Chapter 6

Parallel Algorithms

Chapter by M. Ghaffari. Last update[1]: December 15, 2020.

This chapter provides an introduction to *parallel algorithms*. Our high-level goal is to present "how to think in parallel"— roughly speaking, how to design and analyze computations that are more amenable to parallelism. Naturally, we will do this by way of going through basic problems and algorithms. To make things concrete, most of our focus will be on algorithms in the Parallel Random Access Machines (PRAM) model, which considers a number of RAM machines, all of whom have access to some shared memory. We note that this model is most interesting from a theoretical perspective and it ignores a range of practical issues, instead trying to focus on the fundamental aspects of "parallelism in computation". The PRAM model was introduced in the late 1970s and it was studied extensively, especially throughout 1980 to 1995. While there have been a number of different theoretical models introduced throughout the years for parallel computation, PRAM has remained the primary one and it has served as a convenient vehicle for developing algorithmic tools and techniques. Many (though certainly not all) of the ideas developed in the context of PRAM algorithms have proved instrumental in several other models and they are also being used in practical settings of parallel computation.

At the end of this chapter, we briefly discuss algorithms in the Modern/Massively Parallel Computation (MPC) model. This is a more recent theoretical model, which focuses on some more coarse-grained aspects of parallelism, and it is more relevant for some of the modern settings of parallel computation that are designed for processing massive data sets.

[1] This is a draft and it's being polished. Please check again for updates. Comments and feedback would be greatly appreciated, and should be emailed to ghaffari@inf.ethz.ch.

Note: The material in the first six sections of this chapter are based on a compilation of several results in the PRAM model from 1980 to 1995. See the survey of Karp and Ramachandran [KR90] for a nice review of the results before 1990. The material in the last five sections of this chapter are based on a sampling of the recent work on MPC, starting circa 2010.

The interested reader may also want to consult a number of other sources, including: (A) a 1992 textbook of JaJa [JáJ92], titled "An Introduction to Parallel Algorithms", (B) a 1992 textbook by Leighton [Lei14] titled "Introduction to Parallel Algorithms and Architectures", (C) the lecture notes of a recent class titled Parallel Algorithms by Uzi Vishkin at University of Maryland, and (D) the lecture notes of a recent class titled Algorithm Design: Parallel and Sequential by Umut Acar and Guy Blelloch at Cargenie Mellon University. Finally, see our graduate level class Massively Parallel Algorithms for recent progress on algorithmic tools and techniques for MPC.

Contents

6.1 Warm Up: Adding Two Numbers

Instead of starting our discussion with a model description, let us begin with a simple example which nicely illustrates the meaning of "parallelism in computation". This is a computation that we learn in elementary school: adding two numbers. Suppose that we are given two n-bit binary numbers, $a = a_n, a_{n-1}, \ldots, a_1$ and $b = b_n, b_{n-1}, \ldots, b_1$, where a_i and b_i denote the i^{th} least significant bits of a and b, respectively. The goal is to output $a + b$, also in a binary representation.

A Basic Addition Algorithm—Carry-Ripple: A basic algorithm for computing the summation $a + b$ is *carry-ripple*. This is probably what most of us learned in the elementary school as "the way to compute summation". In the binary case, this algorithm works as follows: We compute the output bit s_1 by adding a_1 and b_1 (mod 2), but on the side, we also produce a carry bit c_1, which is 1 iff $a_1 + b_1 \geq 2$. Then, we compute the second output bit s_2 based on a_2, b_2, and c_1, and on the side also produce a carry bit c_2. The process continues similarly. We compute s_i as a (simple) function of a_i, b_i, and c_{i-1}, and then we also compute c_i as a side-result, to be fed to the computation of the next bit of output. Notice that this process can be built as a Boolean circuit with $O(n)$ AND, OR, and NOT gates, $O(1)$ many for the computation of each s_i and c_i.

Parallelism? A shortcoming of the above algorithm is that computing each bit of the output needs to wait for the computation of the previous bit to be finished, and particularly for the carry bit to be determined. Even if many of your friends come to your help in computing this summation, it is not clear how to make the computation finish (significantly) faster. We next discuss an adaptation of the above algorithm, known as *carry look-ahead*, that is much more parallelizable.

A Parallel Addition Algorithm—Carry-Look Ahead: The only problematic part in the above process was the preparation of the carry bits. Once we can compute all the carry bits, we can complete the computation to find the output bits s_i easily from a_i, b_i, and c_{i-1}. Moreover, the computations of different output bits s_i are then independent of each other and can be

performed in parallel. So, we should find a better way to compute the carry bits c_i.

Let us re-examine the possibilities for a_i, b_i, and c_{i-1} and see what c_i should be in each case. If $a_i = b_i = 1$, then $c_i = 1$; if $a_i = b_i = 0$, then $c_i = 0$, and if $a_i \neq b_i$, then $c_i = c_{i-1}$. Correspondingly, we can say that the carry bit is either *generated* (g), *killed* (k), or *propagated* (p). Given a_i and b_i, we can easily determine an $x_i \in \{g, k, p\}$ which indicates in which of these three cases we are.

Now, let us examine the impact of two consecutive bits, in the adder, on how the carry bit gets passed on. We can write a simple 3×3 "multiplication table" to summarize the effect. If $x_i = k$, then the overall effect is k regardless of x_{i-1}; if $x_i = g$, then the overall effect is g regardless of x_{i-1}; and if $x_i = p$, then the overall effect is the same as x_{i-1}.

Let $y_0 = k$ and define $y_i \in \{k, p, g\}$ as $y_i = y_{i-1} \times x_i$, using the above multiplication table. Here, y_0 indicates that there is no carry before the first bit. Once we compute y_i, we know the carry bit c_i: in particular, $y_i = k$ implies $c_i = 0$, and $y_i = g$ means $c_i = 1$. Notice that we can never have $y_i = p$ (why?).

So what remains is to compute y_i for $i \in \{1, \ldots, n\}$, given that $x_i \in \{k, p, g\}$ are known. Notice that each x_i was calculated as a function of a_i and b_i, and all in parallel. Computing y_i is a simple task for parallelism, and it is generally known as *Parallel Prefix*. We next explain a classic method for it: We build a full binary tree on top of the indices $\{1, \ldots, n\}$. Then, on this tree, we pass up the product of all descendants, toward the root, in $O(\log n)$ parallel steps. See the red arrows and values in Figure 6.1. This way, each node in the binary tree knows the product of all x_i in indices i that are its descendants. Then, using $O(\log n)$ extra parallel steps, each node passes to each of its children the product of all of the x_i in the indices that are preceding the rightmost descendant of that child (pictorially, we are imagining the least significant bit in the rightmost part and the most significant part in the leftmost part). See the green arrows and values in Figure 6.1. At the end, each leaf (index i) knows the product of all indices before itself and thus can compute y_i.

The above computation is made of $O(\log n)$ steps, where in each step all tasks can be performed in parallel. Moreover, each step involves at most $O(n)$ computations. In fact, the total amount of computation is also $O(n)$ (why?). This whole process for computing the summation can be built as

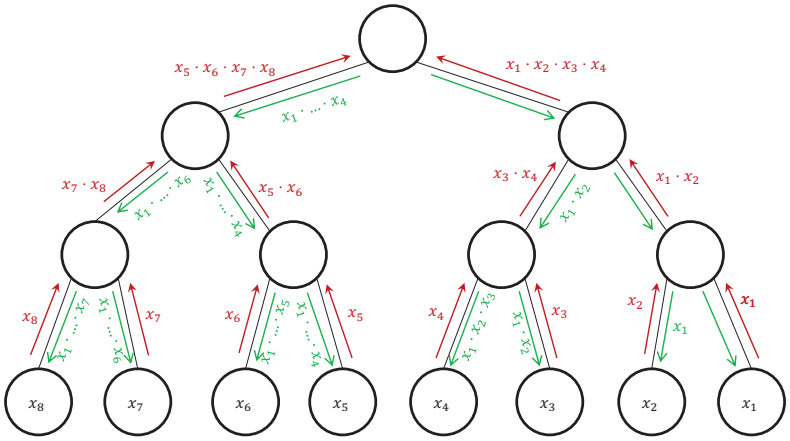

Figure 6.1: Sending up and down the partial multiplications. Red arrows and the red text next them show the values that are sent up, in $O(\log n)$ parallel steps. Green arrows and the text next to them show the values that are sent down, afterward, in some extra $O(\log n)$ parallel steps.

a circuit with depth $O(\log n)$, composed of $O(n)$ gates (each of the units of computation in the above process can be computed using $O(1)$ AND, OR, and NOT gates).

6.2 Models and Basic Concepts

Given the above intuitive warm up, we are now ready to discuss the model that we use to study parallel algorithms. There are two closely related models which both nicely capture parallelism in computation. One is based on logic circuits and the other is PRAM. We next discuss these two models. The model descriptions here are rather brief and try to stay away from some of the detailed technicalities in the model definitions. We provide a level of formality that will suffice for the rest of the material in this chapter, and particularly for understanding the main algorithmic ideas, without getting caught up in lower order details.

6.2.1 Circuits

Here, we consider Boolean circuits made of AND, OR, and NOT gates, which are connected by wires. Two key measures of interest are the num-

ber of gates in the circuit, and the depth (the time needed for producing the output, from the time that the input is provided, if we assume that each gate operation takes one time-unit, i.e. one clock cycle in synchronous circuits). We can classify problems by the related measures of the circuits that can solve these problems. Before that, we should explain one possible distinction: There are two variations possible on the gates: we might assume that the fan-in— that is, the number of inputs to each gate—is bounded or unbounded. The class $NC(i)$ denotes the set of all decision problems that can be decided by a Boolean circuit with $poly(n)$ gates of at most two inputs and depth at most $O(\log^i n)$, where n denotes the input size, i.e., the number of input bits to the circuit. The class $AC(i)$ denotes the set of all decision problems that can be decided by a Boolean circuit with $poly(n)$ gates of potentially unbounded fan-in and depth at most $O(\log^i n)$.

Lemma 6.1. *For any* k*, we have* $NC(k) \subseteq AC(k) \subseteq NC(k+1)$.

Proof Sketch. The first relation is trivial. For the second, use $O(\log n)$-depth binary trees of bounded fan-in gates to replace unbounded fan-in gates. □

We define $NC = \cup_i NC(i) = \cup_i AC(i)$.

6.2.2 Parallel Random Access Machines (PRAM)

In the PRAM model, we consider p number of RAM processors, each with its own local registers, which all have access to a global memory. Time is divided into synchronous steps and in each step, each processor can do a RAM operation or it can read/write to one global memory location. The model has four variations with regard to how concurrent reads and writes to one global memory are resolved: Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), Exclusive Read Concurrent Write (ERCW), and Concurrent Read Concurrent Write (CRCW). When concurrent writes on the same memory location are allowed, there are variations on how the output is determined. A simple rule is to assume that an arbitrarily chosen one of the write operations takes effect. Similar to NC, we use variants with index k—e.g. $CRCW(k)$—to denote decision problems that can be computed by the corresponding version of the PRAM model with $poly(n)$ processors and in $O(\log^k n)$ time steps.

Lemma 6.2. *For any* k*, we have* $\mathrm{CRCW}(k) \subseteq \mathrm{EREW}(k+1)$.

Proof. The proof is based on a binary tree idea, is similar to that of Lemma 6.1, and is left as an exercise. □

It can be seen that PRAM can simulate circuits and vice versa. And as a consequence, we see that $\mathrm{NC} = \cup_k \mathrm{EREW}(k)$. Thus, these two models are tightly related.

The variants mentioned above have some difference in terms of their power. For example, computing the maximum of n numbers can be done in $O(\log n)$ time by any of the variants (and in particular, in the weakest of them, EREW), using a simple binary tree. And it is also known that computing this maximum requires $\Omega(\log n)$ time-steps in the EREW version. However, this can be done in $O(1)$ time-steps in the CRCW model, using $O(n^2)$ processors, as follows: Initialize n entries in the register to be all 0. Use $\binom{n}{2}$ processors, one responsible to compare one pair of numbers. If the i^{th} item loses the comparison (i.e., is less than some j^{th} one), write 1 in the i^{th} entry of the register. Then, make n processors check the n register entries and if an entry is still 0, which means it did not lose a comparison, write its value in the output register.

Exercise 6.1. Maximum, using Fewer Processors
The maximum of n *entries can be computed in* $O(\log \log n)$ *time-steps, using the CRCW version of* PRAM *with* n *processors.*

6.2.3 Some Basic Problems

To get used to these models, we next discuss some example problems. These problems are about simple data operations in arrays and linked lists, and they will be used later as basic subroutines, in more involved computations. We use the PRAM model in these discussions.

Parallel Prefix We have already talked about the *parallel prefix* problem, in the context of summing two n-bit numbers. Concretely, the input is an array A of length n and we want to compute an array B of length n that contains all the prefix sums, i.e., $B[j] = \sum_{i=1}^{j} A[i]$, for all $j \in \{1, \ldots, n\}$. An $O(\log n)$ time EREW PRAM algorithm for this follows from the binary tree idea discussed above, which uses $O(n)$ processors, and also performs $O(n)$ computations in total.

List Ranking We are given a linked list as input, represented by a content array $c[1..n]$ and a successor pointer array $s[1..n]$. That is, $c(i)$ gives the value of the element at location i and $s(i)$ gives the location $j \in \{1, \ldots, n\}$ of the successor to $c(i)$ on the list. The desired output is to know all the suffix sums, from any starting point in the linked list to the end of it. For convenience, we assume that the last element of the list has content zero and its successor pointer points to its own location. We next describe an $O(\log n)$-time EREW PRAM algorithm for this problem. This algorithm is based on the idea of *pointer jumping*, which gets used frequently in the design of parallel algorithms.

List Ranking via Pointer Jumping The algorithm is made of $\log n$ iterations, each of which has two steps, as follows:

(1) In parallel, for each $i \in \{1, \ldots, n\}$, set $c(i) = c(i) + c(s(i))$.

