Theory of Parallel Hardware February 17, 2004
Massachusetts Institute of Technology 6.896
Charles Leiserson, Michael Bender, Bradley Kuszmaul Handout 4

Solution Set 1

Due: In class on Wednesday, February 11.
Starred problems are optional.

Problem 1-1. Design an efficient algorithm to sort N numbers in $\Theta(N)$ time using $\Theta(\lg N)$ processors, where each processor can hold an unlimited amount of data. You may use any fixed-connection network you wish. (*Hint:* Refresh your memory on serial sorting algorithms.)

Solution: We simply simulate merge-sort on a linear array of $\lg N - 1$ processors. Each processor expects two sorted lists of equal length. The first processor would expect sorted lists of length of 1, the second processor would expect length 2, and the ith processor, a length of 2^i. We can actually implement the algorithm such that the processors do not need to know the length of the string to expect, but I will come back to this detail later.

Suppose that the i-th processor has two sorted lists of length 2^i. Then it can output a sorted list of length 2^{i+1}. It just needs to compare the heads of both lists and pop the lesser of the two values to output.[1] The processor continues this process $2(2^i)$ times to produce a sorted output list of length 2^{i+1}. This process is just what merge sort does, so I will not argue the correctness here.

So far, we have assumed that the processor already has two sorted lists in memory that it knows how to access. We still need to describe how the processor can process input to form these lists. If each processor can be different, then the ith processor can expect lists of length 2^i, and breaking the input up is simple. If we want processors to be identical, there are many different techniques we can use, but I will only talk about one here. Let us suppose that the input stream is flagged with some sort of list delimiter marker. Each delimiter marks a separate list. Each of these lists input to a single processor are of the same length. Each processor keeps two queues. It also keeps one other bit to know which queue is the current queue. Then, when the processor receives a number, it pushes it onto the current queue. When the processor receives a list delimiter token, the token is first pushed onto the current queue, then the current queue is toggled—future numbers are pushed onto the other queue. Once the processor has received two list delimiter tokens, it begins to produce output.[2] Now, the processor just compares the heads of both queues as described above. The list delimiter token is considered to have infinite value. Once both queues are headed by a list delimiter, both tokens are popped, but a single token is output. Thus, the processor produced a sorted list of twice the length of its input lists.

The start of the computation is also a detail to address. If we can modify the input to contain list delimiters between each number, then we are done. If not, then we have two options. The first processor can be a special case. We can make the first processor know that its input lists are of length 1 and insert list delimiters as appropriate to the output. If we want all processors to be identical, we can alternatively manage this special case by making the processors expect a special token at the beginning of the input stream as well—when they do not receive the token, they perform the special case. The first processor would be the only one that does not receive the start token.

Correctness of this algorithm should be quite obvious as we are simply performing a merge sort. We only need $\lg N - 1$ processors because the $(\lg N - 1)$th processor produces a sorted list of length $2^{(\lg N - 1)+1} = N$.

We now look at running time. The ith processor receives N numeric inputs and $N/2^{i-1}$ delimiters inputs. This number is at most $2N$ inputs. The processor produces no output for the first $2^i + 2$ inputs as it waits for two complete lists. After this time, it produces one output for every numeric input it receives. Thus, this initial lag is the only component we really need to consider. We just take the summation of all these lags over all the processors

$$T = \sum_{i=1}^{\lg N} (2^i + 2) \tag{1}$$

[1] An empty list is considered to have infinite value, so once all the values from one list are popped to output, the remainder of the other list is output.

[2] The processor can produce output immediately after the first number following the first delimiter token, but the running time is asymptotically the same, and I find this way easier to talk about.

$$= 2\lg N + \sum_{i=1}^{\lg N} 2^i \quad (2)$$

$$= 2\lg N + O(N) \quad (3)$$

$$= O(N)\ . \quad (4)$$

Then we need to add $2N$ time for all the inputs to trickle in and $\lg N$ time to go from he first processor to the last processor, but the total running time is still $O(N)$.

Problem 1-2. Argue persuasively that any P-processor fixed-connection network with at most d neighbors per processor has diameter $\Omega(\log_d P)$.

