CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 19, 6/7/2017. Scribed by A Santucci.

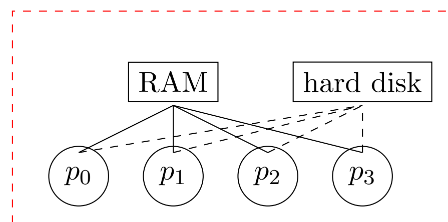
1 PRAM Model

1.1 Sequential model of computation

Random Access Memory (RAM) model is the typical sequential model. We are given a processor p_1, and it's attached to a memory module m_1. The processor p_1 can read and write to memory *in constant time*. This is the model we are used when using `loops` in C or Java. In RAM Models, we typically count the number of operations and assert that the time taken for the algorithm is proportional to the number of operations required. I.e., time taken $\propto$ number of operations.

1.2 Parallel model of computation

In a Parallel RAM (PRAM) Model, we always have multiple processors.



But how these processors interact with the memory module(s) may have different variants. The different ways a PRAM model can be set up is the subset of the following combinations: $\{\text{Exclusive}, \text{Concurrent}\} \times \{\text{Read}, \text{Write}\}$. Of these combinations, the *most popular model* is the concurrent read and exclusive write model.

1.3 Constructing a DAG from an algorithm

It is natural to represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG). Specifically, each fundamental unit of computation is represented by a node. We draw a directed arc from node u to node v if computation u is required as an *input* to computation v. We trivially note that our resulting graph surely ought to be connected (otherwise the isolated node[s] represent useless computations), and by definition it's acyclic (otherwise we have an infinite loop). Hence its a tree. The root of the tree represents the output of our algorithm, and its children are the dependencies.

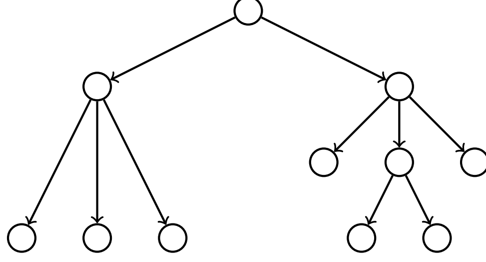


Figure 1: Example Directed Acyclic Graph (DAG)

1.4 Work Depth Model

In a PRAM, we have to wait for the slowest processor to finish all of its computation before we can declare the entire computation to be done. This is known as the *depth* of an algorithm. Let

$$\begin{aligned} T_1 &= \text{amount of (wall-clock) time algorithm takes on one processor} = \text{ \# nodes in DAG, and} \\ T_\infty &= \text{amount of (wall-clock) time algorithm takes on } \infty \text{ processors} = \text{ \# levels in DAG.} \end{aligned}$$

Theorem 1.1 (Brent's Theorem) *We claim that* $\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$.

Proof: On level i of our DAG, there are m_i operations. Hence by that definition, since T_1 is the total work of our algorithm, $T_1 = \sum_{i=1}^{n} m_i$ where we denote $T_\infty = n$. For each level i of the DAG, the time taken by p processors is given as $T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1$. This equality follows from the fact that there are m_i constant-time operations to be performed at m_i, and once all lower levels have been completed, these operations share no inter-dependencies. Thus, we may distribute or assign operations uniformly to our processors. The ceiling follows from the fact that if the number of processors not divisible by p, we require exactly one wall-clock cycle where some but not all processors are used in parallel. Then,

$$T_p = \sum_{i=1}^{n} T_p^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right) = \frac{T_1}{p} + T_\infty$$

■

1.5 Types of Scaling

Let $T_{1,n}$ denote the run-time on one processor given an input of size n. Suppose we have p processors. We define the speed up of a parallel algorithm as $\texttt{SpeedUp}(p, n) = \frac{T_{1,n}}{T_{p,n}}$.

Definition 1.1 *If* $\texttt{SpeedUp}(p, n) = \Theta(p)$, *we say that the algorithm is* <u>*strongly scalable*</u>.

