-up goes to $P/2$.

Solution:

$$\begin{aligned}
\lim_{N\to\infty} SpeedUp &= \lim_{N\to\infty} \frac{t_{seq}(N)}{t_{par}(N)} \\
&= \lim_{N\to\infty} \frac{N^2 t_{up}}{(2P-1)(N/P)^2 t_{up} + 2(P-1)(t_0 + (N/P)t_1)} \\
&= \lim_{N\to\infty} \left((2P-1)/P^2 + 2(P-1)(t_0/N + t_1/P)/N\right)^{-1} \\
&= \frac{P}{2 - \frac{1}{P}}
\end{aligned}$$

For moderate to large values of P, this is roughly $P/2$. I'll accept either answer.

We can divide the tableau into smaller blocks. For any integer $M > 1$, let each block have size $(N/(MP)) \times (N/(MP))$. Now, the entire tableau consists of MP rows of MP columns of these blocks. Assume that N is divisible by MP. The revised algorithm (for processor k) is:

```
for(int m = 0; m < M; m++) {
    c = m*(N/P) + k;
    top_row[J_c] = initial values;
    for(int r = 0; r < P; r++) {
        if(c > 1)
            receive values for T(Ī_r, (c-1)*(N/P)) from process c-1;
        update B(r,c);
        if(c < N)
            send values for T(Ī_r, c*(N/P)) to process c+1;
    }
}
```

(c) (**4** points) Draw the critical path for this computation when $P = 8$ and $M = 2$. You may use the template from Figure 4 when drawing your answer.

Solution: See Figure A.Q5-c. There are many possible critical paths. This one shows a path where process 2 is busy for most of the path. All blocks that are traversed by solid-line portions of the blue line are on the path; other blocks, including those under dashed portions of the blue line are not on the critical path. On this path, process 1 computes $B(1,1)$, then process 2 computes all of its blocks except for thelast one, $B(16,14)$. Processes 3 and 4 then compute $B(15,15)$, $B(15,16)$, and $B(16,16)$.
This path has the properties that as many blocks as possible have $2 \le j < MP$ – these blocks require both a send and a receive during their computation, thus the communication time is maximized. Furthermore, by including most of the computations for process 2 in the path, the computation time is not affected by whether or not process 2 time-shares when it has multiple blocks that can be updated. The same total comptuation time is needed either way.
There are many other ways to choose a critical path that has the properties described above. Any correct answer will get full credit.

(d) (**5** points) Derive a formula for the parallel execution time as a function of N, P, M, t_0, t_1, and t_{up}.

Solution: The critical path includes the updates of $M^2P + (P-1)$ blocks. Each block consists of $\frac{N}{MP} \times \frac{N}{MP}$ cells that must be updated. Thus, the computation time to update a block is $\frac{N^2}{M^2P^2} t_{up}$, and the computation time on the critical path is:

$$\begin{aligned}
t_{compute} &= (M^2P + (P-1))\frac{N^2}{M^2P^2} t_{up} \\
&= \left(1 + \frac{1}{M^2P}\left(1 - \frac{1}{P}\right)\right)\frac{N^2}{P} t_{up}
\end{aligned}$$

All of these block updates involve one receive operation and one send except for the update of $B(1,1)$ which only involves a send, and the update of $B(MP,MP)$ which only involves a receive. Thus there are $M^2P+(P-2)$ receives and $M^2P+(P-2)$ sends on the path. Each message communicates $N/(MP)$ values. This yields a communication time of

$$\begin{aligned} t_{msg} &= (M^2P+(P-2))\left(t_0+\tfrac{N}{MP}t_1\right) \\ &= NMt_1+\tfrac{N}{M}\left(1-\tfrac{2}{P}\right)t_1+(M^2P+P-2)t_0 \end{aligned}$$

The total time for the parallel computation is the sum of the time for computation plus the time for communication:

$$\begin{aligned} t_{par} &= t_{compute}+t_{msg} \\ &= \tfrac{N^2}{P}t_{up}+\tfrac{N^2}{M^2P}\left(1-\tfrac{1}{P}\right)t_{up}+NMt_1+\tfrac{N}{M}\left(1-\tfrac{2}{P}\right)t_1+(M^2P+P-2)t_0 \end{aligned}$$

(e) (**4** points) What is the speed-up with $M=8$ and the same parameters as for part a?

