18.435/6.848 Theory of Parallel and VLSI Computation **Fall 1995**

Solutions to Assignment #1

TA: Daniel Lewin *Prof. Leighton*

Remarks:

In this class, we design parallel algorithms that run on specific networks. For example, Gaussian elimination on a grid, or parallel prefix on a tree with N leaves. These algorithms are nice, but what happens if you have a parallel machine with some other kind of connection network? Can you use the algorithms we designed in class?

Many times the answer is "Yes!", and the technique for doing so is a central paradigm of parallel algorithms. The general idea is to *simulate* the network that you *want* with the network that you *have*. The nice thing about this method is that you can forget about the inner workings of the algorithm. Instead, you can concentrate on showing that the network you're stuck with can efficiently simulate the network you want (e.g. with constant slowdown).

Once you know how to do the simulation, you can just use the "off the shelf" algorithm from the local Sears! No need to rewrite and reprove the algorithm! The slowdown is just the slowdown incurred because of the simulation.

A good example of how the simulation method comes in handy is in problem 9. Take a look at the solution for the details.

Problem 1

Assume that the instances can be run in parallel. We assign P/X processors to each job (if $P/X \leq 1$ then each processor will be responsible for more than one job).

Denote the M-processor algorithm as Algorithm 1 and the M^2-processor algorithm as Algorithm 2. Moreover, let

T_1 = Time taken to solve the problem X times by Algorithm 1
T_2 = Time taken to solve the problem X times by Algorithm 2

We have:

$$\begin{aligned} T_1 &= M\frac{M}{\min\{P/X, M\}} \\ T_2 &= \sqrt{M}\frac{M^2}{\min\{P/X, M^2\}} \end{aligned}$$

Since we cannot utilize more than MX processors for Algorithm 1, and likewise we cannot utilize more than M^2X processors for Algorithm 2.

Equating T_1 and T_2, and solving for P/X we see that it is best to use Algorithm 1 if $P/X \leq M^{3/2}$, otherwise it is best to use Algorithm 2.

Problem 2

First look at a normal binary counter as shown in Figure PS1.1a). It takes input $(0, 1)$. The running sum is stored in the linear array with LSB in the left most processor.

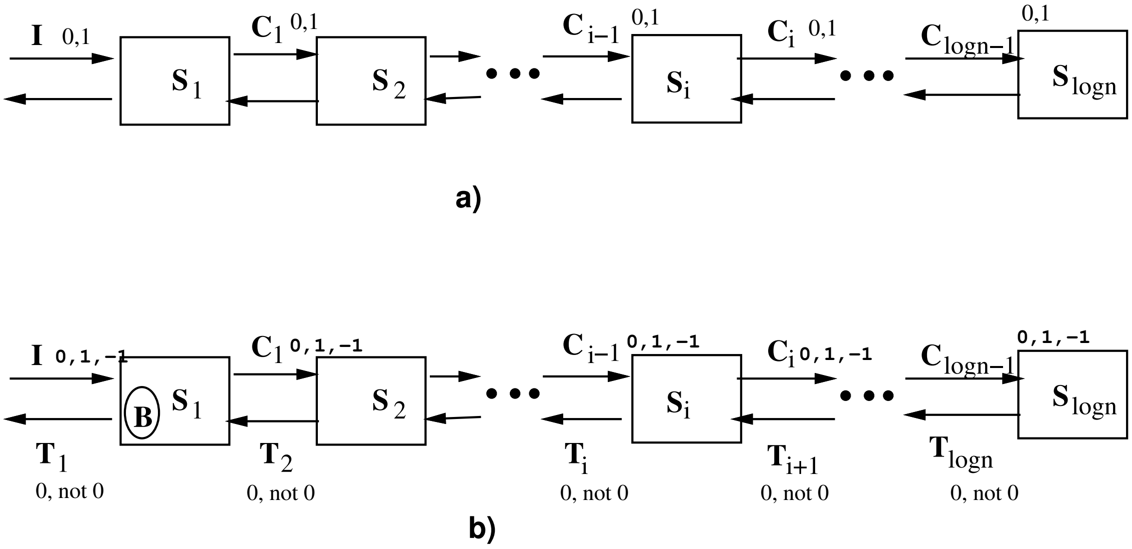

Figure PS1.1: a)Binary counter b)Up/Down counter

As shown in Figure PS1.1b), the up/down counter is similar to a normal counter except that it keeps a special sign bit B in the left most cell indicating the sign of current total. If $B = $ " $-$ ", we flip the sign of the input. So the rest of the array is acting as if the total is always non negative. Since the counter can get input -1, we can have -1 carries - or what is commonly called in the advanced literature a "borrow".