Definition 1.2 *We sometimes are interested to consider the case where for each processor we add, we add more data as well. If* $\texttt{SpeedUp(p,np)} = \frac{T_{1,n}}{T_{p,np}} = \Omega(1)$, *then our algorithm is* <u>*weakly scalable*</u>.

Definition 1.3 *The last notion of scalability is the case when our algorithm is embarrassingly parallel. This is true if our DAG has nodes 0-depth, e.g.* ○ ○ ○ ○

2 Scheduling

Problem Statement We assume that the processors are identical (i.e. each job takes the same amount of time to run on any of the machines). More formally, we are given p processors and an unordered set of n jobs with processing times $J_1, \ldots, J_n \in \mathbb{R}$. Say that the final schedule for processor i is defined by a set of indices of jobs assigned to processor i. We call this set S_i. The load for processor i is therefore, $L_i = \sum_{k \in S_i} J_k$. The goal is to minimize the *makespan* defined as $L_{max} = \max_{i \in \{1,\ldots,p\}} L_i$.

Algorithm We present the following simple algorithm.

```
1 for each job that comes in (streaming) do
2 |   Assign job to lowest burdened machine
3 end
```

Algorithm 1: Simple scheduler

Analysis Greedy algorithm has an approximation ratio of 2.
Proof: We know that the optimal makespan must be at least the sum of the processing times for the jobs divided amongst the p processors, i.e. OPT $\geq \frac{1}{p}\sum_{i=1}^{n} J_i$. A second lower bound is that OPT is at least as large as the time of the longest job, OPT $\geq \max_i J_i$.

Now consider running the greedy algorithm and identifying the processor responsible for the makespan of the greedy algorithm (i.e. k = $\text{argmax}_i L_i$). Let J_t be the load of the last job placed on this processor. Before the last job was placed on this processor, the load of this processor was thus $L_{\max}$ - J_t. By the definition of the greedy algorithm, this processor must have also had the least load before the last job was assigned. Therefore, all other processors *at this time* must have had load at least $L_{\max} - J_t$, i.e. $L_{\max} - J_t \leq L'_i$ for all i. Hence, summing this inequality over all i,

$$p(L_{\max} - J_t) \leq \sum_{i=1}^{p} L'_i \leq \sum_{i=1}^{p} L_i = \sum_{i=1}^{n} J_i \tag{1}$$

In the second inequality, we assert that although J_t the last job placed on the bottleneck processor, there may still be other jobs yet to be assigned, hence we have that the sum of total work placed on each machine cannot decrease after assigning all jobs. The last equality comes from the fact that the sum of the loads must be equal to the sum of the processing times for the jobs.

Rearranging the terms in equation 1 let's us express this as: $L_{\max} \leq \frac{1}{p}\sum_{i=1}^{n} J_i + J_t$. Now, note that our greedy algorithm actually has makespan exactly equal to $L_{\max}$, by definition. Using our lower bounds on OPT, along with the fact that $J_t \leq \max_i J_i$, we get that our greedy approximation algorithm has makespan not more than 2 times more than the optimal. ■

3 All Prefix Sum

Problem Statement Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size $n+1$ elements long, and its first entry is *always* zero.

Algorithm The general idea is that we first take the sums of adjacent pairs of A. So the size of A' is exactly half the size of A. If the size of A not a power of 2, we simply pad it with zeros.

Input : An array A

Output: An array containing a running sum of elements in input A

1 **if** *size of A is 1* **then return** *only element of A*;
2 Let A' be the sum of adjacent pairs // $O(n)$ work
3 Compute $R' = \text{AllPrefixSum}(A')$ // Recursive call of size $n/2$
4 Fill in missing entries of R' using another $\frac{n}{2}$ processors // $O(n)$ work, constant depth

Algorithm 2: Prefix Sum

Pairing entries gets a running sum for even parity indices. We define $r_k = \sum_{i=1}^{k} a_i$, i.e. a running sum through index k. To fill in running sum for odd indices, we adhere to the following formula: $r_i = r_{i-1} + a_i$ for odd indices i.