(2) In parallel, for each $i \in \{1, \ldots, n\}$, set $s(i) = s(s(i))$.

The operation in the second step, where we set the successor to be the successor of successor, is knows as *pointer jumping*.

Lemma 6.3. *At the end of the above process, for each* $i \in \{1, \ldots, n\}$, $c(i)$ *is equal to the summation of the values from the location of that element to the end of the linked list.*

Proof Sketch. After $\log n$ iterations, for all i, the pointer $s(i)$ points to the end of the linked list. By induction, we can show that after each iteration, $c(i)$ is equal to the sum of the elements with rank $r(i), r(i) - 1, r(i) - 2, \ldots, r(s(i)) + 1$, where the rank $r(i)$ denotes the distance of the i^{th} element from the end of the input linked list. □

The above algorithm uses $O(\log n)$ time and $O(n \log n)$ computations.

6.2.4 Work-Efficient Parallel Algorithms

As one can see in the context of the examples discussed above, we can view each computational process as a sequence of rounds, where each round consists of a (potentially large) number of computations (e.g., memory access or RAM operations) that are independent of each other and can be performed in parallel. In this view, we refer to the total number of rounds

as the *depth* of the computation and we refer to the summation of the number of computations performed over all the rounds as the total *work*. Naturally, our primary goal is to have a depth as small as possible, similar to the depth of the circuits. A secondary goal is to also have a small total work. Then, we can relate this depth and work to an actual *time* measure for our computation on parallel processing systems, depending on how many processors we have.

Theorem 6.4 (Brent's Principle). *If an algorithm does x computations in total and has depth t (i.e., t rounds or perhaps more formally a critical path of length t), then using p processors, this algorithm can be run in $x/p + t$ time.*

Proof. If x_i is the amount of work in the i^{th} time-step, we can perform all of them in $\lceil x_i/p \rceil \leq x_i/p + 1$ time, using our p processors. We get that the total time needed is at most $\sum_{i=1}^{t} x_i/p + 1 = x + t$, because $x = \sum_{i=1}^{t} x_i$. □

Remark 6.1. *The above principle assumes that we are able to schedule the computational task of each round on the processors in an ideal way and in particular, it is possible for each processor to determine the steps that it needs to simulate in an online fashion. We will later see that in certain cases, this can be quite non-trivial.*

Exercise 6.2. Brent's Principle on Parallel Prefix
Use Brent's principle to determine the smallest number of processors that would allow us to run the Parallel Prefix algorithm which we saw above in $O(\log n)$ time. Recall that algorithm had $O(\log n)$ depth and $O(n)$ total computation. Explain how the algorithm with this small number of processors works, that is, what each processor needs to do in each time step.

As it is clear from the above principle, to optimize the time needed by the algorithm, besides desiring a small depth, we also want a small amount of work. In the example problems discussed above, we saw a parallel algorithm for list ranking problem which uses an $O(\log n)$ factor more computational work than its sequential counterpart. It is far more desirable if the total amount of work is proportional to the amount of work in the sequential case. We call such parallel algorithms *work-efficient*. Once we have a work-efficient algorithm, in some sense, there is no significant

loss due to the parallelism. Moreover, we can then use a small number of processors to simulate the setting with many processors, in a fully efficient way. In the next sections, when discussing various problems and parallel algorithms for them, our ultimate goal will be to obtain work-efficient or nearly work-efficient algorithms with a depth that is as small as possible.

6.3 Lists and Trees

6.3.1 List Ranking

Previously, we saw an algorithm for list ranking with $O(\log n)$ depth and $O(n \log n)$ computational work. We now go back to this problem and try to improve the algorithm so that it is work-efficient. That is, the goal is to obtain an $O(\log n)$ depth algorithm with $O(n)$ total work[2].

Intuitive Comparison of List Ranking and Parallel Prefix: Notice that in the closely related problem of parallel prefix, we already have an algorithm with $O(\log n)$ depth and $O(n)$ total work. In some sense, we can view the parallel prefix algorithm via binary tree as reducing the problem to computing prefix sums on the $n/2$ elements at even positions on the array. This reduction is produced in $O(1)$ time and with $O(n)$ work. Moreover, once we solve this problem, we can extend the solution to the full prefix sum problem on n elements, using $O(1)$ more time and $O(n)$ work. What is the problem in doing a similar reduction for list ranking?

In short, the difficulty stems from the following issue: An element of the linked list does not know whether it is at an odd or even position of the list (unless we have already computed these positions, but that is actually the same as the list ranking problem). Thus, there is no clear way of structuring which elements should go to the lower level of the recursion on an ideally smaller linked list. We next explain how one can overcome this difficulty.

Outline of the Algorithm for List Ranking: Notice that we do not need exactly even positions; it suffices to construct a set S of cn elements in the list, for

[2]We will see an algorithm that provides bounds that are slightly weaker than this, and we will comment where the final improvement comes from to give this ideal result.

some constant $c < 1$, such that the distance between any two consecutive members of S is small. Then, we can solve the linked list problem by recursion, as follows:

(A) We build a contracted linked list for S, where each element in S has the first next linked-list element that is in S as its successor. The value of each element in the new linked list would be the sum of the values of the elements starting from (and including) itself and ending right before the first next element in S (i.e., its new successor).

(B) Recursively, solve the problem for this contracted linked list, whose size is $|S|$.

(C) Extend the solution to all elements, in time proportional to the maximum distance between two consecutive elements of S in the original list, and with work proportional to the length of the original list.

We can repeat the above idea, recursively. Once the recursion is applied enough that the contracted list size falls below $n/\log n$, we can use the basic linked list algorithm that we saw above. That would solve the instance in $O(\log(n/\log n)) = O(\log n)$ time and with $O(n/\log n \cdot \log(n/\log n)) = O(n)$ total work. If we have a way of choosing S such that $|S| < cn$ for a constant $c < 1$, then $O(\log\log n)$ repetitions of link list contraction suffice. After $O(\log\log n)$ repetitions, we reach such a case where the contracted list size falls below $n/\log n$.

We next discuss two remaining questions in this outline: (1) How should we choose S so that it has the desired properties, (2) How can we "compact" the linked list and build the new linked list for S, as desired in part (A) mentioned above.

Selecting S: We use a simple randomized idea to mark an *independent set* I of elements, i.e., a set such that no two elements of it are consecutive in the linked list. We will think of removing the elements of I from the list, and thus, the set S in the above discussions will simply be all elements besides those of I. Call each element of the linked list head or tail, using a fair coin toss, and independently of each other. Then, an element is in I if and only if it holds a head coin and its successor holds a tail coin. We have $\mu = \mathbb{E}[|I|] \geq n/8$, because if we break all elements into $n/2$ disjoint pairs of consecutive elements, each first element in a pair has at least $1/4$ probability

to be in I. Moreover, these are independent for different pairs. Thus, using the Chernoff bound, we can see that the probability that we have a $\delta = 0.5$ relative deviation from this expectation and get $|I| \leq (1-\delta)\mu = n/16$ is at most $e^{-\delta^2\mu/2} = e^{-n/64} \ll 1/n^{10}$, for n larger than some constant. Thus, with probability at least $1-1/n^{10}$, we have $|I| \geq n/16$. That means $|S| \leq 15n/16$, with high probability.

Compacting the Linked List: To prepare for a recursion, we need to build a linked list of length $|S|$, keep its successor pointers in an array of length $|S|$, and provide an updated content array, for each of the new items (with content equal to the sum from that element to the right before the first next element of S in the linked list). We can solve the compaction problem, using the parallel prefix algorithm. In particular, we first want to number the items in S using distinct numbers from $\{1, 2, \ldots, |S|\}$ (ignoring their order in the linked list). This numbering can be done using parallel prefix, by starting with 1 for each item in S and 0 for each item in I, in the array, and then computing the parallel prefix on the array that keeps the items. We emphasize that the numbering does not need to be monotonic in the linked list; we just need a way of getting distinct numbers for distinct elements.

Once we have these numbers in $\{1, 2, \ldots, |S|\}$, it takes a few simple operations to effectively remove the elements of I and set the appropriate content and successor arrays for those in S. In particular, since I is an independent set, for each element $s \in S$, the successor of S in the new linked list is either its previous successor (in case that one is in S) or the successor of its previous successor. This successor can be found in $O(1)$ depth, for all elements of S in parallel, using $O(|S|)$ total computation. Similarly, we can prepare the content for the new linked list: corresponding to the previous two cases, this new content is either just the previous content or the summation of previous content of s and its successor.

Overall Complexity: The above approach via parallel prefix requires $O(n)$ work and $O(\log n)$ depth, for just one level of compaction. Over all the $O(\log \log n)$ compaction levels until we reach size $n/\log n$, we use $O(n)$ computational work (why?) and $O(\log n \cdot \log \log n)$ depth. While this algorithm is work-efficient, this depth bound is an $O(\log \log n)$ factor away from the ideal bound. There are known methods for reducing the depth complexity to $O(\log n)$. Below, we briefly comment on one. For the rest of

this chapter, we will frequently make use of the list ranking problem as a subroutine, and we will assume that we have a list ranking algorithm with the ideal bounds, i.e., $O(\log n)$ depth and $O(n)$ work.

Known Improvement* We next briefly comment on one, but we note that this is an optional part of the material for this chapter. One particular list ranking algorithm with depth $O(\log n)$ and $O(n)$ work follows from the above outline, but where we use a better parallel prefix algorithm due to Cole and Vishkin [CV89]. That parallel prefix algorithm has $O(\log n/\log\log n)$ depth, and thus, applying it $O(\log\log n)$ times —as we needed above — requires $O(\log n)$ depth. In a very rough sense, the idea in that faster parallel prefix algorithm is to do a pre-computation and build a look-up table, which can be used to quickly infer partial sums, on a few bits. To simplify the intuition, suppose that our elements have only $O(1)$ bits (this assumption is not necessary but without it the process requires a more complex description). The method prepares for every possibility of $L = \Theta(\log^{0.5} n)$ consecutive elements, by building a look-up table that stores the partial sum for each of the possible cases, given that all of this needs $o(n)$ work. Once the input is provided, we perform a certain fast look-up operation to find the relevant sum for each portion of length L, using just $O(1)$ computational depth. This enables us to effectively reduce the problem size by an L factor, with $O(1)$ computational depth. Since $O(\log_L n) = O(\log n/\log\log n)$ repetitions suffice and each repetition has depth $O(1)$, we get a depth of $O(\log n/\log\log n)$ for parallel prefix. Though, we do not discuss that method further here, as it requires a lengthy description.

6.3.2 The Euler Tour Technique

In this section, we see a technique that leads to work-efficient algorithms with $O(\log n)$ depth for a number of problems related to tree structures. This includes problems such as rooting the tree in a given root node r, computing the pre-order or post-order number of each vertex, the depth of each vertex, and the number of the descendants of each vertex.

Input Format: Suppose that the we are given a tree $T = (V, E)$ with $|V| = n$ as an input. This tree input is provided as the adjacency lists of its vertices.

More precisely, for each vertex $v \in V$, the vertices adjacent to v are given in a linked list $L[v] =< u_0, u_1, u_2, \ldots, u_{d-1} >$, where d is the degree of the vertex v. Here, the order of the vertices in the list can be chosen arbitrarily.

Exercise 6.3. adjacency matrix to adjacency lists
Suppose that instead of adjacency lists, the graph is input as an $n \times n$ binary adjacency matrix where the entry at location (i, j) is 1 if the i^{th} and the j^{th} nodes are adjacent, and 0 otherwise. Devise an algorithm with $O(\log n)$ depth and $O(n^2)$ work that transforms this adjacency matrix to adjacency linked lists, one for each vertex.

First Problem — Rooting the Tree and Determining Parents: Besides the tree $T = (V, E)$ provided as input by its adjacency linked lists, suppose that we are also given a root vertex $r \in V$. The objective is to root the tree such that each node $v \neq r$ knows its parent $parent(v)$, i.e., the neighbor that is strictly closer to the root r.

Defining an Eulerian Tour Consider a directed variant $T' = (V, E')$ of the tree $T = (V, E)$ where we replace each edge $e = \{u, v\} \in E$ with two directed arcs $< u, v >$ and $< v, u >$, in the opposite direction of each other. Then, the directed graph T' has an Eulerian tour, i.e., a directed cycle that goes through each arc exactly once (why?). Moreover, we can define such a tour easily, by identifying for each arc $< u, v > \in E'$ the successor arc $< v, w >$ that the cycle should take after arriving at node v through arc $< u, v >$. In particular, given the adjacency lists $L[v]$ which are provided in the problem, we define the successor pointers as follows: for each $i \in \{0, \ldots, d-1\}$ where d denotes the degree of v, set $s(< u_i, v >) =< v, u_{(i+1) \bmod d} >$. That is, intuitively, we turn the adjacency list $L[v]$ into a cyclic order and the successor of each incoming arc from a neighbor is the outgoing arc to the next neighbor, in this cyclic order. One can also define predecessor pointers similarly, by setting $p(< v, u_i >) =< u_{(i-1) \bmod d}, v >$.

Determining Parents using the Eulerian Tour: Break the Eulerian cycle into a path by effectively removing the last incoming arc $< u_{d(r)-1}, r >$ of the root node r (and setting the successor of its predecessor to be NULL), where $d(r)$ denotes the degree of r. We next want a numbering $\eta : E' \to \{1, \ldots, 2n-2\}$ of the arcs in the cycle, such that it is monotonically increasing along the

successor pointers of the arcs of the Eulerian cycle. We can compute such a numbering using an instance of the *list ranking* problem: we put the content of each arc to be 1 and then we compute all the partial prefix sums of these contents, according to the linked list of successor pointers. This can be done using a parallel algorithm with $O(\log n)$ depth and $O(n)$ total computation work, as we saw in the previous section. Given this numbering, it is now easy to identify the parents. For each node v and each edge $\{v, u\} \in E$, either v is the parent of u or node u is the parent of v. If $\eta(< v, u >) < \eta(< u, v >)$— that is, if the Eulerian path traverses first from v to u and then, sometime later, from u to v—then node v is the parent of node u. Otherwise, u is the parent of v. We can check the condition for all edges $\{v, u\} \in E$ in parallel. Thus, after having the numbering η, with $O(1)$ depth and $O(n)$ work, we can determine the parent $parent(v)$ of each node v.