For each cell i, we define S_i=stored value, C_{i-1}=carry from left($I = C_0$ is the input sequence), C_i=carry to the right, T_{i+1}=input from right (one of $0, \neg 0$), T_i=output to the left. Following is the algorithm:
Initially, $B = $ " $+$ ", $S_i = C_i = T_i = 0$, $\forall i$.
At step j, do

- Input $I = x_j$.
- If $T_1 = 0$, $B = $ " $+$ " and $I = -1$, then, $B \leftarrow$ " $-$ ".
  If $T_1 = 0$, $B = $ " $-$ " and $I = 1$, then, $B \leftarrow$ " $+$ ".

- If B = " $-$ ", flip the sign of I.
- $S_i \leftarrow S_i + C_{i-1}$
  if $S_i = -1$, then $S_i \leftarrow 1$, $C_i \leftarrow -1$
  if $S_i = 2$, then $S_i \leftarrow 0$, $C_i \leftarrow 1$
  otherwise, $C_i \leftarrow 0$.
- If $S_i = C_i = T^0_{i+1} = 0$, then $T_i \leftarrow 0$, otherwise, $T_i \leftarrow \neg 0$.
  (Here we use T^0_{i+1} to indicate that it is the value calculated in the last iteration. we will argue below that this will still give us the right answer.)
- Output T_1.

Next, we will use induction to prove that this seemingly simple algorithm works. First it is easy to see that B always keeps the right sign given the right calculation for the T_1 bit. So, now we only need to prove that T_1 is right. Inductive hypothesis is: Given that the current total is non negative, $T_i = 0$ iff the sum of total input to the $\log n - i - 1$-cell linear array starting from cell i to cell $\log N$ is 0 (We call this sum SUM_i in the following proof). The base case is the linear array of a single cell, which is trivial. Now let us prove that the $\log N$ cell array starting at cell 1 will output $T_1 = 0$ iff $\sum_{k=1}^{j} x_k = 0$:

($\Rightarrow$):
From the algorithm, it is easy to see that $T_1 = 0$ iff $S_1 = C_1 = T^0_2 = 0$ after adding input x_j. $C_1 = 0$ means that there will be no carry input to cell 2, therefore T_2 would not change after this iteration. By induction we know that $SUM_2 = 0$. Therefore, $\sum_{k=1}^{j} x_k = S_1 + 2 * SUM_2 = 0$.

($\Leftarrow$):
If $T_1 = \neg 0$, then either $S_1 = 1$, or $C_1 = 1$ or -1, or $T^0_2 = \neg 0$.

- $S_1 = 1$: $\sum_{k=1}^{j} x_k = S_1 + 2 * SUM_2 \neq 0$, whatever SUM_2 might be.
- $S_1 = 0, C_1 = -1$: not possible, because $S_1 = 0$ can only come from $S^0_1 = 1, I = 1$ or -1, C can only be 1 or 0 in either case.
- $S_1 = 0, C_1 = 1$: We are considering only the case that B = " $+$ ". Therefore, before input C_1 to cell 2, $SUM^0_2 \geq 0$, and $\sum_{k=1}^{j} x_k = S_1 + 2 * C_1 + 2 * SUM_2 \neq 0$.
- $S_1 = 0, C_1 = 0, T^0_2 = \neg 0$: No input to cell 2 at this iteration, T_2 will be unchanged, i.e., $T_2 = \neg 0$, by induction, $SUM_2 \neq 0$, therefore, $\sum_{k=1}^{j} x_k = S_1 + 2 * C_1 + 2 * SUM_2 \neq 0$.

Problem 3

We first employ the simulation method and assume that the network is of the form shown in figure PS1.2. Clearly a single comb can simulate this network with constant slowdown.

