Recitation 10: Distributed and Sub-Linear Algorithms

1 Distributed Algorithms

Distributed algorithms are algorithms meant to work in distributed networks or on multiprocessor computers. With the right algorithms, distributed systems can solve computational problems faster, up to a maximum speedup linearly proportional with the number of processors (of course, not all distributed algorithms reach this factor of speedup).

1.1 Models of Processing

There are a variety of ways to construct models for distributed algorithms. In lecture, we saw a synchronous network model, with many computers in an unknown network topology that could perform unbounded computations within themselves, but whose goal was carrying out tasks through communication.

Today, we’re going to be looking at a single machine with multiple processors but shared memory. This model is called a Parallel Random Access Machine. We’ll again use the concept of “rounds” to describe execution. We assume an unlimited number of processors, and also that all processors already have assigned roles, e.g. id numbers that allow them to know which pieces of data to read and write when carrying out a particular algorithm.

As with the network model, we assume some way of making timing guarantees. Finally, we assume infinite memory, to avoid discussion of the time for memory allocation.

Even within this model, there are a few paradigms:

- Exclusive-Read, Exclusive-Write: You’re never allowed to ask two processors to the same piece of data within the same round.

- Concurrent-Read, Exclusive-Write: You are not allowed to ask 2 processors to write to the same location, but any number of processors may read the same piece of data. It’s assumed that you will not be
attempting to read a piece of data that is being written this round, lest
you run into undefined behavior.

• Concurrent-Read, Concurrent-Write: Any number of processors can
read or write from the same piece of data, so algorithms must take
special care of under what conditions they concurrently write (e.g. all
writing the same piece of data).

The CRCW design is what we will focus on here, as it is clearly the most
powerful, given that you can trivially add exclusivity constraints within the
algorithms that use this design (if you want to).

1.2 CRCW-PRAM

We will examine how a concurrent-read, concurrent-write parallel random
access machine works in practice. Concurrent read does not actually cause
discrepancies as long as you are careful to not read data that is also being
written to in the current round. We’re assuming that the model takes on an
“arbitrary” assignment for its concurrent write protocol, which means that
one processor’s write succeeds and the rest fail. However, our algorithms will
have all processors that are writing to a specific piece of data write the same
thing, rendering this point moot.

1.2.1 AND/OR

AND and OR are the simplest examples. The input is assumed to be a list
of boolean variables. The rounds to execute OR are as follows:

1. All processors write a 0 to the same bit in storage, x.

2. All processors read the single variable they were assigned from the input
list. Processors beyond the length of the input list skip step 3.

3. Each processor writes a 1 to x iff that variable’s value was 1 (or nonzero
   if the input list contains arbitrary numbers instead of boolean vari-
   ables). Otherwise, the processor does nothing.

4. All processors read x, which is the result of OR.
AND is constructed similarly, but with \( x \) initialized to 1 and set to 0 if a variable’s value is 0.

It’s easy to see that this satisfies our requirement that the same value is being written by any currently-writing processor. Thus, this is a constant-time distributed algorithm for calculating the ANDs and ORs of arbitrarily many elements.

### 1.2.2 SUM

For SUM, we have an input list of numbers. To make the concept of rounds less confusing, we will assume fixed-bit integer addition, but as long as timing is made synchronous somehow, it doesn’t matter how long a round actually is.

It turns out that this algorithm works on an EREW machine, but as stated earlier, the CRCW machine can emulate this functionality trivially. Rounds are as follows:

1. Processors are uniquely assigned 2 items in the input array by some known formula according to their IDs. For example, assume processor IDs range from 0 to infinity. An easy distribution method would be elements \( 2 \times ID, 2 \times ID + 1 \). If there are an odd number of elements, one processor is assigned the leftover element. The processors read their assigned items in this round.

2. Processors that were assigned items compute their sum and write them to element \( ID \) of a new array. If a leftover element existed, the sum is the single element.

These rounds are done iteratively on the newly-written data for \( \lg(n) \) passes, where \( n \) is the size of the original input list, resulting in \( 2 \lg(n) \) rounds. A final round can then be for all processors to read the single resulting element; they know what round to do this in because it is precisely deterministic in the size of the input list. Thus, this ends up being \( O(\log(n)) \) summation. It’s pretty similar to how the circuit you developed in problem set 4 would run, since all voltage sources for that are provided simultaneously.

### 1.2.3 MERGE

The problem here is to merge two already-sorted lists into a single sorted list, as in the time-intensive step of MERGESORT. As stated, your input is two
already-sorted lists. We will assume for simplicity that the lists have equal sizes for the first algorithm, and also assume that all elements are distinct (or else collisions become impossible to avoid without performance costs).

There’s a pretty easy way to do it relatively quickly:

1. Each processor is assigned an element from one of the two lists. It knows which list it’s part of, and reads that element in the first round.

2. \(O(\log(n))\) rounds are spent by each processor doing a binary search into the other list. Once the assigned element’s place is found in the other list, the element’s location in the final sorted list is known to the processor.

3. Each processor writes its element’s value to the appropriate index in the output list.

This gets us an \(O(\log(n))\) merge operation, which speeds up a merge sort dramatically. But can we do better? For this algorithm, we will consider one list as \(A\) with elements \(a_i\) and the other as \(B\) with elements \(b_j\). Consider the following divide-and-conquer approach, in which simplicity miraculously dictates that the lists no longer need equivalent size:

1. Divide \(A\) into \(\sqrt{k}\) regions of length \(\sqrt{k}\), \(B\) into \(\sqrt{m}\) regions of \(\sqrt{m}\). Each processor is assigned a pairwise comparison between two elements, \(a_i, b_j\) at the front of their respective regions in opposite lists. There are \(\sqrt{k} \ast \sqrt{m}\) such pairs, so fewer than \(k + m\) comparisons are made this way.