Exercise 6.4. Brent's Principle on Rooting a Tree
Explain how to compute the parents for all nodes, using $O(n/\log n)$ *processors and in* $O(\log n)$ *time. In particular, what should each processor do in each time step? You can assume that internal parts of the prefix computation on the linked list already works with* $O(n/\log n)$ *processors and in* $O(\log n)$ *time.*

Second Problem — Computing a Pre-Oder Numbering of Vertices: Consider a Depth First Search traversal of the vertices (according to the adjacency lists, which actually happens to coincide with how we defined the Eulerian path). Our objective is to compute a pre-oder numbering $pre : V \to \{0, \ldots, n-1\}$ of the vertices. That is, in this numbering, for each node v, first v appears, then a pre-order numbering of the subtree rooted in the first child of v, then a pre-order numbering of the subtree rooted the second child of v, etc.

Using the Eulerian tour technique, we can solve the problem easily, as follows: After having identified the parents as above, we now define a new weight for the arcs. We set $w(< parent(v), v >) = 1$ and $w(< v, parent(v) >) = 0$. Notice that the former are forward arcs in the DFS and the latter are backward arcs. Then, we compute all the prefix sums of these weights, on the linked list provided by our Eulerian path (i.e., maintained by the sucessor pointers). Hence, each arc knows the number

of forward arcs before it (and including itself), in the Eulerian path. Set $\mathrm{pre}(r) = 0$ for the root node r. For each node $v \neq r$, set $\mathrm{pre}(v)$ to be the prefix sum on the arc $< \mathrm{parent}(v), v >$, i.e., the total number of forward arcs before and including this arc. This gives exactly our desired pre-order numbering (why?).

Exercise 6.5. Post-order Numbering

Modify the above approach so that it provides a post-order numbering $\mathrm{post} : V \to \{0, \ldots, n-1\}$ *of the vertices. That is, for each node* v*, we have a post-order numbering of the subtree rooted in the first child of* v*, then a post-order numbering of the subtree rooted in the second child of* v*, etc, followed by node* v *itself. In particular, you should have* $\mathrm{post}(r) = n-1$*. Argue that the algorithm provides the correct ordering, and explain why it has* $O(\log n)$ *depth and* $O(n)$ *computation.*

Third Problem — Computing the Depth of Vertices: The objective is to determine for each node v the length of the shortest path connecting the root to node v. Hence, $\mathrm{depth}(r) = 0$ and for each node v, we have $\mathrm{depth}(v) = \mathrm{depth}(\mathrm{parent}(v)) + 1$.

The solution is similar to the method for computing a pre-order numbering, with one exception in defining the weight of the backward arcs. In particular, we set $w(< \mathrm{parent}(v), v >) = 1$ and $w(< v, \mathrm{parent}(v) >) = -1$. Then, we define the depth of each node v to be the prefix sum of the arc $< \mathrm{parent}(v), v >$, on the linked list provided by the successor pointers. Notice that the prefix sum on the arc $< \mathrm{parent}(v), v >$ is exactly the number of forward arcs from the root r until reaching v, on the shortest path connecting r to v. Concretely, for all the other forward arcs the appear before v on the Eulerian path, if they are not on this shortest path connecting r to v, their contribution to the sum gets canceled when the path traverses the same edge backward.

Exercise 6.6. Number of Descendants

Devise a parallel algorithm with $O(\log n)$ *depth and* $O(n)$ *total computation that computes for each node* v *the number of its descendants, i.e., the total number of nodes in the subtree rooted at node* v*.*

Exercise 6.7. Sum of Descendant Leaves

Suppose that each leaf node u *is given a number* $b(u)$*. Devise a parallel algorithm with* $O(\log n)$ *depth and* $O(n)$ *total computation that*

computes for each node v the summations of the $b(u)$ values over all the leaves u that are descendants of v.

6.4 Merging and Sorting

6.4.1 Merge Sort

Recall the *merge sort* algorithm: we break the input of n items into two parts of size at most $\lceil n/2 \rceil$, sort each of them separately, and then we merge the resulting two sorted arrays of length $\lceil n/2 \rceil$ into a sorted array of length n. Clearly, the work in sorting the two parts are independent of each other, and thus, can be performed in parallel. The main step that we need to investigate closer, from the perspective of parallelism, is the merging step.

Basic Merge Problem: Consider two sorted arrays $A[1..n/2]$ and $B[1..n/2]$, each of length $n/2$, that contain comparable items (for simplicity, suppose that no two items are equal). How can we merge these two arrays into a sorted array $C[1..n]$ of length n?

Basic Merging Algorithm: For each item $x \in A \cup B$, it suffices to know the number k of items in $A \cup B$ that are smaller than this item x. Then, we would set $C[k+1] = x$. In particular, we use $n/2$ binary searches, one for each item $A[i]$, so that this item (formally the processor simulating this item's action) knows the number j of items in B that are smaller than $A[i]$. Then, there are exactly $(i-1)+j$ items that are smaller than $A[i]$ and we should set $C[i+j] = A[i]$. The $n/2$ binary searches can be done in $O(\log n)$ depth, using $O(n \log n)$ work, in total. We can then do something similar for items of B so that each of them also knows the number of items in A that are smaller than it, and then goes to the appropriate place in C.

Parallel Merge Sort via Basic Merging: If we use the above basic merging algorithm for each step of the merge, we get a sorting algorithm with depth $\sum_{i=1}^{\log n} \log(n/2^i) = \sum_{i=1}^{\log n} (\log n - i) = O(\log^2 n)$. Furthermore, the total work is $\sum_{i=1}^{\log n} 2^i \cdot O(\frac{n}{2^i} \log(\frac{n}{2^i})) = \sum_{i=1}^{\log n} O(n(\log n - i)) = O(n \log^2 n)$.

Ideal Bounds? It would be much more desirable if we could have a sorting algorithm with depth $O(\log n)$ and total work $O(n \log n)$. The total work would be similar to sequential comparison-based algorithms and we would get that work-efficiency with only an $O(\log n)$ depth. Below, we discuss a method that gets us almost there. Concretely, it achieves these bounds up to an $O(\log \log n)$ factor. We also comment briefly how that factor can be removed. The general methodology that we will explain is quite useful and similar ideas get used frequently for improving the bounds of basic algorithms, by first solving a properly chosen subset of the problem.

Improved Merging Algorithm Consider two sorted arrays $A[1..n]$ and $B[1..m]$. The generality of allowing n and m to be different will be useful, as we create a recursive algorithm. First, we choose $\sqrt{n}$ evenly spaced-out fenceposts $A[\alpha_1], A[\alpha_2], \ldots, A[\alpha_{\sqrt{n}}]$ where $\alpha_i = (i-1)\sqrt{n}$. Similarly, we choose $\sqrt{m}$ evenly spaced-out fenceposts $B[\beta_1], B[\beta_2], \ldots, B[\beta_{\sqrt{m}}]$ where $\beta_i = (i-1)\sqrt{m}$. We perform all the $\sqrt{nm}$ pairwise comparisons between these fenceposts using $\sqrt{nm} = O(n+m)$ processors.

Then, use the same number of processors so that in $O(1)$ time in the CREW model, we find for each α_i the j such that $B[\beta_j] \leq A[\alpha_i] \leq B[\beta_{j+1}]$ (how?). Here, for the sake of simplicitly of notation, suppose $B[0] = -\infty$ and $B[m+1] = +\infty$, so that β_j is well-defined. In parallel, compare $A[\alpha_i]$ to all $\sqrt{m}$ elements in $B[\beta_j..\beta_{j+1}]$. Then, we know the exact rank of each α_i in the merge of A and B. We can use these α_i fenceposts to break the problem into $\sqrt{n}$ many subproblems, each about merging $\sqrt{n}$ elements of A with some number of elements of B. The total number of elements of B, among different subproblems, remains m but there is no guarantee on their size being smaller and it is possible that one of them is as large as m. Regardless, we get the following recursion about the depth: $T(n) = O(1) + T(\sqrt{n})$. Thus, after $O(\log \log n)$ recursions, the problem boils down to finding one items placement in another array of length m: a problem that can be solved using m processors in $O(1)$ time, in the CREW model.

Parallel Merge Sort via Improved Merging: As a corollary, we can merge two sorted arrays of length $n/2$ with each other, forming a sorted array of length n, in $O(\log \log n)$ time and using $O(n \log \log n)$ computation. Plugging this into the merge sort outline provides a merge sort algorithm with $O(\log n \log \log n)$ depth and $O(n \log n \log \log n)$ computation.

Further Improvement* There is a method that removes the $\log\log n$ factor from the time and computation complexity of the above process, by "pipelining" the merges. This was presented by Richard Cole [Col88]. In a rough sense, partial merge results of one level are used to obtain partial merge results for the higher level, in a way that there is $O(1)$ time delay between each two consecutive levels. Since the method is detailed, we do not discuss it here.

6.4.2 Quick Sort

Let us recall quick sort: pick a random index $i \in \{1, \ldots, n\}$ of the array $A[1..n]$, break the array into two parts of $B[1..j]$ and $B'[1..(n-j)]$ by comparing all elements to $A[i]$, and putting the smaller ones in B. Then, we recurse on each of B and B' separately. What are the performances of this algorithm, if we turn it into a parallel algorithm?

Basic Quick Sort: The main operation is to compare all elements with $A[i]$ and building B and B'. We can easily compare each $A[k]$ with $A[i]$. But even if we know $A[k] < A[i]$ which implies we should write $A[k]$ somewhere in B, to know exactly where, we need some more care. Suppose for each k, the bit $b(k) \in \{0, 1\}$ indicates whether $A[k] \leq A[i]$ or not (1 in the former case). Then, a parallel prefix sum in A on content $b(k)$ can determine for each k such that $A[k] \leq A[i]$ the number $x(k)$ of indices $k' \leq k$ such that $A[k'] \leq A[i]$. Then, we should write $B[x(k)] = A[k]$. A similar operation can be used for B' by setting $b(k) = 1$ iff $A[k] > A[i]$. These parallel prefix subproblems can be solved with $O(\log n)$ depth and $O(n)$ work, using what we saw previously in this chapter. This is the amount of work and depth that we use for performing one level of quick sort. What remains is to analyze the number of recursion levels.

We claim that this is $O(\log n)$, with high probability—e.g., with probability $1 - 1/n^3$. Let us call a level of quick sort, on an n length arrary, *success* if the random index $i \in [n/4, 3n/4]$, and *fail* otherwise. Then, the probability of success is at least $1/2$. In one branch of the recursion, once we have $\log_{4/3} n$ successes, the array size is 1 and we are done with that branch of recursion (why?). Now, what is the probability that we have a branch of length $20 \log n$ but still less than $\log_{4/3} n \leq 2.5 \log n$ successes? Notice that each step of a branch is a success with proba-

bility at least $1/2$. Thus, in a branch of length $20 \log n$, we expect at least $\mu = 10 \log n$ success. Using a Chernoff bound, we can see that the probability that we have a $\delta = 0.75$ relative deviation from this expectation and the number of successes is below $(1-\delta)\mu = 2.5 \log n$ is at most $e^{-\delta^2\mu/2} = e^{-(0.75)^2 \cdot 10 \log n/2} = e^{-2.8125 \log n} < 2^{-4 \log n} = 1/n^4$. Thus, the number of recursion levels before we have at least $\log_{4/3} n$ successes is at most $50 \log n$, with probability at least $1 - 1/n^4$. A union bound over all branches suffices to show that all branches of recursion finish within $50 \log n$ repetitions, with probability at least $1 - 1/n^3$. Now, given that we have $O(\log n)$ recursion levels, each implemented in $O(\log n)$ depth and $O(n)$ work, the overall complexities are $O(\log^2 n)$ depth and $O(n \log n)$ computational work.

Ideal Bounds? While the total work is as desired—e.g., similar to the best possible sequential comparison-based sorting algorithms— the depth has room for improvement. The main shortcoming comes from the somewhat inefficient splitting: we perform an $O(\log n)$ depth computation but we break the size of the (worst branch of the) problem only by a constant factor.

Improved Quick Sort using Multiple Pivots—Outline: Next, we present an outline of an improved parallel quick sort algorithm. The base idea is to use multiple pivots, so that the effort of having $O(\log n)$ depth needed to send each element to the appropriate branch of the problem is balanced with a more significant in the reduction of the problem (i.e., much than the constant factor reduction that we had above).

(A) Pick $\sqrt{n}$ pivot indices in $\{1, ...n\}$ at random (with replacement).

(B) Sort these $\sqrt{n}$ pivots in $O(\log n)$ depth and using $O(n)$ total work (how? hint: we can perform all the pairwise comparisons simultaneously and then use parallel prefix on the results to know the number of pivots smaller than each pivot).

(C) Use these sorted pivot elements as *splitters*, and insert each element in the appropriate one of the $\sqrt{n}+1$ branches — between two consecutive splitters or alternatively, before the smallest or after the largest — in $O(\log n)$ depth of computation and $O(n \log n)$ total work. Notice

that two things need to be done: (1) Identifying for each element the subproblem in which this element should proceed — this can be done using n separate binary searches, one for each element, searching in between the sorted splitters. (2) Creating an array for the subproblem between each two splitters, which contains all the elements in that subproblem[3]. This second part can also be done using $O(\log n)$ depth $O(n \log n)$ work and is left as Exercise 6.8 presented below.

(D) Recurse on each subproblem between two consecutive splitters. In this recursion, once a subproblem size reaches $O(\log n)$—where n is the size of the original sort problem— solve it in $O(\log n)$ time and with $O(\log^2 n)$ work, by a brute force deterministic algorithm (how? hint: simultaneously for all i, compare element i with all the other elements j, and compute the number of them that are smaller than element i).