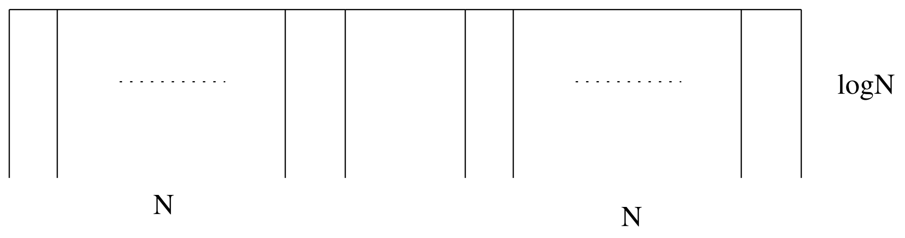

Figure PS1.2: The double comb network. Two copies of the comb network side-by-side. A single edge connects the "backbones" of the two combs.

The left hand comb contains the unsorted numbers - a number in each tooth of the comb. The right hand side of the comb accumulates the numbers in sorted order. The numbers pass from the left side to the right in encoded form. We need to show what the code is, how it is constructed by the comb on the left and how it is decoded by the comb on the right.

The Code:

Let a_i be the number if i's $(0 \leq i \leq N-1)$ in the input. The code is a sequence of bits where the number of zeros immediately before the i'th 1 is a_{i-1}. For example the encoding of the sequence: $1, 1, 3, 3, 4, 5, 5$ is $1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1$. Thus, the encoding of N numbers has $2N-1$ bits in it.

Constructing the Code:

Each tooth of the comb on the left hand side is actually an Up/Down counter which is initialized to the value of the input number.

Initially all of these Up/Down counters are "active".

The code is constructed along the backbone of the comb. The left most processor produces a sequence of 1's which move from left to right along the backbone. When a 1 passes an active column, the Up/Down counter decrements. If the Up/Down counter becomes 0, a 0 is inserted into the sequence on the backbone before the 1, and the Up/Down counter becomes "inactive". In order to insert the 0 into the sequence, the top processor acts as a buffer for 1 bit. Since each top processor only has to insert one bit into the stream, this can clearly be done with $O(1)$ memory. The resulting sequence of bits which pass between the left hand comb and the right hand comb is the encoding of the input.

(One detail is that the Up/Down counter has a delay of one clock before the output can be read. We can slow down the sequence on the backbone by a factor of two so that we have time to read the output of the counter and insert a 0 if needed.)

Decoding the Code:

Each tooth in the right hand comb is a regular binary counter which is initialized to 0. All these counters are initially "active".

The code is fed into the left side of the comb. When an active counter sees a 1 it increments and passes the 1 to the right. When an active counter sees a 0 it becomes "inactive", and does not propagate the 0 to the right. Otherwise, inactive counters propagate all values to the right.

In the end, the i'th counter will contain the number of 1's seen before the i'th zero. Thus, the comb contains the numbers in sorted order.

Running Time

The time to create the code is $O(2N-1)$ steps, and the time to decode is $O(2N-1+\log(N))$. These times overlap, but even if they did not, the total time is $O(N)$.

Lower Bound:

We look at the edge in the middle of the backbone of the comb network. Assume that numbers are sorted by the network from left to right. Consider the set of inputs where the right side of the comb contains $N/2$ zeros and the left side of the comb contains $N/2$ distinct non-zero numbers. Every set of values $\{a_1, a_2, \ldots, a_{N/2}\}$ on the left side must cause a *different* sequence of bits to pass over the central edge in the backbone because the final configuration on the right in each case if different. The total number of such sets is:

$$\binom{N^2}{N} \geq \left(\frac{N^2}{Ne}\right)^N = \left(\frac{N}{e}\right)^N$$

In order to distinguish between this many cases, at least:

$$\log\left(\frac{N}{e}\right)^N = O(N \log N)$$

bits need to pass over the central edge.

Problem 4

In this problem we have to coordinate an N-cell linear array of bit processors so that they all fire simultaneously. We assume that each processor contains a constant amount of information about its physical position in the array. Thus, the leftmost and rightmost processors know that they are on the ends, and the processors in the middle know that they are in the middle. We assume that the processors are supposed to sit in a wait state for an arbitrarily long period of time until the leftmost processor receives a "fire" signal from some external source. We would like all of the processors to enter a special fire state $\theta(N)$ steps after the fire signal.

We give a high level description of a divide-and-conquer algorithm to solve this problem. The idea is to split the list exactly in half, and then recursively apply the algorithm to each half. In order to the find the middle element of the array, the leftmost processor sends two signals to the right, one traveling 3 times as fast as the other. When the faster signal reaches the right end of the array, it bounces back. If the number of processors in the array is odd, the two signals meet in the middle processor. This processor now acts as an end processor in both halves of the split array and sends signals to recursively fire both halves. If the number of processors in the array is even, the two signals pass over each other between the two middle processors. Each processor sends signals to recursively fire its half of the array. The base condition is when a processor sends out the fast signal, and receives it back in the next time step. When this happens, the processor tells both its neighbors to fire in the next step. It is intuitive, and in fact easy to prove by induction, that the processors all enter the special fire state simultaneously.