Solution: Each block is $\frac{N}{MP}\times\frac{N}{MP}=7.8125\times 7.8125$ elements. You might ask how we can have a fractional number of elements in a block? That's a reasonable question; I'm just taking it as a reasonable approximation for the next part when we consider large values of N, M, and P. Of course, I'll give full credit to any solution that handles this is a reasonable way. The time for a block update is

$$\begin{aligned} t_{blk} &= \tfrac{N^2}{M^2P^2}t_{up} \\ &= \tfrac{1000^2}{8^2 16^2}20\text{ns} \\ &= 1.22\mu\text{s} \end{aligned}$$

There are $M^2P+(P-1)=1039$ blocks on the critical path; so the total compute time is $1039*1.2207\mu\text{s}=1.2683\text{ms}$. I'll also use my formula for $t_{compute}$ above and get

$$\begin{aligned} t_{compute} &= \left(1+\tfrac{1}{M^2*P}\left(1-\tfrac{1}{P}\right)\right)\tfrac{N^2}{P}t_{up} \\ &= \left(1+\tfrac{1}{8^2*16}\left(1-\tfrac{1}{16}\right)\right)\tfrac{1000^2}{16}20\text{ns} \\ &= 1.2683\text{ns} \end{aligned}$$

The two calculations agree – I'm happy.
There are 1038 sends and receives of 7.8125 values. Each send+receive takes time

$$t_0+7.8125t_1=200\text{ns}+7.8125*10\text{ns}=278.125\text{ns}$$

The total time for sends and receives is $1038*278.125\text{ns}=288.7\mu\text{s}$. I'll also use my formula for t_{msg} above and get

$$\begin{aligned} t_{msg} &= NMt_1+\tfrac{N}{M}\left(1-\tfrac{2}{P}\right)t_1+(M^2P+P-2)t_0N*M*t_1+(N/M)(1-2/P)*t_1+(M^2*P+P-2) \\ &= 1000*8*10\text{ns}+\tfrac{1000}{8}\left(1-\tfrac{2}{16}\right)10\text{ns}+(8^2*16+16-2)*200\text{ns} \\ &= 288.7\mu\text{s} \end{aligned}$$

Again, the two calculations agree, and I'm happy.
The total time is

$$\begin{aligned} t_{par} &= t_{compute}+t_{msg} \\ &= 1.557\text{ms} \end{aligned}$$

I'll also use my formula for t_{msg} above and get

$$\begin{aligned} t_{par'} &= t_{compute}+t_{msg} \\ &= \tfrac{N^2}{P}t_{up}+\tfrac{N^2}{M^2P}\left(1-\tfrac{1}{P}\right)t_{up}+NMt_1+\tfrac{N}{M}\left(1-\tfrac{2}{P}\right)t_1+(M^2P+P-2)t_0 \\ &= ((1000^2)/16)*20\text{ns}+\tfrac{1000^2}{8^2*16}(1-(1/16))*20\text{ns}+1000*8*10\text{ns}+(1000/8)(1-(2/16))10\text{ns}+(8^2 \\ &= 1.557\text{ms} \end{aligned}$$

Good, that works out too.
Finally,

$$\begin{aligned} SpeedUp' &= \frac{t_{seq}}{t_{par'}} \\ &= \frac{20\text{ms}}{1.557\text{ms}} \\ &= 12.85 \end{aligned}$$

(f) (**4** points) Show that in the limit as $N \to \infty$, M can be chosen so that the speed-up will converge to P.

Solution: From part (c) we have:

$$t_{par} = \frac{N^2}{P} t_{up} + \frac{N^2}{M^2 P}\left(1 - \frac{1}{P}\right) t_{up} + NMt_1 + \frac{N}{M}\left(1 - \frac{2}{P}\right) t_1 + (M^2 P + P - 2)t_0$$

and the problem statement gave us that $t_{seq} = N^2 t_{up}$. We want $lim_{N\to\infty} \frac{t_{seq}}{t_{par}} = P$. Equivalently, we want to choose M so that

$$\begin{aligned} & lim_{N\to\infty} \frac{N^2}{M^2 P}\left(1 - \frac{1}{P}\right) t_{up} + NMt_1 + \frac{N}{M}\left(1 - \frac{2}{P}\right) t_1 + (M^2 P + P - 2)t_0 &= o((N^2/P)t_{up}) \\ \Leftrightarrow \quad & lim_{N\to\infty} \frac{1}{M^2} + \frac{MP}{N} + \frac{P}{NM} + \frac{M^2 P^2}{N^2} &= o(1) \end{aligned}$$

If $P = o(N)$ then let $M = \sqrt{\frac{N}{P}}$, and the conclusion follows. If P is not $o(N)$, then we have an example of excessive parallelism which may be fun for algorithms on a PRAM, but it not the topic of this course.