Exercise 6.8. Generating Arrays for Subproblems
Suppose that we are given an array of length n and each element in it is tagged with a number in $\{1, 2, \ldots, \sqrt{n}\}$, which indicates the index of its subproblem. Devise an algorithm that in $O(\log n)$ depth and using $O(n \log n)$ work, creates one array A_i for each of the subproblems $i \in \{1, \ldots, \sqrt{n}\}$, holding all the elements tagged with number i. Each element should be in exactly one array and the summation of the lengths of the arrays should be n.

Hint: *First, think about creating linked lists instead of arrays. In particular, imagine a balanced binary tree on the n elements, where each node keeps a number of linked lists, one for each of the subproblems present in its descendants. Then, when you pass up the linked lists, from the children of one node v to the node v itself, you can merge these linked lists in $O(1)$ depth.*

Intuition of the Analysis: Intuitively, we expect each of the branches to have size roughly $\sqrt{n}$. Thus, if we use $T(n)$ to denote the depth of the sorting algorithm on arrays of length n, we intuitively have the following recursion: $T(n) = O(\log n) + T(\sqrt{n})$. Hence, $T(n) = O(\log n + \log n/2 + \log n/4 +$

[3] We do not want arrays of length n here, for each subproblem, as that would incur $O(n\sqrt{n})$ work for the recursions, even to find the non-empty entries. Each array should be of size exactly the same as the number of the elements in the recursion.

$\dots) = O(\log n)$. Similarly, the work per element for finding its subbranch follows a recursion as $W(n) = O(\log n) + W(\sqrt{n}) = O(\log n)$. This is a total of $O(n \log n)$ over all elements. The extra work for comparing the pivots is $O((\sqrt{n})^2) + \sqrt{n} \cdot O((n^{1/4})^2) + \cdots = O(n \log n)$. Thus, intuitively, we expect the above algorithm to provide the desired bounds of $O(\log n)$ depth and $O(n \log n)$ work. However, one has to be cautions that doing such an argument based on just expectations is not a proper analysis, and such intuition can deceive us into making incorrect claims; the subtlety is that the computation has many branches and, something could go wrong in one of them, even if the expected behavior of each is as desired. In the following, we turn the above intuition into a formal analysis, by providing a probabilistic analysis for each branch of the recursion and then taking all the branches into account, using a simple union bound.

Analysis: We first analyze the depth of the computation, throughout the recursion. Let us classify the problem based on the size: we say the problem size is in class i if its size is in $[n^{(2/3)^{i+1}}, n^{(2/3)^i}]$. Notice that when we start with a problem in class i—whose size is at most $n^{(2/3)^i}$—for one subproblem, the expected size is at most $\sqrt{n^{(2/3)^i}} = n^{(2/3)^i \cdot 1/2}$. We say this branch of recursion fails at this point in class i if the size of the subproblem is at least $n^{(2/3)^i \cdot 2/3} \gg n^{(2/3)^i \cdot 1/2}$. Notice that in this case, the problem fails to move to class $i+1$ and remains in class i (temporarily). We next upper bound the probability of this failure and also use that to upper bound the probability of failing many times at a level i. These are then used to argue that, with high probability, we do not remain within each level too many times and thus we can upper bound the computational depth of the recursion.

Lemma 6.5. *The probability of failing is at most* $\exp(-\Theta(n^{(2/3)^i \cdot 1/6}))$.

Proof. Consider the time we start with a problem in class i—whose size is at most $\eta = n^{(2/3)^i}$, and let us focus on one of the subproblems created there. We want to argue that with probability at least $1 - \exp(-\Theta(n^{(2/3)^i \cdot 1/6}))$, this one subproblem will have size at most $\eta' = n^{(2/3)^{i+1}}$. Consider the pivot that defines the beginning of the interval given to this subproblem, and then η' elements after that pivotr. The probability that none of these η' elements is chosen as a pivot is at most $(1 - \frac{\eta'}{\eta})^{\sqrt{\eta}} \leq \exp(-\eta'/\sqrt{\eta})$. Noting that $\eta'/\sqrt{\eta} = \Theta(n^{(2/3)^i \cdot 1/6})$, we conclude that with probability at

least $1-\exp(-\Theta(n^{(2/3)^i \cdot 1/6}))$, the number of elements between the two pivots of the subproblem that we are focusing on is at most $\eta' = n^{(2/3)^{i+1}}$. That is, the probability of failing in class i is at most $\exp(-\Theta(n^{(2/3)^i \cdot 1/6}))$. □

Lemma 6.6. *Define* $d_i = (1.49)^i$. *So long as* $n^{(2/3)^i} = \Omega(\log n)$, *the probability of having at least* d_i *consecutive failures at class* i *before moving to class* $i+1$ *is at most* $1/n^3$.

Proof. Left as exercise. □

Notice that one step of recursion where we are at a class i has depth $O\left(\log(n^{(2/3)^i})\right)$. This is in terms of the number of dependent computational steps. Hence, we can bound the total depth to be $O(\log n)$, with high probability, as follows:

Lemma 6.7. *With probability* $1 - 1/n^2$, *the depth of each branch of recursion is* $O(\log n)$.

Proof. Consider one branch of recursion. With probability $1 - 1/n^3$, we have at most $d_i = (1.49)^i$ subproblems along this branch that are in class i. Each of these has a computation with $O\left(\log(n^{(2/3)^i})\right)$ depth. Thus, overall, the computation depth of this branch is $\sum_{i=1} O\left(\log(n^{(2/3)^i})\right) \cdot (1.49)^i = \sum_{i=1} \log n \cdot (1.49/1.5)^i = O(\log n)$. A union bound over all the at most n branches of the recursion shows that with probability $1-1/n^2$, all branches have depth at most $O(\log n)$. □

Lemma 6.8. *With probability* $1 - 1/n^2$, *the total work over all branches is* $O(n \log n)$.

Proof. We can charge the work in each step of recursion to the elements in that step in a way that when dealing with a problem of size x, each element gets charged $O(\log x)$ work. Notice that this is similar to the depth of computation, in that step. Hence, the total charged work for each element is asymptotically upper bounded by the depth of the corresponding recursion branch. Hence, from Lemma 6.7, we can conclude that the total work over all branches is $O(n \log n)$, with high probability. □

Exercise 6.9. Selection

Devise a randomized parallel algorithm that given an array of length n, *finds the* k^{th} *smallest elements of this array, with probability at least* $1 - 1/n^5$. *Ideally, your algorithm should have* $O(\log n)$ *depth and* $O(n)$ *work.*

Exercise 6.10. Permutation
Devise a randomized parallel algorithm that given an array $A[1..n] =< 1, 2, \ldots, n >$*, it produces a random permutation of the elements and writes them in some array* $A'[1..n]$*. Your algorithm should terminate with* $O(\log n)$ *depth and* $O(n)$ *work, with probability at least* $1 - 1/n^5$*.*

Exercise 6.11. Sampling without replacement
Devise a randomized parallel algorithm that given an array of length n*, picks exactly* k *elements of the array uniformly at random, without replacement. Your algorithm should terminate with* $O(\log n)$ *depth and* $O(n)$ *work, with probability at least* $1 - 1/n^5$*.*

6.5 Connected Components

Next, we discuss an algorithm that given an undirected graph $G = (V, E)$, with $V = \{1, 2, \ldots, n\}$ and $m = |E|$, it identifies the connected components of G. As for the the input format, we assume that the graph G is provided as a set of adjacency linked lists $L(v)$, one for each vertex $v \in V$. The output is a component identifier for each component; i.e., for each vertex $v \in V$, we will output an identifier $D(v)$ such that $D(v) = D(u)$ if and only if u and v are in the same connected component of G.

Exercise 6.12. adjacency linked lists to adjacency lists
Explain how—using the algorithms that we have discussed in the previous sections—we can transform the adjacency linked lists $L(v), \forall v \in V$*, to adjacency lists stored as arrays* $A(v)$*, one for each* $v \in V$*, with length* $\deg(v)$*, which denotes the degree of node* v*. Here, the* i^{th} *entry in the array* $A(v)$ *should be the* i^{th} *neighbor of* v*, in its adjacency linked list. What is the depth and total computational work of your solution?*

Exercise 6.13. adjacency linked lists to array of edges
Explain how—using the algorithms that we have discussed in the previous sections—we can transform the adjacency linked lists $L(v), \forall v \in V$*, to one array* $AE[1..m]$ *of edges where each* $AE[i]$ *indicates one of the edges* $\{v, u\} \in E$*. What is the depth and total computational work of your solution?*

Exercise 6.14. array of edges to adjacency linked lists
Explain how—using the algorithms that we have discussed in the previ-

ous sections—we can transform an array $AE[1..m]$ *of edges where each* $AE[i]$ *indicates one of the edges* $\{v, u\} \in E$ *to the adjacency linked lists* $L(v)$*, one for each* $v \in V$*. What is the depth and total computational work of your solution?*

Remark 6.2. *Notice that isolated vertices — i.e., those that do not have any edge — can be identified using* $O(1)$ *depth and* $O(n)$ *work. Moreover, we can remove them from the set* V*, by renumbering* V *and then* E*, using* $O(\log n)$ *depth and* $O(m)$ *work, by an application of parallel prefix (how?). Hence, for the rest of this section, we focus on graphs where* G *has no isolated vertices. Thus, we can assume* $m \geq n/2$*, which allows us to write* $O(m)$ *instead of* $O(m + n)$*, when discussing the amount of work in certain steps.*

Roadmap: Throughout the rest of this section, we work with the ARBITRARY CRCW variant of the PRAM model, which allows concurrent reads and concurrent writes. In the case of multiple simultaneous writes to the same register, we assume that an arbitrarily chosen one of these writes takes effect. In this section, we first see a deterministic algorithm that solves the connected components problem, with $O(\log^2 n)$ depth and $O(m \log^2 n)$ work. We also then comment on how a more involved variant of that approach leads to $O(\log n)$ depth and using $O(m \log n)$ work. Instead of covering that deterministic algorithm, we discuss a much simpler randomized algorithm that leads to the same $O(\log n)$ depth and $O(m \log n)$ work bounds, with high probability.

6.5.1 Basic Deterministic Algorithm

Algorithm Outline: The algorithm is made of $\log n$ iterations. At the beginning of each iteration i, all nodes are in (vertex-disjoint) fragments F_1, F_2, ..., F_{n_i}. For instance, at the beginning of the first iteration, we have $n_1 = n$ fragments, each being simply one of the nodes of the graph. Throughout the iterations, we gradually merge more and more of these fragments with each other, forming new fragments, while maintaining the property that, at all times, all nodes of each fragment F_j belong to the same connected component of G.

Maintaining Fragments as Stars: Moreover, each F_j is maintained as a *star* with one root node and a number of leaf nodes. Each node v has a pointer $D(v)$. For each root node r, this pointer is a self-loop and we have $D(r) = r$. For each leaf node v, the pointer $D(v)$ is equal to the root node r of the corresponding star fragment. We will refer to this common value $D(v)$, which is the identifier of the root node, as the identifier of the fragment. In the beginning of the first iteration, where each fragment is simply a singleton node, each node is the root of it fragment and has a self-loop pointer to itself. A key property of these star shape structures is that, given any two nodes $v, u \in V$, it is easy to check whether they are in the same fragment or not: we just need to test whether $D(v) = D(u)$.

Merge Proposals: During each iteration, we merge some of the fragments which are in the same connected component with each other. For each fragment F_j rooted in a node r_j, we would like to identify the minimum root node r_k such that there is an edge $e = (v, u) \in E$ from some node v with $D(v) = r_j$ to some node u with $D(u) = r_k$. In such a case, we say that the fragment F_j *proposes* to merge with fragment F_k. Moreover, we remember this as $p(r_j) = r_k$. If for a fragment F_j there is no such edge e connecting it to other fragments, then we remember a self-loop as its proposal, i.e., we set $p(r_j) = r_j$. We can compute all of these minimums simultaneously, one per fragment, in $O(\log n)$ depth and $O(m)$ work. We next phrase this as an exercise, broken into small concrete steps.

Exercise 6.15. Minimum neighbor per fragment
Explain how we can identify the minimum neighbors as desired above, simultaneously for all the fragments, using $O(\log n)$ depth and $O(m)$ work. You can do this in three steps, as outlined below:

(A) First, create an array for the edges incident on the nodes of each fragment, in a manner that the sum of the lengths of these arrays over all fragments is at most $2m$. This should be doable using $O(\log n)$ depth and $O(m + n \log n)$ work (how? hint: think about the idea in Exercise 6.8).

(B) Then, explain how to discard the entries of the array of each fragment that do not connect to other fragments. Moreover, for entries that are not discarded, add to them the identifier of the

root node of the other endpoint's fragment. These two operations should be doable using $O(1)$ *depth and* $O(m)$ *work.*

(C) Finally, explain how we can compute the minimum among the entries that are not discarded, simultaneously for all fragments, using $O(\log n)$ *depth and* $O(m)$ *work.*

Now, the set of proposed merge edges, declared by pointers $p(r_j)$, one for each root r_j, determine a *pseudo-forest* $\mathcal{P}$ among the nodes of the graph. In a *pseudo-forest*, each connected component is a *pseudo-tree*, which is simply a tree plus one extra edge that creates one cycle. In our particular case, we have even more structure: the arcs are oriented and each node has out-degree exactly one. This implies that each connected component is a (directed) tree plus one cycle-creating arc. Given that we defined the proposed arcs to be those to the minimum neighboring fragment, we get some additional nice property about the cycles:

Lemma 6.9. *Each cycle in* $\mathcal{P}$ *is either a self-loop or it contains exactly two arcs.*

Proof. Left as an exercise, with the hint that each fragment F_j proposes the neighboring fragment F_k with the minimum identifier. Thus, we cannot have a cycle of length 3 or higher in the proposals (why?). □

Transforming Pseudo-Trees to Stars: The pseudo-forest obtained with the merge edges is great in that it has all of its vertices from the same connected component of G. However, it does not have the nice star-shape that we assumed, which facilitated our job for figuring out which edges are going to other fragments. To prepare for the next iteration, we would like to transform each pseudo-tree in $\mathcal{P}$ into a star shape, where all nodes of each fragment point to the root of that fragment and the root points to itself.