The algorithm requires about $(3/2)(N + N/2 + N/4 + \cdots + 1) = 3N$ message passing steps after the first fire signal.

Problem 5

Given a double Eulerian cycle in a network we can embed a $2E$ node linear array along the path so that each edge is used at most 2 times (once in each direction), and each node is assigned to simulate no more than d nodes in the array. Each processor simulates no more than d virtual processors, so the slowdown is d. Moreover, every processor simulates some node in the array.

There are several ways to interpret the extension of the Firing Squad Problem to any N-cell (connected) bounded degree network (The Firing Mob Problem). We make the strongest set of assumptions here. We assume that each node in the network contains a constant amount of information about its physical position in the network. This information may be computed off-line. We assume that a single processor has been designated to receive the fire signal.

The trick to the Firing Mob Problem is to embed a linear list in the network, as is proven possible above , and then to apply the algorithm for the Firing Squad Problem to this list. The embedding of the list is computed off-line. Break the cycle at the designated start node by removing an edge. A processor may appear a constant number of times in the resulting list. It will simulate each appearance as if it were a separate (or virtual) processor.

It is important that $d = O(1)$ because the simulation requires $O(d)$ memory and each processor has only constant memory. (In addition, the slowdown is about d so the "mob" fires in time $O(Nd)$ and we want this to be $O(1)$).

This is a good example of the "simulation method" discussed at the beginning of this solution set.

Problem 6

Suppose, without loss of generality, that $N \leq M$. I will use the term *line* to denote either a column or a row, so the $M \times N$ mesh contains $M + N$ lines. The key idea in the proof is that the removal of fewer than N edges leaves too many lines of edges entirely intact. Let B be a set of edges whose removal bisects the mesh, and suppose B contains fewer than N edges. Since each edge in B lies in either a row or column, the edges in B lie in at most $N - 1$ different lines. Therefore the removal of B leaves at least $M + N - (N - 1) = M + 1$ of the $M + N$ lines intact. Furthermore, by the pigeonhole principle, there must be at least one row, say row i, and one column, say column j, intact. Thus, the vertices in the $M + 1$ intact lines are all connected to each other since a single intact row i contains a vertex in each intact column, and vice versa, a single intact column j contains a vertex in each intact row. We use a simple counting argument to show that the intact lines contain more than half of the vertices in the mesh. Let S be the set of vertices in the intact lines. Each intact line contains at least N vertices, so to get a lower bound on $|S|$, we count each of the at least N vertices in the $M+1$ intact rows and columns, and then divide by two since a vertex may be counted once in its column and once in its row. We have

$$|S| \geq N(M+1)/2 > \lceil NM/2 \rceil$$

for $N \geq 2$. But if S contains more than $\lceil NM/2 \rceil$ vertices, then the removal of B did not bisect the graph, a contradiction. Therefore, B cannot bisect the graph and contain fewer than N edges.

Problem 7

The algorithm has two stages:

Stage 1:

Input the $N \log N$ entries into the N leaves in blocks of $\log N$. Each leaf has $\log N$ storage so this is possible. Each leaf then computes sequentially the prefixes of the inputs it has. All this takes $O(\log N)$ time.

Stage 2:

Run parallel prefix on the tree where the inputs are the complete prefixes of the blocks at each leaf (computed in stage 1). Each leaf prepends the prefix computed by this stage to the prefixes computed in the first stage. All this takes $O(\log N)$ steps.

Addition:

We can now apply this trick to add two $N \log N$-bit numbers on a N-leaf complete binary tree in $O(\log N)$ time, since we have already seen how to implement carry lookahead as a parallel prefix computation.

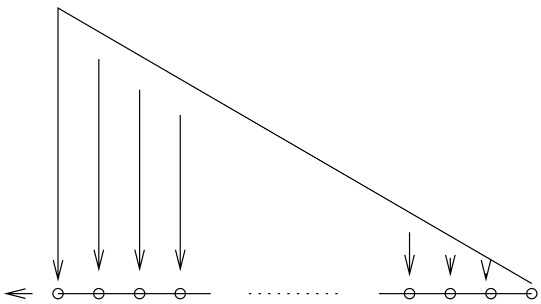

Figure PS1.3: Another solution (slightly better than $2\sqrt{N}$) for problem 8.