6. **Sorting on a Mesh (28 points).**
Let a be an array with of N values, $[a_0, a_2, \ldots, a_N - 1]$. Every operation that I describe in this problem can be implemented using sorting networks. Thus, we can assume that every element of a has a value of 0 or 1.

If $N = N_1 N_2$, then we can arrange the elements of a in a two dimensional array, A, where

$$\begin{aligned} A_{i,j} &= a_{N_2 * i + j}, && \text{if } i \text{ is even} \\ A_{i,j} &= a_{N_2 * (i+1) - (j+1)}, && \text{if } i \text{ is odd} \end{aligned}$$

This scheme arranges a into a "snaking" in A with even indexed rows going left-to-right, and odd-indexed rows going right-to-left.

For simplicity, assume that N_1 is a power of 2 greater than 1. We will say that a row of A is "clean" if all of its elements have the same value, and "dirty" if the row contains at least one 0 and at least one 1.

In the first step of our sorting algorithm, we will sort all even-indexed rows to be ascending to the right, and all odd-indexed rows to be ascending to the left, i.e.:

$$\begin{aligned} &\forall 0 < j < N_2.\ A(i, j-1) \le A(i, j), && \text{if } i \text{ is even} \\ &\forall 0 < j < N_2.\ A(i, j-1) \ge A(i, j), && \text{if } i \text{ is odd} \end{aligned}$$

(a) (**5 points**) Show that initially, the number of clean rows of A can be any value from 0 to N_1 (inclusive).

Solution: Assume $N_2 \ge 2$. Let $0 \le d \le N_1$ be the desired number of dirty rows. Let a be the sequence with:

$$\begin{aligned} a_i &= 1, && \text{if } i \bmod N_2 = 0 \text{ and } i/N_2 < d \\ &= 0, && \text{otherwise} \end{aligned}$$

The first d rows of A have one 1, and $N_2 - 1$ 0's, and the remaining rows are all zero. Thus, A has d dirty rows.

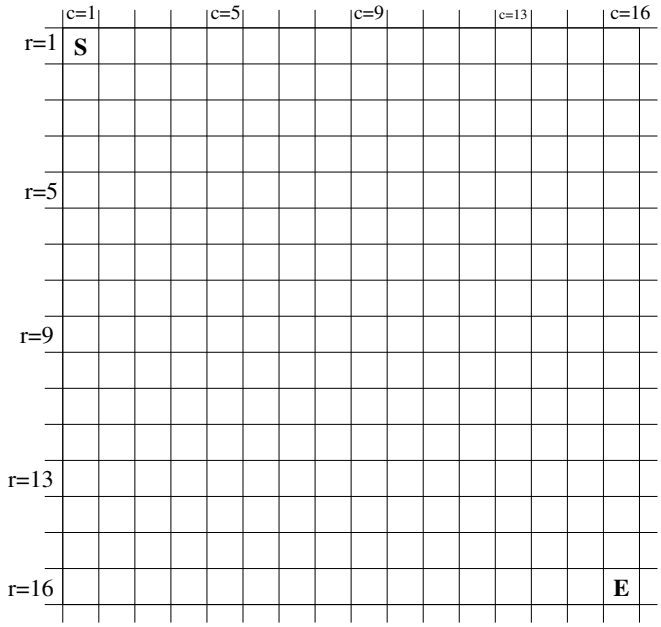

Figure 4: Critical path template for revised parallel editing distance computation