For each root node r, remove its self-loop in its fragment and instead set $D(r) = p(r)$, i.e., the root of the proposed fragment with which the fragment of r wants to merge. Then, we do $\log n$ repetitions of *pointer-jumping*, similar to what we saw in Section 6.2.3. In particular, for $\log n$ repetitions, for each node v, set $D(v) = D(D(v))$. It can be seen that after these, the pointer $D(v)$ of each node is to one of the at most two nodes in the cycle of $\mathcal{P}$ containing v (why?). As a final step, set $D(v) = \min\{D(v), p(D(v))\}$, so that all nodes of the same component of $\mathcal{P}$ point to

the same root node. This provides us the desired star shapes. In particular, again each fragment (which is basically a shrunk version of the a component of $\mathcal{P}$) is a star-shape where all nodes point to one root node.

This part of transforming pseudo-trees to star shapes is basically $O(\log n)$ pointer jumping iterations — plus one final step of taking the minimum of two pointers — and thus it uses $O(\log n)$ depth and $O(n \log n)$ work.

Analysis: First, we argue that $\log n$ iterations suffice to reach a setting where each fragment is exactly one of the connected components of the graph G—more formally, the nodes of the fragment are the nodes of one connected component of G. It is clear that we can never merge nodes from two different connected component of G. Let us focus on one connected component $\mathcal{C}$ of G. During each iteration, so long as the nodes of $\mathcal{C}$ are in two or more fragments, each of the fragments of $\mathcal{C}$ merges with at least one other fragment of $\mathcal{C}$. Hence, the number of fragments shrinks by a 2 factor. Thus, after at most $\log n$ iterations, $\mathcal{C}$ has exactly one fragment.

As for the complexity analysis, each iteration uses $O(\log n)$ depth — mainly to find the proposal edges and also to shrink pseudo-trees to stars — and $O(m \log n)$ work. Hence, over all the $\log n$ iterations, we have $O(\log^2 n)$ depth and $O(m \log^2 n)$ work. There is a (deterministic) algorithm that follows the same general approach, but uses only $O(1)$ repetitions of pointer jumping per iteration, and achieves $O(\log n)$ depth and $O(m \log n)$ work. See [JáJ92, Section 5.1.3]. Though, that algorithm requires more elaborate merge structures, in terms of what merges are allowed at each time step and why these restricted merges suffice for reaching connected components. Next, in Section 6.5.2, we see a much simpler *randomized* algorithm that achieves $O(\log n)$ depth and $O(m \log n)$ work.

Exercise 6.16. Minimum as Identifier

Prove that at the end of the above algorithm, each component will have its identifier equal to the minimum identifier among the nodes of that component.

Exercise 6.17. Maximal Spanning Forest

Modify the above algorithm so that it outputs a maximal spanning forest of the graph G*. In particular, for each connected component* C *of* G*, you should output a tree* $T \subseteq G$ *that spans all vertices of* C*. The output format for* T *should be adjacency linked lists.*

6.5.2 Randomized Algorithm

We now discuss how to overcome the need for the time-consuming fragment shrinkage steps. Recall that previously, in each iteration of fragment merging, we used $O(\log n)$ repetitions of pointer jumping to reduce the height of each fragment and turn it into a star. The need for these $O(\log n)$ iterations comes from the possibility that the edges that we choose for a merge in one iteration can form long chains. We next explain how to overcome this, using a simple randomization idea. Before that, let us relax how we computed the proposal edges.

Proposed Merge Edges: In the algorithm discussed above, we computed for each fragment F_j the minimum identifier neighboring fragment F_k, and as a result, we recorded $p(r_j) = r_k$, as a proposed edge for a merge. For the new algorithm, we will not need to rely on this minimum, and we can do with just any neighboring fragment. For each edge $(v, u) \in E$ such that $D(v) \neq D(u)$, write $p(D(v)) = D(u)$. This is done in $O(1)$ depth and $O(m)$ work, in the ARBITRARY CRCW version of the PRAM model. Notice that $O(m)$ writes are trying to take place at the same time, many of them competing for the same register $p(D(v))$. We will not assume anything particular about which one succeeds; we just use the ARBITRARY CRCW version which guarantees that one of these writes will take effect for each register and thus, there will be some proposed edge for each fragment that is not alone in its component. Of course, this still does not overcome the challenge that these proposal pointers can form long chains (or more formally, deep psuedo-trees) and thus shrinking them to stars would require many repetitions of pointer jumping.

Randomization to Overcome Long Chains: Faced with this chaining issue, we can call a simple randomized idea to the rescue. It is worth noting that this idea is similar in spirit to what we saw in Section 6.3.1. For each fragment, we toss a HEAD/TAIL coin. We then *accept* a *proposed* merge edge only if it points from a TAIL fragment to a HEAD fragment. The great positive of this choice is that, now, only HEAD fragments can have others merge it, and the depth of the resulting merge is $O(1)$ (why?). Hence, only $O(1)$ iterations of pointer jumping suffice to turn each new fragment into a star shape. The potential negative is that, we did not use *all* of the proposed

merge edges. We next argue that still $O(\log n)$ iterations of merging should suffice, with high probability.

Lemma 6.10. *In each iteration i where $n_i \geq 2$, the expected number n_{i+1} of new fragments is at most $7/8$ of the number n_i of the old fragments.*

Proof. Suppose that $n_i \geq 2$. Each fragment will have one proposed edge. Each proposed edge gets accepted with probability at least $1/4$ (why?). Hence, we expect at least $n_i/4$ accepted proposed merges. Noting that an edge might be proposed from both endpoint, we conclude that the number of fragments should reduce by at least $n_i/8$, in expectation. Hence, $\mathbb{E}[n_{i+1}] \leq \frac{7}{8} \cdot n_i$. □

Lemma 6.11. *After $L = 20 \log n$ iterations, the number of fragments $n_L = 1$, with probability at least $1 - 1/n^2$.*

Proof. Define $z_i = n_i - 1$ to be the excess number of fragments in one component. We first show that $\mathbb{E}[z_L] \leq 1/n^2$. With a slight adjustment to the argument of Lemma 6.10, we can write $\mathbb{E}[z_{i+1}] \leq z_i - \frac{1}{8} \cdot n_i \leq \frac{7}{8} \cdot z_i$, where the expectation is over the random events of iteration i. Hence, considering the random events in two consecutive iterations i and $i+1$ (and given that the random events of different iterations are independent), we can write

$$\begin{aligned}
\mathbb{E}[z_{i+2}] &= \sum_{\zeta} \mathbb{E}[z_{i+2} | z_{i+1} = \zeta] \cdot \Pr[z_{i+1} = \zeta] \\
&\leq \sum_{\zeta} \frac{7}{8} \cdot \zeta \cdot \Pr[z_{i+1} = \zeta] = \frac{7}{8} \mathbb{E}[z_{i+1}] \\
&\leq (7/8)^2 z_i.
\end{aligned}$$

Repeating the same kind of argument over different iterations of the algorithm, we get that $\mathbb{E}[z_L] \leq (7/8)^{L-1} z_1 \leq \frac{1}{n^3}(n-1) \leq 1/n^2$. Now, since $\mathbb{E}[z_L] \leq 1/n^2$, by Markov's inequality, we get that $\Pr[z_L \geq 1] \leq 1/n^2$. Since z_L must be an integer, that means, with probability at least $1 - 1/n^2$, we have $z_L = 0$ and thus $n_L = 1$. □

6.6 (Bipartite) Perfect Matching

In this section, we discuss parallel algorithms for the problem of computing a perfect matching. For the sake of simplicity, we limit the discussion to

bipartite graphs. Before starting the discussion about perfect matching, let us discuss a bigger picture view of the parallel algorithms that we have seen so far, and how what we are about to see compares with them.

6.6.1 Discussions and a Bigger Picture View

A Bigger Picture View: In the previous subsections, we discussed parallel algorithms for several problems. In all of these, the target was to obtain an algorithm with a small depth —e.g., $O(\log n)$ — which is also *work-efficient*, meaning that its total computational work is asymptotically the same (or close to) the sequential variant, e.g., $O(n \log n)$ work for sorting. While such low-depth work-efficient algorithms are the ultimate goal for all problems, for many problems, the current state of the art is far from that. For these harder problems, the primary objective is to first obtain a small depth algorithm —i.e., $O(\log^k n)$ for as small as possible constant $k > 0$ — while allowing the total work to be any polynomial in n. Recall from Section 6.2 that this exactly corresponds to the circuit complexity of the problem, e.g., $NC(k)$ is the class of decision problems on n-bit inputs that are solvable using circuits of depth $O(\log^k n)$ and size $\text{poly}(n)$. A natural generalization is to randomized algorithms (and randomized circuits), which have access to random bits. We say a decision problem is in $RNC(k')$ if there is a randomized circuit with depth $O(\log^k n)$ and size $\text{poly}(n)$ which outputs the correct solution with probability at least $2/3$. Notice that this probability can be easily amplified to any other desirable value, by repetition (how?). Again, we define $RNC = \cup_k RNC(k)$. Clearly, we have $NC \subset RNC$. A fundamental and important question in the *theory of computation*, and particularly circuit complexity and the theory of parallel computation, is whether $RNC = NC$. That is,

> *does every decision problem that admits a poly-logarithmic depth polynomial work* randomized *parallel algorithm also admit a poly-logarithmic depth polynomial work* deterministic *parallel algorithm?*

More generally, we can also consider search problems. Again, our goal is to find parallel algorithms with depth $\text{poly}(\log n)$ depth and $\text{poly}(n)$ work. This is what we will consider as an efficient parallel algorithm.

Finding a perfect matching problem is a concrete instance for which we have known efficient randomized parallel algorithms for a long while,

since the marvelous work of [MVV87], and we still do not have an efficient deterministic parallel algorithm for it[4]. However, recent years witnessed breakthrough progress on this problems [FGT16, ST17], leading to a poly-logarithmic depth deterministic algorithm with a quasi-polynomial amount of work, i.e., $n^{\log^{O(1)} n}$. At the moment, it seems quite reasonable to hope that we will see an efficient deterministic parallel algorithm for perfect matching in the near future.

In this section, we discuss developements on this topic. We start with an efficient randomized parallel algorithm for computing a perfect matching in bipartite graphs; this is a result from the 1980s. Then, we discuss the recent results that provide a deterministic parallel algorithm that almost achieves a similar efficiency— it also has $\text{poly}(\log n)$ depth but it uses quasi-polynomial work, $n^{O(\log n)}$ to be concrete.

Before discussing matching, let us remark about parallel algorithms for matrix computations.

Remark 6.3. *There is a randomized parallel algorithm that, using $O(\log^2 n)$ depth and $\tilde{O}(n^{3.5}k)$ computation, can compute the determinant, inverse, and adjoint of any $n \times n$ matrix with k-bit integer entries. Recall that the adjoint of a matrix A is an $n \times n$ matrix whose entry (i, j) is $(-1)^{i+j}\det(A_{ji})$, where A_{ji} denotes the submatrix of A resulted from removing its j^{th} row and i^{th} column.*

Comment: The above remark states the best known bounds for matrix operations, and for this chapter, you do not need to know how that algorithm works. In fact, to prepare the discussions of this section, it suffices that you know that we can perform the above computation using poly-logarithmic depth and polynomial work. For this more relaxed goal, the algorithms are much simpler. In particular, think about how you would compute the multiplication of two $n \times n$ matrices, using $O(n^3)$ work and $O(\log n)$ depth. Inverse, determinant, and adjoin can be computed based on this parallel building block of matrix multiplication, when added to other classic algorithms.

[4]Formally, this is not exactly about RNC vs NC, as we are talking about a search problem—finding a perfect matching—and not a decision problem.

6.6.2 Randomized Parallel Algorithm

The Bipartite Perfect Matching Problem: Let $G = (V, U, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $U = \{u_1, u_2, \ldots, u_n\}$ and $E \subseteq V \times U$, be our input bipartite graph. Our goal is to compute a perfect matching M, which is a set of n edges no two of which share an endpoint, and thus we have exactly edge in M incident on each node of the graph.

Intuitive Discussions Recall from Section 5.3 that the question of whether G has a perfect matching or not can be solved by computing the determinant of a certain $n \times n$ matrix (what matrix?). Thanks to the remark above, computing the determinant can be done with polylogarithmic depth and polynomial work. But we are interested in one step further: we want to compute a perfect matching.

Coming from the viewpoint of centralized algorithms, a natural idea for computing a perfect matching would be as follows: remove each edge e, and see if the resulting graph still has a perfect matching. If not, that edge belongs to the perfect matching. If yes, we can discard that edge without any concern and just search for a perfect matching without it. One could think that we can parallelize this idea by checking different edges in parallel. However, this will not work, at least not in general. If the graph has many perfect matching, machines checking edge e_1 could decide to drop it, because $G \setminus \{e_1\}$ has a perfect matching, and machines checking edge e_2 could decide to drop it because $G \setminus \{e_2\}$ has a perfect matching, but actually each perfect matching has one of these two and thus, now that we have decided to drop both, we lost the perfect matching. Of course, the situation can be far more complex, with many edges having such a situation. But there is still some scenario in which this idea would work: if the graph has a unique perfect matching! (why?) Our goal would be to try to mimic such a setting in general graphs. In particular, we will use weights on the edges so that the minimum-weight perfect matching is unique. Then, finding that unique matching will be doable, very much similar to computing the determinant as we discussed above.

To materialize this idea, we first discuss the *isolation lemma*, which is a beautiful abstract result about how the minimum of set weight behaves under random weight assignment to elements. Then, we use this isolation lemma to obtain our randomized parallel perfect matching algorithm.