Analysis Given by

$$W(n) = W(n/2) + O(n) \implies W(n) = O(n)$$
$$D(n) = D(n/2) + O(1) \implies D(n) = O(\log n)$$

4 MergeSort

All the interesting work in this algorithm below is actually done in the `parallel-merge` subroutine.

Input : Array A with n elements

Output: Sorted A

1 **if** $|A|$ *is 1* **then return** A;
2 **else** // (In parallel, do)
3 $L \leftarrow$ `merge-sort(A[0:n/2])` // Recursive call of size $n/2$
4 $R \leftarrow$ `merge-sort(A[n/2:n])` // Recursive call of size $n/2$
5 **return** *parallel-merge(L,R)* // $O(n \log n)$ work, $O(\log n)$ depth
6 ;

Algorithm 3: `merge-sort`

When we use `parallel-merge` in our `merge-sort` algorithm, we realize the following work and depth, by the master theorem:

```
Input   : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order
1 for each a ∈ A do                                              // In parallel...
2     Do a binary search to find where a would be added into B,    // O(log n) work
3     The final rank of a given by rank_M(a) = rank_A(a) + rank_B(a).;
```

Algorithm 4: `parallel-merge` subroutine

$$W(n) = 2W(n/2) + O(n \log n) \implies W(n) = O(n \log^2 n),$$
$$D(n) = D(n/2) + \log n \implies D(n) = O(\log^2 n).$$

By Brent's Theorem, we get $T_p \leq O(n \log^2 n)/p + O(\log^2 n)$, so for large p we significantly outperform the naive implementation!

5 Parallel Quick-Select and Order Statistics

Given a sequence a and an integer k where $0 \leq k < |a|$, find the kth smallest element in the sequence, i.e. the kth order statistic. Naively, we may first sort the data, then select the kth element; this requires $O(n \log n)$ work. In lecture 4, we learn to get away with linear work *in expectation* and $O(\log^2 n)$ depth.

```
Input   : Array A with n entries, integer k ≤ n
Output: Value of the kth largest element in A
1 p ← a value selected uniformly at random from A
2 L(, R) ← elements in A which are less (,larger) than pivot p // Θ(n) work, O(log n) depth
3 if |L| is k − 1 then return p;
4 if |L| > k then return Select(L, k);
5 else               return Select(R, k-|L|-1);
```

Algorithm 5: Select(A, k)

The intuition for analysis is to define "phases" of the algorithm when the input shrinks by 3/4; we then realize a geometric series, whereby master theorem instructs us that since we perform linear work per iteration that the total work done is still only linear. We do have to be smart about how we construct L or R, in order to be depth-efficient.

Input : Array A with n entries, pivot element p
Output: Elements in A that are less than (or greater than) pivot p

1 Construct indicator list $B_L[0, ..., n-1]$, where $b_i = \mathbf{1}\{a_i < p\}$ `// ` $O(n)$ `work, ` $O(1)$ `depth`
2 Compute **PrefixSum** on B_L `// ` $O(\log n)$ `depth`
3 Create the array L of size $\textbf{PrefixSum}(B_L)[n-1]$
4 **for** $i = 1, 2, \ldots, n$ **do** $L\big[\textbf{PrefixSum}(B_L[i])\big] \leftarrow a[i]$;

Algorithm 6: Constructing L (or R)

6 Parallel Quick Sort

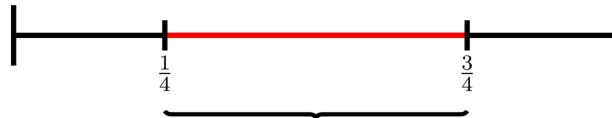
Parallel quick sort is very similar to parallel selection. We randomly pick a pivot and sort the input list into two sublists based on their order relationship with the pivot. However, instead of discarding one sublist, we continue this process recursively until the entire list has been sorted.