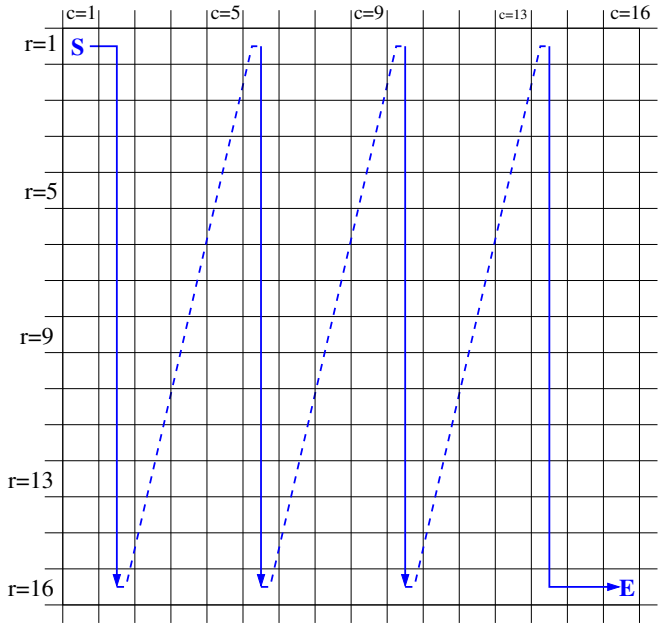

Figure A.Q5-c: Path for revised parallel editing distance computation

(b) (**5 points**) For every even numbered i with $0 \leq i < N_1$, do a compare-and-swap of $A(i,j)$ and $A(i+1,j)$ for all $0 \leq j < N_2$. Show that after the compare-and-swap, at least one of row i or row $i+1$ must be clean.

Hint: consider what happens if the total number of zeros in rows i and $i+1$ is greater than the total number of ones. What if the number of ones is greater than the number of zeros?

Solution: If the total number of zeros in rows i and $i+1$ is greater than or equal to number of 1's then the total number of zeros is greater than or equal to N_2. Because i is even, row i is sorted ascending to the right. Let

$$\begin{array}{rcll} j_0 & = & \min_j \text{ s.t.} A(i,j) = 1, & \text{if } \exists j \in \{0, \ldots N_2 - 1\} \text{s.t.} A(i,j) = 1 \\ j_1 & = & \min_j \text{ s.t.} A(i+1,j) = 0, & \text{if } \exists j \in \{0, \ldots N_2 - 1\} \text{s.t.} A(i+1,j) = 0 \end{array}$$

The total number of zeros in row i is j_0, and the total number of zeros in row $i+1$ is $N_2 - j_1$. Thus,

$$\begin{array}{rcll} j_0 + N_2 - j_1 & > & N_2, & \text{total number of zeros} > N_2 \\ j_0 > j_1 & N_2 & & \end{array}$$

This means that for all $0 \leq j < N_2$, at least one of $A(i,j)$ or $A(i+1,j)$ must be zero. After the compare and swap operation, all elements of $A(i,j)$ must be 0, and row i is clean.
If the total number of zeros in rows i and $i+1$ is less than the number of ones, then a similar argument shows that after the compare-and-swap, all elements of $A(i+1,j)$ must be 1, and row $i+1$ is clean.
With either case, at least one of row i or row $i+1$ must be clean.

(c) (**5 points**) Instead of doing the compare-and-swap from part b, sort each column into ascending order, i.e.:

$$\forall 0 < i < N_1.\ A(i-1,j) \leq A(i,j)$$

Show that after all of the columns are sorted in this way, A has at least $N_1/2$ clean rows.

Solution: After sorting the columns, each column consists of a sequence of zeros followed by a sequence of ones. The height of the sequence of zeros is equal to the number of zeros in the unsorted column. Performing a compare-and-swap as in part (b) does not change the number of zeros in each column; thus, we could perform this compare and swap on each pair of rows before the column sort and not change the outcome of the sort. This produces at least $N_1/2$ clean rows. For every row that is all 0's before the column sort, there must be a row that is all 0's after the column sort (these rows "sink" to the bottom). Likewise, row's of 1's float to the top.
Combining these two observations, A must have at least $N_1/2$ clean rows after the column sort.

(d) (**8 points**) Show that if the following algorithm is executed:

```
for k = 1 to 1 + log2 N1 do
   forall i in (0..(N1-1)) do
      if(i is even)  sort row i of A ascending to the right;
      else sort row i of A descending to the right;
   end
   forall j in (0..(N2-1)) do
      sort column j of A into ascending order;
end
```

array A has at most one dirty row.

Solution: As shown above, after the first iteration of the `for k` loop, A must have at least $N_1/2$ clean rows. The rows of zeros at the bottom of the array will be unaffected by further row or column sorts. Likewise for the rows of ones at the top. After the first iteration, A will have at

most $N_1 - N_1/2 = N_1/2$ dirty rows, and these rows are the only ones modified by subsequent iterations of the `for k` loop.
In the same manner, each successive iteration of the `for k` loop will reduce the number of dirty rows by a factor of two. More precisely, at the end of iteration `k`, A will have at most $\lceil N_1/2^{-k} \rceil$ dirty rows. At the end of the $\log_2 N_1$ iterations, A will have at most one dirty row. The final iteration sorts this row (and the column sort will do nothing).