Lemma 6.12. *(Isolation Lemma) Let* (E, F) *consist of a finite set* E *of elements, with* $m = |E|$*, and a family* F *of subsets of* E*, i.e.,* $F = \{S_1, S_2, \ldots, S_N\}$*, where* $S_i \subseteq E$*,* $\forall 1 \leq i \leq N$*. Assign to each element* $e \in E$ *a random weight* $w(e)$ *uniformly and independently chosen from* $\{1, 2, \ldots, 2m\}$*. Define the weight of any set* $S_i \in F$ *to be* $\sum_{e \in S_i} w(e)$*. Then, regardless of how large* N *is, with probability at least* $1/2$*, there is a unique minimum weight set in* F*.*

Before going to the proof, let us discuss an intuitive aspect of the lemma. In some sense, the lemma might be seen as surprising (although the proof is really simple and elementary). The reason is that N could be as large as 2^m, while the weight of each set is certainly in the range $[1, (2m)^2]$. Hence, naively thinking, we could expect to have $\frac{2^m}{(2m)^2} \gg 1$ many sets have each particular weight value. Something that suggest it is highly unlikely to have a unique set with a given weight. However, thanks to the structure of how sets in a ground set E can relate, this is not the case for the minimum-weight set. We will see that with a constant probability, the minimum weight set is unique!

Proof of Lemma 6.12. Let us focus on elements $E' \subseteq E$ that appear in at least one set of F. Let us call an element $e \in E'$ *ambiguous* if there is a minimum-weight set in F that contains e and another minimum-weight set in F that does not contain e. We will show that, with probability at least $1/2$, there is no ambiguous element. Thus, with probability at least $1/2$, the minimum-weight set is unique. Let us now focus on one element $e \in E'$. Suppose that we have fixed the random weight of all elements besides e (in fact, arbitrarily, from the range $\{1, \ldots, 2m\}$). Let W be the weight of a minimum weight set containing e, according to the weights of all elements besides e (i.e., ignoring the weight of e which is not determined yet). Similarly, let $\overline{W}$ be the weight of a minimum weight set that does not contain e. Let $\alpha = \overline{W} - W$. Imagine increasing $w(e)$ from $-\infty$ to ∞, or more realistically, just from $-2m^2$ to $2m^2$. When $w(e)$ is very small $(-\infty)$, then every set of minimum weight must contain e, and when $w(e)$ is very large (∞), then every set of minimum weight must exclude e. The transition happens in exactly one value. More concretely, whenever $w(e) < \alpha$, every minimum weight set must contain e because there is a set containing e that has weight $W + w(e) < \overline{W}$, i.e., less than the minimum weight set among those that do not contain e. Similarly, whenever $w(e) >$

α, every minimum weight set must exclude e because there is a set that excludes e and has weight $\overline{W} < W + w(e)$, i.e., less than the minimum weight set among those that contain e. Thus, if $w(e) \neq \alpha$, then e is not ambiguous. Now e is chosen randomly from a range $\{1, 2, \ldots, 2m\}$, and independent of all other elements. Hence, $\Pr[w(e) = \alpha] \leq 1/(2m)$. Thus is, the probability that e is ambiguous is at most $1/(2m)$. Now, by a union bound over all the at most m elements of E', we get that the probability that there is at least one ambiguous element is at most $m \times 1/(2m) = 1/2$. That is, with probability at least $1/2$, there is no ambiguous element. That means, with probability at least $1/2$, the minimum weight set is unique. □

A Bipartite Perfect Matching Algorithm Given the isolation lemma, and the intuitive discussion above, the algorithm should now be clear: we set a random weight $w(e)$ for each edge $e \in E$ uniformly and independently chosen from $\{1, 2, \ldots, 2m\}$. Then, the isolation lemma tells us that, no matter how many perfect matchings there are in the graph, with probability at least $1/2$, the minimum weight perfect matching is unique. Now, we just need to find the minimum weight perfect matching, which we can do as follows.

- Let A be the $n \times n$ matrix where $a_{ij} = 2^{w(e_{ij})}$ if there is an edge $e_{ij} = (u_i, v_j)$, and $a_{ij} = 0$ if there is no such edge.
- Compute $\det(A)$ using the parallel algorithm of Remark 6.3 and let w be the highest power of 2 that divides $\det(A)$.
- Compute $\mathrm{adj}(A)$ using the parallel algorithm of Remark 6.3, where the (i, j) entry will be equal to $(-1)^{i+j}\det(A_{ji})$. Here, A_{ji} denotes the submatrix of matrix A resulted from removing the j^{th} row and the i^{th} column of A.
- For each edge $e_{ij} = (u_i, v_j)$, do in parallel:
  - Compute $\frac{\det(A_{ij})2^{w(e_{ij})}}{2^w}$. If it is odd, include $e_{ij} = (u_i, v_j)$ in the output matching.

In the following two lemmas, we explain that if the random weights are chosen such that the minimum-weight perfect matching is unique, then the above procedure will output it (with a deterministic guarantee). To amplify

the success probability of the isolation lemma, we can run it several times in parallel; then we output a perfect matching if any of the runs found a perfect matching. Since each run has success probability at least $1/2$, running it $10 \log n$ times ensures that we find a perfect matching with probability at least $1 - (1/2)^{10 \log n} = 1 - 1/n^{10}$. We now discuss the two key lemmas, which explain why the above procedure identifies the unique minimum-weight perfect matching.

Lemma 6.13. *Suppose that $G = (V, U, E)$ has a unique minimum weight perfect matching M and its weight is w. Then, the highest power of 2 that divides $\det(A)$ is 2^w.*

Proof. Recall from Section 5.3 that each perfect matching in G corresponds to one permutation $\sigma \in \mathcal{S}_n$. For a given permutation, define its value to be $value(\sigma) = \prod_{i=1}^{n} a_{i\sigma(i)}$. We have $\det(A) = \sum_{\sigma \in \mathcal{S}_n} sign(\sigma) \cdot value(\sigma)$. Here $sign(\sigma)$ is $+1$ for even permutations and -1 for odd permutations (though for this proof, the sign is not relevant). Recalling that M is the unique minimum weight perfect matching of G, the value of the permutation σ_M corresponding to M is

$$value(\sigma_M) = \prod_{i=1}^{n} a_{i\sigma(i)} = \prod_{i=1}^{n} 2^{w(e_{i\sigma(i)})} = 2^{\sum_{i=1}^{n} w(e_{i\sigma(i)})} = 2^w.$$

For any other permutation σ', the value is either 0 or a strictly higher power of 2. Hence, $\det(A)$ which is simply a summation of them (with signs) is divisible by 2^w but no higher power $2^{w'}$ for $w' \geq w + 1$. Notice that the higher powers might cancel each other, due to the ± 1 signs, but they cannot cancel 2^w, as it is a strictly lower power of 2. □

Lemma 6.14. *Suppose that $G = (V, U, E)$ has a unique minimum weight perfect matching M and its weight is w. Then, edge $e_{ij} = (u_i, v_j)$ is in M if and only if $\frac{\det(A_{ij}) 2^{w(e_{ij})}}{2^w}$ is odd.*

Proof. Notice that

$$|\det(A_{ij}) 2^{w(e_{ij})}| = |\sum_{\sigma \in \mathcal{S}_n : \sigma(i) = j} sign(\sigma) \cdot value(\sigma)|.$$

Let σ_M be the permutation corresponding to M. If $e_{ij} = (u_i, v_j) \in M$, then the term corresponding to σ_M in the above summation is 2^w, while

all the other terms are either zero or strictly higher powers of 2 than 2^w. That means, $\frac{\det(A_{ij})2^{w(e_{ij})}}{2^w}$ is odd. If $e_{ij} = (u_i, v_j) \notin M$, then all terms in the summation, which correspond to all potential perfect matchings that include e_{ij}, are either zero or strictly higher powers of 2 than 2^w. Thus, $\frac{\det(A_{ij})2^{w(e_{ij})}}{2^w}$ is even. □

6.6.3 Deterministic Parallel Algorithm

In the algorithm discussed in the previous subsection, the only place where we used randomness was the Isolation Lemma. We used random edge weights so that, with a high probability, there is exactly one minimum-weight perfect matching. To obtain a deterministic algorithm, we would like to replace these random edge weights with deterministic edge weights. This remained an open question for decades until a recent 2016 breakthrough of Fenner, Gurjar, and Thierauf [FGT16, FGT19], which almost resolves the problem, though using quasi-polynomial amount of work. This is what we cover in this section.

Concretely, we will define a family $\mathcal{W}$ of $n^{O(\log n)}$ weight functions $w : E \to \mathbb{R}_0^+$, in each of which each edge has a positive integer weight that is at most $n^{O(\log n)}$, with the following property: for any bipartite graph, there is at least one weight function $w \in \mathcal{W}$ for the edges such that there is exactly one minimum-weight perfect matching. We call $\mathcal{W}$ an *Isolating Family of Weight Functions*. Then, via checking all the weight functions in $\mathcal{W}$ in parallel, and trying to obtain a perfect matching using the scheme discussed in the previous subsection for each weight function, we can compute a perfect matching in parallel. Overall, the algorithm uses $n^{O(\log n)}$ work and $\text{poly}(\log n)$ depth.

The rest of this section is dedicated to developing such an Isolating Family of Weight Functions, as captured by the following statement.

Theorem 6.15. *consider two n-node sets of vertices U and V and all possible bipartite graphs between them. There is a family $\mathcal{W}$ of $n^{O(\log n)}$ weight functions $f : ([U] \times [V]) \to \mathbb{R}_0^+$, in each of which each possible edge $e \in [U] \times [V]$ has a positive integer weight $w(e) \leq n^{O(\log n)}$, with the following property: for any bipartite graph $G = (U, V, E)$, there is at least one weight function $w \in \mathcal{W}$ for the edges such that if we set*

the weight of the edges E *according to* $w(e)$*, then graph* G *has exactly one minimum-weight perfect matching.*

Intuitive Discussion, and Circulations We will build the family of weight functions gradually, for that a key concept will be *circulation* in each cycle. Let us discuss these in an intuitive manner:

Consider a weighted bipartite graph, where $w(v,u)$ denotes the weight of the edge between nodes v and u. Consider two perfect matchings M_1 and M_2 and their disjoint union $M_1 \sqcup M_2$, that is, edges that appear in exactly one of M_1 and M_2 (and not both). Observe that $M_1 \sqcup M_2$ is simply a number of cycles, each of even length (as we are in a bipartite graph), where per cycle the edges alternate between those of M_1 and those of M_2. Define the *circulation* of a cycle to the absolute difference of the summation of its odd edges and the summation of its even edges. That is, for a cycle $C = \left(v_1, v_2, v_3, \dots, v_k\right)$, define its circulation

$$\mathrm{circ}_w(C) = |w(v_1,v_2) - w(v_2,v_3) + w(v_3,v_4) - \cdots + w(v_{k-1},v_k) - w(v_k,v_1)|.$$

Notice that this definition is independent of the matching that we are considering, and also independent of what we viewed as the starting point of the cycle. If all the cycles in $M_1 \sqcup M_2$ have zero circulation, then M_1 and M_2 have the same weight. Interestingly, we have a converse of this if both matchings are min-weight matching: If M_1 and M_2 are two min-weight perfect matchings, then in their disjoint union $M_1 \sqcup M_2$, every cycle must have zero circulation. We next state a more general claim, which talks about the union of all min-weight perfect matchings. This will be a key tool in devising the Isolating Family (the proof provided here departs from that of Fenner et al. [FGT16, FGT19], and is instead based on an observation of Anup Rao, Amir Shpilka, and Avi Wigderson [GG17]):

Lemma 6.16. *Let* $G = (V,E)$ *be an weighted bipartite graph. Let* $G' = (V,E')$ *be the subgraph where* E' *is the set of all edges* $e \in E$ *that are in at least one minimum weight perfect matching. Then, for each cycle* C *in* G'*, we have* $\mathrm{circ}_w(C) = 0$*. Said differently, any cycle* C' *in* G *that has* $\mathrm{circ}_w(C') \neq 0$ *cannot be fully in* G' *and at least one of its edges is not in* E'*.*

Proof. Suppose that $G = (V,E)$ has d min-weight perfect matchings. Also, let Z be the weight of each minimum-weight perfect matching. Let H be the

multigraph obtained by including all of these d many min-weight perfect matchings: if an edge $e \in E$ appears in say k different min-weight perfect matchings of G, there are k copies of e in the multigraph H. Notice that H is a bipartite and d-regular graph, i.e., each node has degree d.

Suppose that there is a cycle C' that has nonzero circulation in G'. This cycle appears also in H. Then, we can alter H to remain d-regular but with a smaller total weight. E.g., if in the cycle C', odd number edges have a strictly larger weight than even numbered edges, we then remove one copy of the odd numbered edges of C' from H and instead add one copy of the even numbered edges of C' to H. Graph H remains d-regular but it has weight strictly less than dZ. Graph H can be decomposed in d disjoint perfect matchings (why? hint: repeatedly apply Hall's condition). Hence, there is a perfect matching in H that has weight strictly smaller than Z. But the same matching is present in G (as edges of H are just copies of edges of G), which means G has a perfect matching of weight strictly smaller than Z. This contradicts the definition of Z. Hence, our initial assumption must have been wrong and cycle C' that has nonzero circulation cannot be present in G'. □

Intuitive Plan Because of Lemma 6.16, to achieve Theorem 6.15, our plan of attack is to devise a weight function so that cycles have non-zero circulation. For a small collection of cycles, we can do this directly: We next discuss in Lemma 6.17 how to define a collection of polynomially many weight functions so that for a small collection—say $\text{poly}(n)$ many—of cycles, at least one of the weight functions ensures that none of these cycles has zero circulation. However, unfortunately, a graph has exponentially many cycles and that complicates the task of designing the weight function family. To handle that, we will handle the cycles in $\log n$ iterations, where in iteration i we target cycles of length 2^i. We will see in Lemma 6.18 that, per iteration, we will have to target only $\text{poly}(n)$ cycles (because of the weights of the previous iterations that handled smaller length cycles). This will enable us to apply Lemma 6.17 per iteration.