Problem 8

The main point of this problem was to decide what length array you should use. If the array is very long, say N, then there is not enough time to propagate the information from all processors to the output node. Thus, the array should have $O(\sqrt{N})$ processors in it.

First Solution:

Note that exclusive-or is both commutative and associative, so we may operate on bits without worrying about order. For each of the first $\sqrt{N}$ steps, each processor in a $\sqrt{N}$-cell linear array receives a bit of input and updates its current value by taking the exclusive-or of this bit with its previous current value, which is initially set to 0. After $\sqrt{N}$ steps, each of $\sqrt{N}$ processors contains a single bit, the exclusive-or of $\sqrt{N}$ bits. These must be combined into one exclusive-or bit in the next $\sqrt{N}$ steps by starting with the rightmost bit and passing the cumulative exclusive-or from right to left applying exclusive-or with the current position at each step. All this takes $2\sqrt{N}$ steps.

Second Solution:

Use an array of $\sqrt{2N}$ processors. Input $\sqrt{2N}$ inputs to the left most node (the output node). Input $\sqrt{2N} - 1$ inputs to the next processor , $\sqrt{2N} - 2$ to the next, and so on...

As in the first solution each processor takes the XOR of its inputs. When a processor doesn't get any more inputs it passes its value to the left. The cumulative XOR passes from right to left and is output from the leftmost processor. All this takes $\sqrt{2N}$ time (slightly better than $2\sqrt{N}$).

See figure PS1.3.

Problem 9

We know that for a constant degree network of size N, the time for parallel prefix is just the diameter of the network. We can embed in the $N \times N$ grid a tree with N^2 leaves so that each processor simulates both a leaf node, and the node immediately above the leaf. (Imagine a spanning tree of the grid with "hairs" hanging off each node) Thus we can immediately say that since the $\sqrt{N} \times \sqrt{N}$ array (with the "hairs" hanging off') has diameter $O(\sqrt{N})$ (and constant degree), we can compute the prefix problem on such an array in $O(\sqrt{N})$ steps.

However, if each node has $N^{1/3}$ memory then we can do better. Since each node has $\Theta(N^{1/3})$ memory, each leaf in a spanning tree of the $N^{1/3} \times N^{1/3}$ grid can be responsible for $\Theta(N^{1/3})$ of the N values (the same trick as in problem 7!). We use the same trick of embedding a spanning tree into the grid so that each node simulates one leaf and the node above the leaf. Since the diameter of the $N^{1/3} \times N^{1/3}$ grid is $O(N^{1/3})$, parallel prefix takes $O(N^{1/3})$ time.

Problem 10

Each x_i may be computed locally when all the m_i and s_i are known, so encoding is an $O(1)$ time operation. Note that for decoding, $m_i = x_i - m_{i-1} - s_i = (x_i + s_i) + m_{i-1} \pmod 2$, and since mod 2 addition is associative, we may use parallel prefix. Either note that $m_j = \sum_{i=1}^{j} x_i + s_i \pmod 2$ allowing inputs $x_i + s_i$ and associative operator mod 2 addition, or note that

$$\begin{pmatrix} 1 & x_j + s_j \\ 0 & 1 \end{pmatrix} \begin{pmatrix} m_{j-1} \\ 1 \end{pmatrix} = \begin{pmatrix} m_j \\ 1 \end{pmatrix},$$

so that

$$\begin{pmatrix} 1 & x_j + s_j \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 1 & x_1 + s_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} m_0 \\ 1 \end{pmatrix} = \begin{pmatrix} m_j \\ 1 \end{pmatrix},$$

with matrix multiplication as the associative operator. Either takes $O(\log N)$ time.

1 Problems For Extra Credit

Problem 1

To begin with we assume that $u = u_{N-1} \ldots u_1 u_0$ and $v = v_{N-1} \ldots v_1 v_0$ are twos-complement N-bit values initially stored in the array with the least-significant bits in p_0 on the right. We think of each processor as having a top and a bottom so the bits of u are stored in the top and the bits of v are stored in the bottom.