Input: An array A
Output: Sorted A

1 $p \leftarrow$ element of A chosen uniformly at random `// ` $\Theta(\log n)$ `work, ` $O(1)$ `depth`
2 $L \leftarrow [a | a \in A \text{ s.t. } a < p]$ `// Implicitly: ` $B_L \leftarrow \mathbb{1}\{a_i < p\}_{i=1}^n$`, prefixSum(`$B_l$`),`
3 $R \leftarrow [a | a \in A \text{ s.t. } a > p]$ `// which requires ` $\Theta(n)$ `work and ` $O(\log n)$ `depth.`
4 **return** $[\mathsf{QuickSort}(L), p, \mathsf{QuickSort}(R)]$

Algorithm 7: QuickSort

In a similar fashion to our analysis for `QuickSelect`, we may consider the number of *recursive calls* made to our algorithm when our input array is of size $\left(\frac{3}{4}\right)^{k+1} n < |A| < \left(\frac{3}{4}\right)^k n$, and denote this quantity by X_k. We saw that if we select a pivot in the right way, we can reduce input size by 3/4.



If we select any of these elements in red as a pivot, where we visualize the elements being in sorted order, *then* both L and R will have size $\leq \frac{3}{4}n$.

Figure 2: Ideal Pivot Selection

Depth analysis We saw that $\mathbb{E}[X_k] = 2$, since it's a geometric random variable (i.e., we continually make calls to `QuickSort` until we successfully reduce our input size by at least 3/4. In expectation, this takes two trials. Hence in expectation we require $2c \log_{4/3} n$ recursive calls to be made before we reach a base case. At each level in our DAG, our algorithm requires $O(\log n)$ depth since the bottleneck is in constructing L and R and specifically in the call to Prefix Sum. Hence in

expectation our total depth given by the expected number of levels times the expected depth per level: $\mathbb{E}[D(n)] = O(\log^2 n)$.

Work analysis There are in expectation $O(\log n)$ levels in our DAG, and in each we perform $O(n)$ work, whence the algorithm requires $O(n \log n)$ work.

7 Matrix Multiplication on PRAM

We now consider parallel matrix-multiplies on a PRAM. The idea behind Strassen's algorithm is in the formulation of matrix multiplication as a recursive problem. We first cover a variant of the naive algorithm, formulated in terms of block matrices, and then parallelize it. Assume $A, B \in \mathbb{R}^{n \times n}$ and $C = AB$, where n is a power of two. We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where block matrices A_{ij} are of size $n/2 \times n/2$ (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}, \quad C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Further, realize that the four block entries of C may be computed independently from one another, hence we may come up with the following recurrence for *work:*

$$W(n) = 8W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 8}) = O(n^3)$$

since there are 8 matrix multiplies, each of size $n/2$, as well as 4 matrix additions of size $n \times n$.

7.1 Strassen's algorithm

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{\log_2 7})$. How do we do this? We use a factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M_k may be calculated simply from products and sums of sub-blocks of A and B, where crucially each factor requires only *one* matrix-multiply; since each factor expands by the distributive property, they contain additional information. Whence the work is given by

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

8 Graph algorithms

8.1 MST

Kruskal's algorithm is instructive in that it uses the Cut Property which we proved last lecture. Our parallel algorithm, which is really Boruvka's algorithm, will also use this property. A single

node is a subset of the nodes, i.e. each node can be considered individually to define a cut. And so if you look at a single node and its incident edges, the smallest one must be in the MST by applying the cut property. Realize that if we determined the smallest weight edge incident to *each* node in our graph, we *must* find at least $n/2$ unique edges belonging to the MST.

Input : Graph G, represented by adjacency list
1 **for** *each node, in parallel* **do**
2 Compute smallest weight incident edge // $O(m)$ `work,` $O(\log n)$ `depth`
3 **end**
4 Contract edges // $O(m \log n)$ `work`
5 Merge Adjacency Lists // $O(n)$ `work and` $O(1)$ `depth`
6 Recurse

Algorithm 8: Boruvka's Algorithm

For detailed analysis, see lecture 6. To find smallest weight incident edges, we scan through $\sum_{v \in V} \deg(v) = 2m$ edges in total. The depth given by $O(\sum_{v \in V} \log \deg(v))$ since we use parallel quick-select to find the smallest weight edge in each nodes incident list.