2. Make groups of the \(\sqrt{m}\) processors assigned to the same \(a_i\) to find each \(b_j < a_i < b_{j+1}\) in constant time, for example by having all processors that found \(a_i > b_j\) checking the output of the computation performed with the next lower \(b_j\). Only one processor should find the region, so it writes the region in a location corresponding to that \(a_i\).

3. With the same groupings, compare each selected \(a_i\) to every item between \(b_j\) and \(b_{j+1}\) (again, \(\sqrt{m}\) elements).

4. As in step 2, use these comparisons to now find an exact location in \(B\) for \(a_i\).
5. This method allows us to map a $\sqrt{k}$ region in $A$ perfectly to a consecutive (but arbitrarily-sized) chunk of $B$. If we call this algorithm recursively on the mappings constructed this way, we eventually reach a base case, and a merge of 2 to 4 elements is trivial.

Thus, this algorithm gets us a running time of $T(n) = T(\sqrt{n}) + O(1) = O(\log \log n)$.

Thus, with only $n$ processors, that gets us a total of $O(\log \log n)$ runtime for a merge operation, which makes a standard MERGESORT using this merge take only $O(\log n \log \log n)$ time. It turns out that with $n \log n$ processors, you can utilize this merge and reach the optimal $O(\log n)$ parallel MERGESORT. It’s also possible to do optimal MERGESORT in $O(\log n)$ time with $n$ processors, but we won’t discuss how to make either of these improvements.

## 2 Sublinear Algorithms

The gold standard of achievement for most algorithms is reaching linear time in the input size. This seems natural, as it seems difficult to make a decision without inspecting the entirety of your input. Sometimes, we can do better. There are problems for which exact deterministic sublinear algorithms work, but we will be examining problems that need randomization to be solved this efficiently. In particular, here we’re going to talk about property testing.

### 2.1 Property Testing

Property testing is a notion of approximation for decision problems. The algorithms for these are structured similarly to Monte Carlo algorithms, with some key differences: their runtimes are, of course, always sub-linear, and there is a region of uncertainty within which behavior is undefined. That is, there’s one-sided error, but the probabilistic guarantees over that error only take effect once the property of the input is sufficiently distant ($\epsilon$-far) from the property we’re testing for. We’ll go straight into an example with diameter-clustering.

#### 2.1.1 Testing of Clustering

For this problem, we want to see if a graph with an arbitrary distance metric can be $(k,b)$-clustered, that is, clustered into $k$ subsets such that no subset has
diameter greater than $b$. As mentioned, we’re generous with our probabilistic guarantees: our algorithm will have undefined behavior when the acceptable diameter for $k$ clusters is between $b$ and $2b$, but will always be correct when $(k,b)$-clusterable, and will be correct at least $2/3$ of the time when $\epsilon$-far from $(k,2b)$-clusterable, which in this case means that more than $\epsilon n$ points have to be removed in order for it to be $(k,2b)$-clusterable.

The idea we’ll be exercising is finding $k$ representative points that could be part of circles of diameter $b$. We do this by using the points as centers of new, larger circles of radius $b$; if we still find $k+1$ representative points this way, we know the graph must not be $k$-clusterable, so we output false. If we found $k$ or fewer representative points, we output true.

The algorithm is as follows:

1. Let $rep_1$ be an arbitrary point in $G$, representing the first cluster.
2. Set $i$ to 1, find-new-rep to true
3. while $i < k + 1$ and find-new-rep == true:
   (a) Select a sample of size $\ln(3k)/\epsilon$ points uniformly at random
   (b) If there exists a point $x$ s.t. $\text{distance}(x, rep_j) > b$ for every $j \leq i$, increment $i$ and set $rep_i = x$
   (c) else set find-new-rep to false (and therefore break from loop)
4. If $i \leq k$ output true, else output false

There are at most $k$ iterations of the while loop, and each one computes no more than $k \ln(3k)/\epsilon$ distances, so the running time is $O(k^2 \log(k)/\epsilon)$. Note that this may be arbitrarily small for a sufficiently large graph.

**Theorem 1.** This algorithm is a diameter-clustering tester within a bound of clusters of size $2b$.

**Proof:** First, if $G$ is $(k,b)$-clusterable, then it is impossible to have $k+1$ “cluster centers” as defined in the algorithm, as all $k+1$ points would be required to be in different clusters in the original algorithm. Thus, our algorithm never erroneously outputs false.

Next, we show that if $G$ is $\epsilon$-far from $(k,2b)$-clusterable, we output false with probability at least $2/3$. Consider one iteration of the while loop.
If $i \leq k$, there must be more than $\epsilon n$ points that are still valid “representative points” for new clusters; otherwise, we would show that by removing the points, G becomes $(k,2b)$-clusterable, which violates our assumption about $\epsilon$-farness. Finally, if we uniformly select $\ln(3k)/\epsilon$ points from G, the probability that no valid representative is chosen is less than $(1 - \epsilon)^{\ln(3k)/\epsilon} < \exp(-\epsilon(\ln(3k)/\epsilon)) = 1/3k$. Thus, the probability that this ever occurs in any iteration is at most $1/3$ (this is for $k = 1$; it gets better from there), so our probability bound is met.