The basic operations are as follows:

1. If u and v are both even then $\gcd(u,v) = 2\gcd(u/2, v/2)$.

2. If u is even and v is odd then $\gcd(u,v) = \gcd(u/2, v)$. Likewise if u is odd and v is even then $\gcd(u,v) = \gcd(u, v/2)$.

3. If u and v are both odd then $\gcd(u,v) = \gcd(u, (u \pm v)/4) = \gcd((u \pm v)/4, v)$ with $u + v$ or $u - v$ chosen to make the result divisible by 4.

By looking at the least-significant bits the rightmost processor can determine which of these 3 cases applies. We think of this rightmost processor as the Captain since it makes all the decisions.

If the Captain sees that both of its bits are zero then case 1 applies. The Captain will store a zero in its processor and then relinquish control to the next processor to its left making that processor the new Captain. (Storing the zero records the factor of two and designating the next processor to the left as Captain divides u and v by two.)

If the Captain sees that one of its bits is zero and the other is one then case 2 applies. The Captain will send a signal to the left telling each processor to shift one of its bits (top or bottom according to which one was zero in the Captain) over to the right. The leftmost processor will have to hold on to its bit as well as passing a copy to the right since this is an arithmetic shift.

If the Captain sees that both of its bits are one then case 3 applies. By looking to its left neighbor the Captain can further determine whether to form $(u+v)/4$ or $(u-v)/4$. Accordingly the Captain will send a signal to the left telling each processor to replace either its top bit or its bottom bit (we'll get to that decision momentarily) with the sum or difference and (arithmetically) shift the result two positions to the right.

Notice that the addition or subtraction (as well as the shifting) is being done in ripple fashion from right to left and therefore takes N steps. Fortunately there is no need to wait for one of these operations to complete as the Captain can go on to the next step as soon as the current one has been initiated.

As alluded to before, in the case of both u and v odd there is still the problem of determining whether to replace $\gcd(u,v)$ with $\gcd(u,(u\pm v)/4)$ or $\gcd((u\pm v)/4,v)$. This is truly a problem since making the wrong decisions can lead to an infinite loop, and the obvious solution of always replacing the one with larger absolute value is difficult to implement since it is hard to keep the Captain apprised of which one has the larger absolute value.

Here is one possible solution. Let i_u be a lower bound on the number of leading (most-significant) zeros in the N-bit representation of the absolute value of u; that is if we write $|u|$ as an N-bit number then there will be at least i_u leading zeros. Likewise define i_v. We will keep track of $i_u - i_v$ in our array as a unary number using tokens. If we place tokens in the top positions of the k rightmost processors then $i_u - i_v = k$, while if we place tokens in the bottom positions of the k rightmost processors then $i_u - i_v = -k$. Since $0 \le i_u, i_v \le N$, $-N \le i_u - i_v \le N$ so this representation scheme is valid.

Notice that the Captain can immediately see if $i_u - i_v$ is positive or negative and thereby determine which of i_u or i_v is greater. Notice also that with this unary representation, the Captain can increment or decrement $i_u - i_v$ by passing an appropriate message to the left. Thus in the case of both u and v odd we can implement the following strategy: if $i_u \ge i_v$ then we replace $\gcd(u,v)$ with $\gcd(u,(u\pm v)/4)$, otherwise ($i_u < i_v$) we replace $\gcd(u,v)$ with $\gcd((u\pm v)/4,v)$. In other words we replace the value with fewer (known) leading zeros in the N-bit representation of its absolute value.

Now consider the effects that our three primitive operations will have on i_u, i_v, and therefore on $i_u - i_v$. (We start with $i_u = i_v = i_u - i_v = 0$.) In case 1, both u and v get divided by two so i_u and i_v both increment by one and the difference is unchanged. In case 2, one of u or v gets divided by two so one of i_u or i_v gets incremented and the difference is either incremented or decremented. Finally in case 3, if $i_u \geq i_v$ then $i_{(u \pm v)/4} > i_v$, so we can increment i_v and decrement $i_u - i_v$; likewise if $i_u < i_v$ then we can increment i_u and decrement the difference.

Notice that if $i_u + i_v \geq 2N$ then at least one of i_u or i_v is at least N, and therefore at least one of u or v is zero. Thus when this occurs we are done. We have just seen that each of our primitive operations will increment either i_u or i_v (or both) and that the Captain can begin a new primitive operation at each step of the algorithm, therefore the algorithm runs in $O(N)$ steps. I have not given the details of maintaining $i_u - i_v$ or of detecting termination, but they are not hard to work out.