9 Optimization algorithms

Many learning algorithms can be formalized with the following unconstrained optimization problem:

$$\min_{w} \quad F(w) = \sum_{i=1}^{n} F_i(w, x_i, y_i)$$

where i indexes the data, $x_i \in \mathbb{R}^d$ is an input vector, $y_i \in \mathbb{R}$ is a label and w is the parameter we wish to optimize over. The general idea of gradient descent is based on initializing w at random and performing the sequential updates: $w_{k+1} \leftarrow w_k - \alpha \nabla f(w_k)$, where if α small enough, then F is guaranteed to decrease.[1] The objective function F depends on the problem at hand. E.g.

- squared error loss function, yields least squares $F_i(w, x_i, y_i) = ||x_i^T w - y_i||_2^2$ (strongly convex)
- a logistic loss leads to logistic regression (convex)
- cross-entropy (e.g: for logistic regression or neural nets) (not convex)
- hinge loss $F_i(w, x_i, y_i) = \max(1 - y_i(x_i^T w))$ with $y_i \in \{-1, +1\}$ (e.g: for SVM). (convex)

9.1 Analyzing least squares gradient descent

Consider the complexity of computing the gradient of the least squares objective,

$$F(w) = \sum_{i=1}^{n} F_i(w, x_i, y_i) = \sum_{i=1}^{n} ||x_i^T w - y_i||_2^2.$$

[1] Want $\alpha < \frac{1}{L}$ where L is the Lipschitz constant for F.

Evaluating the objective function requires work $O(nd)$ and depth $O(\log n + \log d)$. Its gradient is

$$\nabla_w F = \sum_{i=1}^{n} \nabla_w F_i(w) = \sum_{i=1}^{n} 2(x_i^T w - y_i)x_i \quad \in R^d$$

Here, we see that the gradient is just a re-scaling of x_i. Computing the gradient for a single datapoint can be done in $O(\log d)$ depth (the computation being dominated by the dot product operation). The global gradient $\nabla_w F$ then requires $O(\log d) + O(\log n)$ depth.

9.2 Stochastic Gradient Descent

SGD *only* makes sense for separable objective functions $F = \sum_{i=1}^{n} F_i(w, x_i, y_i)$; it's not even a well formed question to ask whether we can do SGD on a non-separable objective. How does it work? Randomly select one training point and evaluate the gradient there. If we sample uniformly at random, we hope that our algorithm will converge (and it does). Let s_k be index uniformly sampled at time step k. Then, the SGD iteration as follows:

$$w_{k+1} \leftarrow w_k - \alpha \nabla F_{s_k}(w_k).$$

What are the issues that arise when trying to evaluate SGD iteration? Suppose we have many cores, and the current model w is on shared memory. The cores all have access to the training data as well. Why can't each core, say, do one sample and then do the update? Without communication between processors, two issues may occur

1. Model used by core might be obsolete by the time it tries to write the update, and
2. the new update may be overwritten by a stale one.

Going Hogwild! If we assume sparsity of our data, then we can go Hogwild! and write concurrently. The idea is that each processor only overwrites a small, distinct location of w each iterate.

10 Streaming algorithms

Covered in detail on midterm.

11 Communication patterns

- None: `map, filter`
- All to one: `reduce`
- One to all: `broadcast`
- All to all: `join, groupByKey, reduceByKey`

We emphasize that `reduce` and `reduceByKey` are two completely different communication patterns.

12 SQL using mapreduce

12.1 Group-by using map reduce

We first implement a `group-by`, which groups all rows that have the same key and performs an aggregation function on remaining columns. A map reduce is just that. To implement a `group-by` using map reduce, all we have to do is

1 Map: emit the group-by column as key and pertinent columns for computation as value
2 Reduce: perform desired aggregation function on non-group-by columns in the reducer

Algorithm 9: group by using map reduce

12.2 SQL (inner) join using map reduce

Suppose we want to merge tables T_1 and T_2, where table T_1 is small enough to fit in memory.

