These lecture notes have not undergone rigorous peer-review. Please email quanquan.liu@yale.edu if you see any errors.

1 Introduction

These notes heavily rely on Chapters 1-4 and 6 from [BDS21]. This lecture will give you a crash course on parallel algorithms. We begin with a brief history of parallel algorithm. Parallel algorithms has had a long and rich history of development starting from the early 1970s. Many of these algorithms have gone on to survive the test of time and are still fundamental primitives in many systems. Nowadays, parallel algorithms outperform the best sequential algorithms on the same machine, even ones with a smaller number of cores such as your laptop. Such concepts are evend incorporated in several intro to algorithms courses.

When considering algorithms that run well on modern machines, we must first consider models that allow for us to come up with realistic algorithms.

2 Models of Computation

The traditional model used for sequential computation is the *Random Access Model (RAM)* where we have one processor, some registers which can store words of size $O(\log n)$ bits and a main memory. The instructions in this model include instructions for moving data in between registers and between memory. The model assumes that each instruction takes $O(1)$ units of time. Hence, the cost of the model is the total number of instructions from the start of the program till the end of the program. The one major drawback of the RAM is that it does not take into account the memory hierarchy that exists in real-world systems.

To mitigate this problem, researchers have come up with a more realistic model known as the Parallel RAM (PRAM). This model was used in prior works on parallel algorithms. In this model, there are p synchronous processors accessing a shared memory. The costs are given in terms of the number of processors and the number of timesteps. While this model is a model that models real-world architecture more closely, it unfortunately does not allow for easily generating pseudocode or real code. Furthermore, the model assumes a number of fixed processors which is not realistic in today's modern architecture. In real-life, in today's systems, we often have many technologies in place for dynamic allocation.

The modern parallel model we often use is the ***work-depth*** or ***work-span*** model in which we have a shared random access memory. Furthermore, there is dynamic "forking" of new processes. ***Work*** is defined as the total number of instructions/computation performed, i.e. the sequential runtime on one processor. Then, ***depth*** is the longest chain of sequential dependencies, i.e. the parallel time or scalability of the algorithm.

We can further define these definitions in terms of a computational DAG representing any algorithm where the work is the number of nodes in the DAG and the depth is the longest chain of sequential dependencies in the DAG. A parallel algorithm is ***work-efficient*** if the work performed by the algorithm is the same asymptotically as the best-known sequential algorithm for solving the problem.

The goal of parallel algorithms is for the depth/span to not dominate the cost of the algorithm. Namely, we want $\mathrm{poly}(\log n)$ depth/span. The parallelism of the algorithm is calculated as the work/span. If the work of an algorithm is $O(n \log n)$ and the span of the algorithm is $O(\log^2 n)$, then the ***parallelism*** of the algorithm is $O(n/\log n)$ which is considered high. When the work equals the depth, then the parallelism is 1 which means that the algorithm is a single-processor algorithm. There are various different types of forking, e.g. binary and arbitrary forking, which can make a difference in the span bounds.

3 Example Primitive in the Work-Span Model

Here, we'll discuss a primitive in the work-span model called ***scan*** (prefix-sum) which is a very important primitive. Suppose that ***scan*** takes the following variables: A, the sequence of real valued numbers, f, an associative function, $\perp$, the left element, and computes values r_i where $r_i = \perp$ when $i = 0$ and $r_i = f(r_{i-1}, A_i)$ for $0 < i \leq |A|$. Here, r_i is the sum of the prefix $A[0, i]$ of A with respect to function f. The returned value is an array $[r_0, \ldots, r_{|A|-1}]$ and the value $r_{|A|}$.

The parallel implementation of this primitive is a 2-pass *scan-up* and *scan-down* divide-and-conquer procedure. The scan up procedure scans up from the array and records the values of the left subtree. See Fig. 1. The scan-up procedure performs divide-and-conquer where we start with the middle and partition the array A into a left and right array. Then, the arrays are partitioned again and so on. Values are summed up the tree created by the divide-and-conquer algorithm and each node sums the value received from its left and right subtrees and sends its value to its parent. Then, the values obtained for the left subtrees are written into an output array L in the left-to-right order of the created subproblems. See Fig. 1 for an example run of this algorithm. The result of the scan-up is the sum of all numbers in A and the partial sums of the left subtrees.

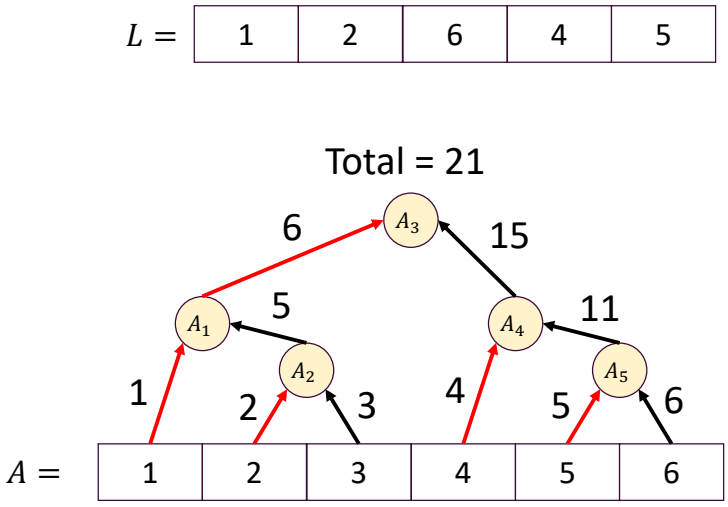

Figure 1: Example run of scan-up: A is our sequence. We perform divide-and-conquer and label the nodes of the divide-and-conquer from left to right. Then, we send the value in A and each node up to their parents. The parents sum the values they received and sent the summed value up. The values sent up from the left subtrees are stored in an array L in the order from left to right of the divide-and-conquer nodes they sent the values to.