(e) (**5 points**) Show that at the end of step d, A is sorted into ascending "snake" order (i.e. a is sorted into ascending order when mapped to A as described above).

Solution: Let $0 \le i_1, i_2 \le N_1$ and $0 \le j_1, j_2 < N_1$. Let

$$\begin{array}{rcll} m_1 & = & N_2 * i_1 + j_1, & \text{if } i_1 \text{ is even} \\ & = & N_2 * (i_1 + 1) - (j_1 + 1), & \text{if } i_1 \text{ is odd} \\ m_2 & = & N_2 * i_2 + j_2, & \text{if } i_2 \text{ is even} \\ & = & N_2 * (i_2 + 1) - (j_2 + 1), & \text{if } i_2 \text{ is odd} \end{array}$$

I'll show that if $m_1 \le m_2$, then $A(i_1, j_1) \le A(i_2, j_2)$. First, I'll show that if $i_1 > i_2$, then $m_1 > m_2$. If i_1 is even, then either i_2 is odd or $i_1 - i_2 \ge 2$, thus

$$\begin{array}{rcl} m_1 - m_2 & \ge & N_2 * i_1 + j_1 - \max(N_2 * (i_1 - 2) + j_2, N_2 * ((i_1 - 1) + 1) - (j_2 + 1)) \\ & \ge & N_2 * i_1 + j_1 - N_2 * i_1 - \max(j_2 - 2 * N_2, -(j_2 + 1)) \\ & \ge & j_1 - \max(-N_2 - 1, -j_2 - 1) \\ & \ge & j_1 + 1 \\ & \ge & 1 \end{array}$$

Thus, if i_1 is even and $i_1 > i_2$, then $m_1 > m_2$. A similar argument shows that if i_1 is odd and $i_1 > i_2$, then $m_1 > m_2$. Thus, if $i_1 > i_2$, then $m_1 > m_1$.
Now, assume $m_1 \le m_2$, then $i_1 \le i_2$ as just shown. If $i_1 < i_2$, then at least one of row i_1 or i_2 must be clean, and it follows that $A(i_1, j_1) \le A(i_2, j_2)$. If $i_1 = i_2$, then if i_1 is even then $j_1 \le j_2$, and if i_1 is odd then then $j_1 \ge j_2$. Consider the case when i_1 is even. Row i_1 is sorted in the final step into ascending order. Because $j_1 \le j_2$, $A(i_1, j_1) \le A(i_2, j_2)$. Otherwise, i_2 is odd; $j_1 \ge j_2$, row i_1 is sorted into descending order; and $A(i_1, j_1) \le A(i_2, j_2)$. Thus, if $m_1 \le m_2$, $A(i_1, j_1) \le A(i_2, j_2)$, and $a(m_1) \le a(m_2)$. The two-dimensional array A has been sorted into "snake order," and the one-dimensional array a has been sorted into ascending order.
Note: This algorithm is known as "Shear Sort" see

- I.D. Scherson, S. Sen: "Parallel sorting in two-dimensional VLSI models of computation." *IEEE Transactions on Computers* vol. C-38, no. 2, pp. 238-249 (1989)
- http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/twodim/shear/shearsorten.htm

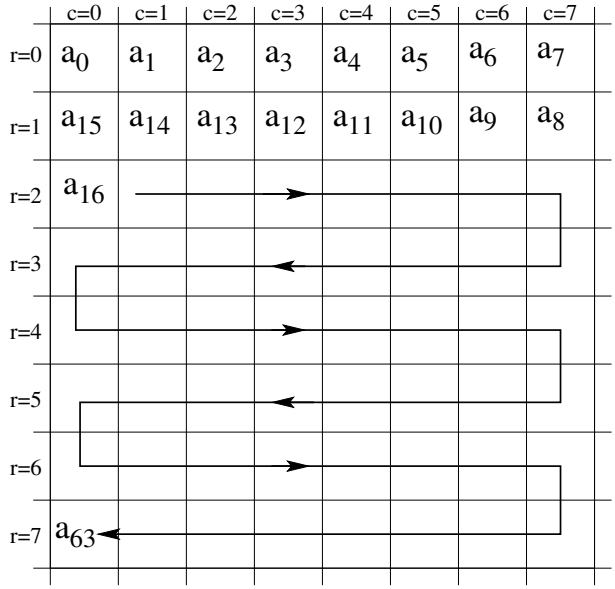

Figure 5: Snake-ordering of the elements of an array