Lemma 6.17. *Consider a graph and collection $\mathcal{C}$ of s many cycles in it. There is collection of $O(n^2 s)$ weight functions for the edges, where each edge gets a weight in $\{0, 2, \ldots, O(n^2 s)\}$, such that for at least one of the weight functions, none of the cycles in $\mathcal{C}$ has zero circulation.*

Proof. Let $E = \{e_1, \ldots, e_m\}$ be the set of the edges of the graph. Consider an initial weight function $w(e_i) = 2^{i-1}$ for all $i \in \{1, 2, \ldots, m\}$. These weights give non-zero circulation in each cycle (why?) but unfortunately these are exponentially large weights. We want the weights to be polynomially large.

Let $t = n^2 s$. Define the family $\mathcal{F}$ of weight functions $f : E \to \{0, \ldots, t\}$ where for each integer $j \in [2, t]$, we define $f_j(e_i) = w(e_i) \bmod j$. We show that for any collection $\mathcal{C}$ of s many cycles, there is at least one weight function $f_j \in \mathcal{F}$ that gives nonzero circulation for all cycles of $\mathcal{C}$.

Suppose for the contradiction that this is not the case. Then, for each $j \in [2, t]$, there is at least one cycle in the collection $\mathcal{C}$ that has zero circulation in f_j. Thus, the circulation of that cycle according to initial weights $w(e_i) = 2^{i-1}$ is divisible by j. That would mean, the multiplication $\prod_{k=1}^{s} \mathrm{circ}_w(C_k)$ of the circulations of all cycles $C_k \in \mathcal{C}$—with the initial weights $w(e_i) = 2^{i-1}$— is divisible by each each integer $j \in [2, t]$. This is because for each j there is at least one term in the multiplication that is divisible by j. However, the least common multiple of all integers in $[2, t]$ is strictly greater than 2^t [Nai82]. In contrast, the product $\prod_{k=1}^{s} \mathrm{circ}_w(C_k)$ is at most $2^t = 2^{n^2 s}$ because each term $\mathrm{circ}_w(C_k)$ is at most 2^{n^2}. This is a contradiction.

Having arrived at a contradiction, we conclude that our assumption must have been wrong and thus, there is at least one $j \in [2, t]$ such that f_j gives nonzero circulation to all the cycles in $\mathcal{C}$. □

Devising the overall Isolating Family of Weight Functions We will devise the Isolating Family in iterations, where we successively target longer and longer cycles, in smaller and smaller subgraphs. In the very first iteration, we target cycles of length at most 4 in $G_0 = G$. There are at most n^4 many such cycles. We apply Lemma 6.17 to get $O(n^6)$ weight functions, each with values at most $O(n^6)$, such that at least one function gives non-zero circulation to all cycles of length at most 4.

We then move toward targeting longer cycles. Let us discuss the second iteration. For now, focus on that one good function from iteration one (at the end we will try all in parallel). Let G_1 be the spanning subgraph of G defined by edges that are in a min-weight perfect matching, according to this good weight function. By Lemma 6.16, G_1 cannot have any cycle of length 4. We then target cycles of length at most 8 in G_1. Thanks to the

next lemma, we know there are at most n^4 such cycles.

We repeat this process for $\log n$ iterations. In general, suppose that we have done i iterations and consider the good weight function at the end of the i iterations. Let G_i be the subgraph defined by all edges that are in a min-weight perfect matching in this weight function. Then, we know by Lemma 6.16 that G_i has no cycle of length at most 2^i. By Lemma 6.18, subgraph G_i has at most n^4 cycles of length at most 2^{i+1}. Then, we invoke Lemma 6.17 to devise updated weights so that these cycles of length 2^{i+1} also have nonzero circulations. In general, the weight $w_i + 1$ in iteration i is defined as $w_{i+1} = Nw_i + w'$ where w_i is (any of) the weight function(s) from the previous iteration, w' is (any of) the weight function(s) provided by Lemma 6.17, and N is set equal to $2n \max_e w'$. Notice that $N = O(n^7)$. Because of this large N factor, any cycle that previously had a nonzero circulation will remain with a non-zero circulation regardless of what w' weights we set on its edges: the reason is that even a circulation of ± 1 in w_i is now scaled to a circulation of $\pm N$, and that cannot be canceled by up to n edge additions of each at most $\max_e w'$. Then, we move to the next iteration.

At the end of $\log n$ iterations, we know that the subgraph defined by edges that are in a min-weight perfect matching has no cycle. That means the min-weight perfect matching is unique. In each iteration, we apply any of the $O(n^6)$ weight functions. Hence, overall, this is $n^{O(\log n)}$ different weight functions — the key is that we do not need to find the right one for the graph; the entire family is our Isolating Family. We will just use them all in parallel and one of them will be the good weight function that ensures that there is exactly one min-weight perfect matching. Moreover, the maximum (additional) weight of any edge is $O(n^6)$ per iteration but it gets multiplied by $N = O(n^7)$ as we move from the weights w_i of this iteration to those weights w_{i+1} of the next iteration. Hence, over the $\log n$ iterations, the final values of weight are at most $n^{O(\log n)}$. This gives Theorem 6.15.

What remains from the above outline is to state and prove Lemma 6.18.

Lemma 6.18. *Consider a graph H and suppose that it has no cycle of length r, for some even number $r \geq 4$. Then, the number of cycles of length at most $2r$ is at most n^4.*

Proof. Consider an arbitrary cycle C of length at most $2r$ in H. We associate C with a 4-tuple of nodes: Choose four nodes (u_1, u_2, u_3, u_4) in the

cycle where each two consecutive ones are $r/2$ steps apart in the cycle. We claim that there cannot be any other cycle C' of length at most $2r$ that is also associated with this same 4-tuple (u_1, u_2, u_3, u_4). For contradiction, suppose there is such a cycle C'. Each of these two cycles is made of four paths: Cycle C consists of path P_1 from u_1 to u_2, path P_2 from u_2 to u_3, path P_3 from u_3 to u_4, path P_4 from u_4 to u_1. Cycle C' consists of path P'_1 from u_1 to u_2, path P'_2 from u_2 to u_3, path P'_3 from u_3 to u_4, path P'_4 from u_4 to u_1. Since the two cycles are not the same, for at least some i, we have $P_i \neq P'_i$. But those paths P_i and P'_i would be two distinct paths of length $r/2$ between the same nodes, which would mean that the graph has a cycle of length at most $r/2+r/2 = r$. This is a contradiction. Hence, each 4-tuple of nodes (u_1, u_2, u_3, u_4) has at most one cycle of length at most $2r$ associated with it. There at most n^4 such 4-tuples of nodes. Hence, the number of cycles of length at most $2r$ is also at most n^4. □

This brings us a concrete open problem: the above gives a parallel deterministic algorithm with $n^{O(\log n)}$ work and $\text{poly}(\log n)$ depth for bipartite perfect matching. Can we find such an algorithm with only $n^{O(1)}$ work?

6.7 Modern/Massively Parallel Computation (MPC)

6.7.1 Introduction and Model

Parallelism: Fine-grained versus Coarse-grained: In the previous sections, we discussed parallel algorithms in the classic PRAM model, where multiple processors all have access to a globally shared memory and each step of computation is one RAM operation, or a read/write access to the shared memory. This is in some sense a rather fine-grained view of parallelism. When designing algorithms for PRAM, we are trying to break the computation into very *fine-grained* operations, each being an individual RAM operation—such as adding and multiplying—or a memory access operation to the global shared memory. Moreover, we are somehow implicitly relying on the assumption that these tiny operations will all be done at the very same speed, by different processors across the whole system. If one processor is slightly slower at a certain point, this design means all processors should be slowed down by that much, before proceeding to their next step.

If a processor fails/crashes for one reason or another (something that occurs frequently when you have thousands of processors), then some other processor has to pick up that task and perform it, before we can proceed to the next step. This causes a significant overhead on the whole system, but all because of a fault in a tiny step of computation. Furthermore, we have generically discarded all *communication* issues, e.g., we assumed that all the accesses to the shared memory can be performed in a unit time, regardless of how far the physical location of that shared memory cell is. These issues all suggest that the algorithms that are designed in the PRAM mindset may be somewhat far from the realities of the current world, or the near future. This might partially explain the decay of the research activities in the area of parallel algorithms (specifically in the PRAM model), starting in the late 1990s.

We will not attempt to give *practical* parallel algorithms in this section. That would require taking many complex issues into account at the same time, and it is essentially impossible to develop clean, deep, and instructive algorithmic ideas in such a convoluted setting. However, we can argue that a more *coarse-grained* view of parallelism, with a focus on communication bottlenecks, may circumvent many of the issues discussed above, while still providing a clean algorithmic framework. In fact, we will next discuss a theoretical model known as *Massively/Modern Parallel Computation (MPC)* which moves exactly in this coarse-grained direction. The study of MPC algorithms started less than 10 years ago and it has arguably created a renaissance in the area of *parallel algorithms*.

Massively Parallel Computation (MPC) Model: The system is composed of some M number of machines, each of which has a memory of size S words. This memory S is typically assumed to be significantly less than the input size N. As an intuitive guideline, try to think of S as a small polynomial of the input size, e.g., $S = N^{0.9}$, $S = N^{0.5}$, or $S = N^{0.1}$, depending on the problem and the setting.

The input is distributed arbitrarily across the machines, e.g., in the case of sorting each machine holds some of the items, and in the case of graph problems each machine holds some of the edges of the graph. Unless noted otherwise, there is no guarantee on how this input is distributed, besides the trivial limitation that each machine can hold at most S words of the input. Due to the input size, we have $M \geq \frac{N}{S}$. It is common to assume that

$M = C\frac{N}{S}$ for a constant $C \geq 2$, so that the overall memory in the system is slightly greater than the input size, but also not much more. In some problems studied in the literature, a further relaxation is considered where we assume $M = \frac{N}{S} \cdot \log^{O(1)} N$ or even $M = \frac{N}{S} \cdot N^{\delta}$ for some small $\delta > 0$. However, for the purpose of the problems discussed in this chapter, we will not need these relaxations.

The computation in MPC proceeds in lock-step synchronous rounds 1, 2, 3 .... Per round, each machine can do some computation on the data that it holds, and then it sends messages to the other machines. The model does not impose any particular restriction on the computations that each machine can perform in one round. However, we keep in mind that this should be a simple computation (polynomial time in the data size or ideally even near linear time). Most often, the algorithms that we see will use only simple computations, e.g., linear time in the size of the data that the machine holds. The key aspect of the model is to put emphasis on the communication bottlenecks, as they seem to be the main challenge in many practical settings of large-scale computation. As for the communication, the limitation is simple: per round, each machine can send at most S words and it can receive at most S words. Our algorithms need to just describe what information should be sent from each machine to each other machine, subject to the above constraints. The system takes care of routing and delivering this information in one round.

Further Discussions About the Model As can be viewed from above, MPC takes a much more coarse-grained approach to parallelism. It roughly tries to break computation into parallelizable parts, each defined by chunk of data of size S. Making S very small —e.g., $S = O(1)$ — would get us closer to the viewpoint of PRAM algorithms, with fine-grained parallel steps. But we will frequently work in the more coarse-grained regime where S is some small (sublinear) polynomial of the input size N, e.g., $S = \Omega(N^{0.5})$.

Furthermore, the limitation on S is in some (informal) sense modeling the communication bottlenecks at the same time as modeling memory bottlenecks (if they exist). In some sense, we can interpret S as capturing the communication bottleneck, even if there is no memory limitation: Suppose that the communication is limited such that per round we can communicate at most S words per machine, even if it has a larger memory. For most algorithms that we see in the MPC setting, the round complexity will

be rather small — e.g., $O(1)$, $O(\log\log n)$ or $O(\log n)$. Thus, just because of the communication bottleneck, each machine cannot receive much more than $\tilde{O}(S)$ words over the whole run of the algorithm. Thus, the memory restriction does not limit us significantly. In a rough sense, the model is implicitly taking S to be the minimum of communication limitation per round and the memory limitation.

6.7.2 MPC: Sorting

Sorting Problem in MPC: The input is n elements, stored arbitrarily in the machines. We assume that each machine has a memory S words (i.e., $O(S \log n)$ bits), which is considerably smaller than the input size n. In particular, we will assume that $S = n^{\varepsilon}$ for some constant $\varepsilon > 0$, e.g., $S = n^{0.1}$. We assume that we have $M = Cn/S$ machines, for some constant $C \geq 2$. Notice that n/S machines is absolutely necessary (why?) and our assumption means that the overall memory across the system $SP = Cn$ in asymptotically the same as the input size n. The output format is that each machine should know the rank (in the sorted list) of each of the items that it initially held.

QuickSort in MPC: We explain an algorithm that solves the sorting problem in constant rounds, with high probability. To be more precise, the round complexity will be $O(1/\varepsilon^2)$, where $S = n^{\varepsilon}$, with probability at least $1 - 1/n^{10}$. We leave achieving an $O(1/\varepsilon)$ round complexity as an exercise.

(A) First, we select a number of pivots. We make each machine mark each of its elements with probability $p = \frac{n^{\varepsilon/2}}{2n}$. With high probability, the number of marked elements is no more than $n^{\varepsilon/2}$.

(B) Then, we gather all the marked elements in one machine, say the first machine, in one round. This machine sorts all the marked elements.

(C) That machine broadcasts the sorted list to all other machines, in $O(1/\varepsilon)$ rounds, using an $\left(n^{\varepsilon/2}\right)$-ary broadcast tree among the machines. That is, the first machine sends the list to $\left(n^{\varepsilon/2}\right)$ machines, in one round. Then, each of these sends it to $\left(n^{\varepsilon/2}\right)$ new machines, in one round, all in parallel. And so on (Determine the detail on who should send to whom?). After $O(1/\varepsilon)$ rounds, each machine now has

all the pivots, and in a sorted list. At that point, it is clear to which subproblem between these pivots each item belongs.