Solution: For a proof, we just look at how many processors can be reached in i hops from a given processor P_1. In 1 hop, we can only reach d other processors. In 2 hops, we can reach at most $d-1$ new processors from each of the d processors we reached in the first hop, for a total of $d(d-1)$ processors. In 3 hops, we can reach at most $d(d-1)^2$ processors. So, in i hops, we can reach at most $d(d-1)^{i-1} = O(d^i)$ processors. How many hops does it take to reach all the processors? Well, we just set $d^i \geq P$ and take the $\log_d$ of both sides to get $i \geq \log_d P$. Thus, it takes $\Omega(\log_d P)$ to reach all P processors from the processor P_1, so the diameter of the network is $\Omega(\log_d P)$.

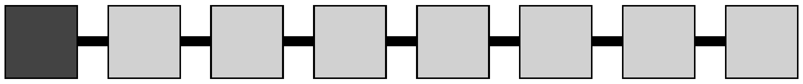

Figure 1

Problem 1-3. * A firing squad has been formed as a one-dimensional array of n soldiers, as shown in Figure 1. Each soldier is a finite automaton whose next state is dependent only on its current state and the current states of its two neighbors. There is no global communication, but the state transitions of the soldiers are synchronized by a global clock. The goal is for all the soldiers to shoot their guns at the same time.

All the soldiers begin in the same QUIESCENT state, and all soldiers are identical, except for first and last soldiers. The last soldier knows he or she is the last soldier. The first soldier is the general, who, at the start of the computation, enters the FIRE-WHEN-READY state. The goal is for all soldiers to enter the SHOOT state *on exactly the same clock tick.*

Give a design for the soldier automaton so that all soldiers fire at the same time. Your algorithm should run in $\Theta(N)$ time.

Solution: Every soldier (including the general) has the following states:

QUIESCENT: wait for signal
FIRE-WHEN-READY: enter from QUIESCENT state when first hearing the signal
BOTH: enter from FIRE-WHEN-READY state when hearing the signal from both neighbors at the same time; when

entering this state, start sending signal to both neighbors in the next cycle.
LEFT: enter from FIRE-WHEN-READY state when hearing the signal from the left neighbor first, then from the right neighbor in the next cycle; when entering this state, start sending signal to the left neighbor.
RIGHT: enter from FIRE-WHEN-READY state when hearing the signal from the right neighbor first, then from the left neighbor in the next cycle; when entering this state, start sending signal to the right neighbor.
FIRE: enter from BOTH, LEFT, or RIGHT state when both of neighbors are in BOTH, LEFT, or RIGHT states as well; when entering this state, fire in the next cycle.

In addition, all soldiers should be able to propagate signals from neighbors, and generate and send the "get ready" signal in two different speeds, X and 3X (where 3X is 3 times faster than X).

The algorithm works as follow:
1) The general will enter the FIRE-WHEN-READY state, fire signal in speed 3X in the first cycle, and fire another signal in speed X in the second cycle.
2) All soldiers receiving this signal will get into the FIRE-WHEN-READY state, and propagate the signal rightwards.
3) As the last soldier receives the first signal in speed 3X, it reflects the signal leftwards in speed 3X.
4) Modulo some implementation details, eventually both signals (in speeds X and 3X) will meet at the middle soldier, which then enter the BOTH state. (Or, if there are even number of soldiers, eventually both signals will meet at the two soldiers in the middle, which then the left one will enter the LEFT state, and the right one will enter the RIGHT state.)
5) The soldier(s) in the BOTH / LEFT / RIGHT state will then repeat what the general did, sending signals in speed 3X and X to both neighbors / left neighbor / right neighbor.
6) By repeating steps 4 and 5, the soldiers are recursively divided into 2 sub groups among themselves, with the middle soldier(s) in the group entering BOTH / LEFT / RIGHT state.
7) Since its always the middle soldier(s) first enter the BOTH, LEFT, or RIGHT state, we will never have a cluster of 3 soldiers being in those states unless every other soldiers are also in those states. Therefore, every soldier enters BOTH, LEFT, or RIGHT state eventually, causing every soldier to enter the FIRE state simultaneously after 2 cycles.
8) The soldiers should fire simultaneously in the following cycle.

Since it takes $\Theta(N)$ time to find the middle soldier among N soldiers, this algorithm takes total time of: $\Theta(N) + \Theta(N/2) + \Theta(N/4) + ... + 1 = \Theta(N)$

Therefore, we can conclude that, this algorithm runs in $\Theta(N)$ time.