Now, we need an additional procedure to compute all of the prefix sums. Essentially, our scan-up procedure gave us the partial sums of the left subtrees and we need to add to it the sums of the right subtrees. Using the scan-up array L, we start from the root of the tree and pass to it $\perp$. Then, each node A_i in the divide-and-conquer tree starting from the top and going to the bottom passes to its left child its value and to its right child, the sum of its value plus $L[i]$. Intuitively, we pass the value to the left since we have the values of the left subtrees and we pass the value plus $L[i]$ to the right since we need to sum together the values of the left and the right subtrees. See Fig. 2 for an example run-through of the algorithm.

There are many more important parallel primitives people use for writing parallel programs but for the sake of time, we do not prove the rest of the primitives in this class.

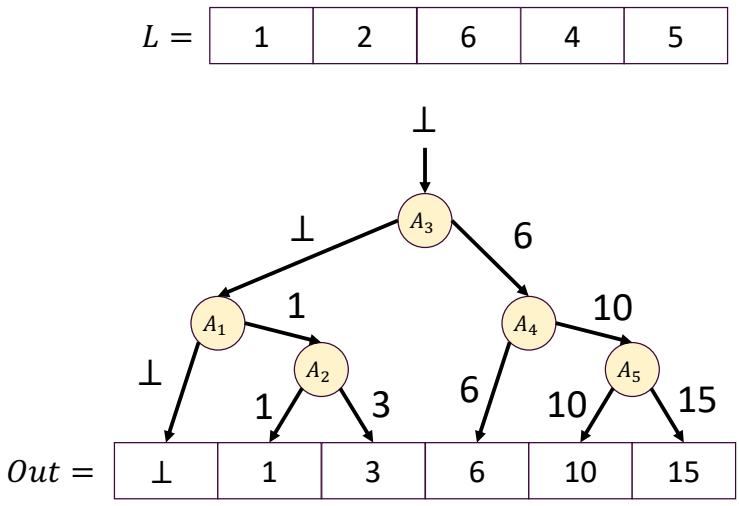

Figure 2: Example run of scan-down: L is the sequence produced by scan-up. First, $\perp$ is passed to the root node. Then, each divide-and-conquer node sends the value it received from its parent to its left child. For example, $\perp$ is sent all the way from the start down the left subtrees and to the first location of the output sequence Out. Then, each node sends the value it received from its parent plus the value stored at the index of the node in L to its right child. For example, A_3 sends $\perp$ plus $L[3] = 6$ (assuming L is 1-indexed) to its right child. Then, A_4 sends $6 + L[4] = 6 + 4 = 10$ to its right child and so on. Out then contains all of the prefix sums of A, except for the sum of all of the values of A.

4 Low-Diameter Decomposition

We now start our description of the low-diameter decomposition algorithm which is used as a subroutine for many parallel graph algorithms including connectivity, spanners, hop-sets, and low stretch spanning trees. The reason for this is because it is currently *not known* how to work-efficiently construct a BFS tree rooted at a vertex in $\text{poly}(\log n)$ depth on general graphs.

A (β, d)-decomposition partitions the vertices V of the input graph $G = (V, E)$ into clusters $V_1, \ldots, V_k$ such that the shortest path between two vertices in V_i, using only vertices in V_i, is at most d (strong diameter). Furthermore, the number of edges between clusters is at most βm, i.e. the number of edges where $u \in V_i$, $v \in V_j$ and $i \neq j$. We will discuss the sequential low-diameter decomposition algorithm today and the parallel low diameter decomposition algorithm in the next class.

The sequential low diameter decomposition algorithm of Awerbuch [Awe85] and Linial-Saks [LS93] is very simple and clever. The algorithm works as follows. We repeatedly pick an arbitrary *uncovered* vertex v and sequentially grow a ball where the radius of the ball increases by 1 each step until the number of edges incident to the boundary of the ball is at most a β-fraction of the number of edges inside the ball. Then, we stop and cover all of the vertex inside the ball and pick another arbitrary vertex if there exists an uncovered vertex.

The analysis is similarly straightforward. Every time a ball stops growing, it has at most an β-fraction of the edges contained within it on its boundary. Thus, let X be the set of edges contained within any ball; the number of edges on the boundary is at most $\beta \cdot X \leq \beta \cdot |E|$. Now, to show the diameter, whenever a ball does not stop growing, it grows the number of edges inside it by a $(1 + \varepsilon)$-factor. This means that the radius grows at most $\log_{1+\varepsilon}(m)$ times before all of the edges are inside the ball. Hence, the strong diameter is $O(\log_{1+\beta}(n))$.

Thus, this algorithm produces a $(O(\log_{1+\beta}(n)), \beta)$-decomposition.

References

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

[BDS21] Guy E Blelloch, Laxman Dhulipala, and Yihan Sun. Introduction to parallel algorithms (draft). 2021.

[LS93] Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.