1 Place hashed T_1 on each machine via a broadcast
2 Perform an inner join of whatever data in T_2 is stored locally on each machine with its local copy of (hashed) T_1

Algorithm 10: (Hash or Broadcast) Inner Join

Each machine does a join of what it owns from T_2 with *all-of* T_1. We don't even need to communicate this result to other machines, we can just leave the merged data local. The inner join that is performed locally on each machine can be done just as we did on the midterm.

12.3 Other examples

See this link.

13 Distributed Sort

1 Each machine sorts its own data
2 Each machine samples 100 data points from its local data and sends to a driver node
3 Driver approximates integer distribution by sorting its sampled data
4 From this estimate, driver can determine cutoff thresholds for each machine
5 All thresholds are broadcast to all machines, i.e. each machine knows its relevant data
6 All-to-all communication: each machine shuffles data to correct location
7 Each machine sorts the incoming (sorted) partitions that it gets sent in linear time

Algorithm 11: Distributed sort

14 Mapreduce contract

We're in a distributed computing environment. Map-reduce is a two phase contract.

1. Map phase: data fed into a `map` algorithm (user-provided), outputting key-value pairs.
2. Reduce phase: all pairs with the same key are guaranteed to show up on a single machine.

Between the map and reduce stage, there is a hidden and implicit sort performed. Why is this necessary? Put another way, how can we implement the promise that all pairs with the same key show up on a single machine; how do we implement this shuffle phase? Map each key to an integer, sort the integers, and then the data with the same key will show up on the same machine.

What does map reduce provide? It provides fault-tolerance in the case of hardware failure. They also provide a distributed file system that backs up your data at least three times. It also abstracts away details of communication patterns.

15 Complexity measures for mapreduce

During a MapReduce algorithm, 3 main stages are involved while computing the complexity. First, all of the map tasks have to finish. Second, all of the pairs which need to be on the same machine need to be shuffled. Finally, reducers take some time to finish. As a consequence, we will provide 3 complexity time measures corresponding to these 3 stages:

- **Mappers cost time:** it is simply the time the map phase takes. All the mappers have to do their computation and split out the pairs. (this is embarrassingly parallel work, so we can use the same analysis tools that we know for a single processor computation and take the worst one.)
- **Shuffle cost time:** numbers of items the mappers output to be sorted: how many tuples have to be sorted. Note that although the keys are hashable, you can not just assume that the sort time will be $O(n)$ as in practice, a more complex analysis involving the number of machines and the architecture would have to be done.
- **Reducers cost time:** How long it takes for the slowest reducer to finish.

Any of these measures could be the bottleneck of the overall analysis of a MapReduce algorithm.

16 Practice problem: Leftmost One

Input : A 0-1 bit array A, of length n
Output: Smallest $k \in [1, n]$ such that $A[k] = 1$

Algorithm 12: Leftmost One

Design an algorithm for solving `Leftmost One` problem in $\Theta(n)$ work and $\Theta(\log n)$ depth. Provide pseudocode and an analysis.

17 Practice Problem: Maximal Independent Set

Problem 1 from the following link.

18 Practice Problem: Solving Linear Systems

Consider the system of linear equations

$$\begin{bmatrix} 1 & & & & \\ a_2 & 1 & & & \\ & a_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & a_n & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

where a_is and y_is given and we wish to solve for x. Write a short program to compute x sequentially. What is the running time? Use $O(\cdot)$ notation. Show how to compute all the x_i's in $O(\log n)$ time using parallel prefix.

Additional Thought Exercise How would you generalize your program to a linear system with non-zeroes only on the diagonal and in the first m sub-diagonals.

References

[1] Rezaul A. Chowdhury *Parallel Programming.* Stonybrook Computer Science. midterm, final.

[2] Umut A. Acar and Guy E. Blelloch *Algorithm Design: Parallel and Sequential.* Carnegie Mellon Computer Science. parallel-algorithms-book.

[3] Buddhika Chamith *Modelling SQL queries using Map Reduce.* wordpress.

[4] Jim Demmel *Applications of Parallel Computers.* course website.

[5] Umut A. Acar and Arthur Chargueraud and Mike Rainey *An Introduction to Parallel Computing in C++.* website.