(D) We use $O(1/\varepsilon)$ rounds of communication, performing a convergecast backward in the tree of the previous step, so that the leader machine knows the number of elements in each subproblem. The leader then determines for each i^{th} subproblem the machines allocated for solving it, as a contiguous interval $[l_i, u_i] \subseteq [1, M]$. The allocation is done such that the number of machines $(u_i - l_i + 1)$ for each subproblem is proportional to the number of elements in that subproblem. Since the total number of elements is n and the total memory is Cn for some constant $C \geq 2$, the leader can allocate machines so that, for each subproblem, on average, each machine is responsible for at most $n^\varepsilon/2$ items. Then, we again use $O(1/\varepsilon)$ rounds of communication, on the broadcast tree, to deliver all these indices to all the machines.

(E) Then, we shuffle the data where each machine sends each of the items that it holds to a random one of the machine responsible for the related subproblem. Since the average load per machine is $n^\varepsilon/2$ items, we can see that each machine receives at most n^ε items, with high probability (which fits its memory). During this shuffling, for each item, we remember the name of the initial machine that held it.

(F) Recurse on each subproblem separately. During the recursion, once a subproblem has size n^ε or smaller, sort it in one machine.

Lemma 6.19. *With probability $1 - 1/n^{10}$, after $O(1/\varepsilon)$ iterations of the above algorithm, the output is sorted. Since each iteration is implemented in $O(1/\varepsilon)$ rounds of MPC, the algorithm sorts the input in $O(1/\varepsilon^2)$ rounds of MPC.*

Proof Sketch. Consider one iteration starting with n elements: The number of elements between each two marked ones is at most $30n^{1-\varepsilon/2}\log n$, with high probability. This is because starting from a marked element, the probability of having $30n^{1-\varepsilon/2}\log n$ consecutive non-marked elements is at most $(1 - \frac{n^{\varepsilon/2}}{2n})^{30n^{1-\varepsilon/2}\log n} \leq e^{-15\log n} \leq 1/n^{15}$. Thus, with high probability, the size of each subproblem is at most $30n^{1-\varepsilon/2}\log n << n^{1-\varepsilon/3}$. Similarly, we can see that in each iteration the size of the subproblems is decreased by a factor of at least $n^{\varepsilon/3}$. Therefore, after $O(1/\varepsilon)$ iterations, the size of each subproblem falls below n^ε and we can sort it in one machine. □

Exercise 6.18. Sort Output Format
The above does not quite fit the output format that we wanted; it holds each item somewhere in the system, tagged with its rank and also the name of the initial machine that held it. Explain how we can obtain the desired output, with one additional round. Furthermore, an alternative output format would be to rearrange the items so that the i^{th} *machine, for* $i \in \{1, \ldots, P\}$*, holds items with rank* $(i-1)S+1$ *to* iS *(if such an item exists). Explain how we can obtain also this alternative output, in one additional round.*

Exercise 6.19. * Faster Sort in MPC
Devise an MPC algorithm that sorts n *items in* $O(1/\varepsilon)$ *rounds when each of* P *machines has a memory of* $S = n^{\varepsilon} = \Omega(\log n)$ *words, and we have* $P = \frac{Cn}{n^{\varepsilon}}$ *machines for a constant* $C \geq 2$*.*

6.7.3 MPC: Connected Components

Connected Components in MPC: The input is a graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ and $m = |E| \gg n$, and the objective is to identify the connected components of it. Regarding the memory in each machine of MPC, there are three different regimes, where the complexity of the problem is quite different among them: *strongly superlinear memory*, *almost linear memory*, and *strongly sublinear memory*. For this section, we will discuss only the first of these, which is easiest regime. Here, we assume that each machine has memory $S = n^{1+\varepsilon}$ for some constant $\varepsilon \in (0, 1]$, e.g., $\varepsilon = 0.1$.[5] Notice that this is still much smaller than the (potential) graph size $\Theta(n^2)$. We assume that we have $P = Cm/S$ machines for a constant $C \geq 2$. Notice that again m/S machines would be the bare minimum and our assumption gives us a minimal degree of flexibility so that we do not need to dive into messy details. We assume that the input is provided as follows: each edge $e = (v, u) \in E$ is stored in some machine. There is no guarantee besides this and edges of a node can be distributed arbitrarily across the system. The output format is that for node v, some machine should know the identifier of node v as well as the identifier of its connected component. In fact, in this generous regime of strongly superlinear mem-

[5] In contrast, the almost linear memory assumes $S = n \operatorname{poly}(\log n)$ or ideally just $S = n$, and the strongly sublinear memory regime assumes $S = n^{1-\varepsilon}$ for some constant $\varepsilon \in (0, 1)$.

ory, we can ask for much more: one machine (or even all machines) should know a maximal forest F of G.

Connected Components with Strongly Superlinear Memory: The algorithm is iterative and gradually removes redundant edges, until we have a maximal forest of the whole graph. Let m be the current number of edges. Pick the first $k = 2m/n^{1+\varepsilon}$ machines as players. Make each machine send each of its edges to a randomly chosen player. With high probability, each player receives at most $n^{1+\varepsilon}$ edges. Then, each player computes a maximal forest among the edges that it receives and it discards all the edges that it received but are not in its maximal forest. Discarding these edges cannot hurt (why?). The total number of remaining edges in each player is at most $n-1$. Hence, the total number of remaining edges among all players is at most $k(n-1) < 2m/n^{\varepsilon}$. Thus, in one round, we reduce the number of edges by a factor of $n^{\varepsilon}/2$. After at most $O(1/\varepsilon)$ repetitions, the number of remaining edges is below $n^{1+\varepsilon}$. All these edges can be put in one machine, who can then solve the problem among them and output the maximal forest.

Exercise 6.20. Extension to MST
Modify the above algorithm so that it outputs a minimum-weight spanning tree of any connected graph (or in the case of disconnected graphs, a maximal forest with minimum weight).

Remarks about Connected Components in Other Memory Regimes of MPC: Let us mention what is known for the more stringent settings of memory in MPC. For the linear and near linear memory regime, there is an $O(1)$ round algorithm in the memory regime of $S = O(n)$ [JN18]. In contrast, for the more stringent regime of strongly sublinear memory $S = n^{1-\varepsilon}$, where ε is a constant in $(0,1)$, the best known algorithm runs in $O(\log n)$ rounds (which is left as an exercise below). Furthermore, it remains a major open problem to resolve the round complexity in this strongly sublinear memory regime. Indeed, we do not know an algorithm faster than $O(\log n)$ rounds to distinguish whether the input graph is just a single n-node cycle or two disjoint $n/2$-node cycles (even if we are given the promise that the input is one of these two cases). Recently, an algorithm was developed that is faster whenever each component has a small diameter D, running

in roughly $O(\log D \cdot \log\log n)$ rounds [ASS+18].

Exercise 6.21. * Connectivity with Strongly Sublinear Memory *Devise an MPC algorithm that computes the connected components in $O(\log n)$ rounds when machine has a memory of $S = n^\alpha$ words for a given constant $\alpha \in (0, 1)$. For the input, the edges are distributed arbitrarily across different machines, each holding at most S edges. The output format is that for each node $v \in V$, some machine should hold the identifier of v and the identifier $D(v)$ of its component, where for any two nodes v and u, we have $D(v) = D(u)$ if and only if they are in the same connected component.*

Hint: *Think about the randomized method described in Section 6.5.2, where we merge fragments with each other randomly, using head/tail coin tosses, one per fragment. How do you identify the proposed edge of each fragment and the accepted edges? In general, you should explain what "structure" you maintain for each fragment, so that you can perform the related merges fast.*

6.7.4 MPC: Maximal Matching

We continue our discussion of graph problems, with the usual terminology that the input is a graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ and $m = |E| \gg n$. This time, we are interested in computing a maximal matching of G.

Matching, Maximal Matching, and Maximum Matching: Recall that a *matching* in a graph is a set $M \subseteq E$ of edges, no two of which share an endpoint. A *maximal matching* $M \subseteq E$ is a matching such that there is no matching M' where $M \subset M'$. We emphasize the strict subset, here, i.e., $|M'| \geq |M| + 1$. In simple words, M is a maximal matching if we cannot add any more edge to it, without violating the constraint that it remains a matching. Contrast this with the definition of *maximum matching*, which is the matching with absolutely largest cardinality possible in the whole graph. Notice that any maximum matching is a maximal matching, but the converse is not true (think about a 3-edge path: the central edge on its own is a maximal matching, but the two edges on the ends form the maximum matching).

Lemma 6.20. *Any maximal matching M has at least $|M^*|/2$ edges, where $|M^*|$ is the number of the edges of any maximum matching M^*.*

Proof. Left as an exercise. □

Maximal Matching in MPC: For this chapter, we will focus on the strongly superlinear memory regime and assume that $S = n^{1+\varepsilon}$ for some constant $\varepsilon \in (0, 1]$. We note that, again, the problem has a very different nature in the other regimes of near linear memory and strongly sublinear memory. The input format is that edges are distributed arbitrarily among the machines. The output format is that each machine should know which of its edges belong to the output maximal matching. Whenever $S = \Omega(n)$, we can also move all of the matching to one matching, and output it there.

The algorithm is made of iterations, where we gradually add edges to our matching M. Initially, M is empty. Then, each iteration works as follows:

- Let m be the number of remaining edges at the start of the iteration. Mark each edge with probability $p = n^{1+\varepsilon}/(2m)$ and move all marked edges to one machine (e.g., machine number one), in one round.
- There, compute a maximal matching among these marked edges, and add the edges of it to M.
- Send the names of all matched vertices to all machines, and make them remove from the graph (formally from their edges) all edges incident on these matched vertices.

It is clear that the algorithm removes a node only if it is matched. The main claim is that, with a very high probability, we are done after $O(1/\varepsilon)$ iterations, i.e., at that point, no edge remains in the graph and thus the computed matching M is maximal. That is implied easily by the following lemma, which bounds how the number of edges changes from one iteration to the next.

Lemma 6.21. *After each iteration starting with m edges, with probability $1 - \exp(-\Theta(n))$, the number of the remaining edges is at most $\frac{2m}{n^{\varepsilon}}$.*

Proof. Let I be the set of vertices that remain for the next iteration. Notice that there cannot be any marked edge with both of its endpoints in I.

Otherwise, at least one of those two endpoints would have been matched and thus removed (hence, not being in I). Now, we claim that this implies that the number of edges induced by I must be small. Consider any set $S \subseteq V$ of vertices and call it *heavy* if the subgraph induced by $G[S]$ has at least $\frac{2m}{n^\varepsilon}$ edges. The probability that no edge of $G[S]$ is marked is at most

$$(1-p)^{\frac{2m}{n^\varepsilon}} = \left(1 - \frac{n^{1+\varepsilon}}{2m}\right)^{\frac{2m}{n^\varepsilon}} \leq e^{-n}.$$

By a union bound over at most 2^n choices of S, we can deduce that, with probability at least $1 - e^{-n} \cdot 2^n \geq 1 - exp(-\Theta(n))$, for any heavy set S, we have at least one marked edge with its endpoints in S. Since the set I of the remaining vertices has no marked edge with both of its endpoints in I, we can infer that I cannot be a heavy set, with probability at least $1 - exp(-\Theta(n))$. That is, with probability $1 - exp(-\Theta(n))$, the number of the remaining edges is at most $\frac{2m}{n^\varepsilon}$. □

Remarks about Maximal Matching in Other Memory Regimes of MPC: The above algorithm is based on the work of Lattanzi et al. [LMSV11]. For the near linear memory regime of $S = n \operatorname{poly}\log n$, one can modify the above ideas to obtain an algorithm with round complexity $O(\log n)$ or even slightly below that, particularly $O(\frac{\log n}{\log\log n})$. For the more stringent strongly sublinear memory regime of $S = n^{1-\varepsilon}$, the best known round complexity is $O(\sqrt{\log n} \cdot \log\log n)$ [GU19]. This remains also the best known for the more relaxed $S = n \operatorname{poly}\log n$ memory regime. For that regime, the state of the art seems to be getting close to achieving a round complexity of $O(\log\log n)$—e.g., there is an "almost" maximal matching algorithm and also a $(1+\varepsilon)$-approximation algorithm for the maximum matching, in $O(\log\log n)$ rounds [GGK+18]— though we are not quite there yet.

6.7.5 MPC: Maximal Independent Set

Maximal Independent Set A set $S \subseteq V$ of vertices is called a Maximal Independent Set (MIS) of the graph $G = (V, E)$ if and only if we have the following two properties: (1) no two vertices in S are adjacent, (2) for each vertex $v \notin S$, there is a neighbor u of v such that $u \in S$.

Our focus regime of memory: For a change, we will examine this problem in the near-linear memory range and assume that each machine has memory

$S = \Omega(n \log^3 n)$ words. We will see an $O(\log \log n)$ round MIS algorithm for this regime. The space bound can be optimized to any $\frac{n}{\log^{O(1)} n}$, while keeping the same round complexity, but we do not discuss that here. We also note that for the strongly superlinear memory regime of $S = n^{1+\varepsilon}$ for any constant $\varepsilon \in (0, 1]$, one can obtain an $O(1/\varepsilon)$ round algorithm, based on ideas similar to what we saw in the previous section. In contrast, for the more stringent regime of strongly sublinear memory, where $S = n^{\varepsilon}$ for a constant $\varepsilon \in (0, 1)$, the currently best known algorithm is much slower and runs in $O(\sqrt{\log n} \cdot \log \log n)$ rounds [GU19].

Randomized Greedy Process for Choosing an MIS Consider the following *sequential* process for choosing an MIS. We will late see how to obtain a fast version of this, for the MPC setting. We choose a random permutation $\pi \in \mathcal{S}_n$ of the vertices, and then we process vertices in an order according to π, building an MIS greedily. That is, we check vertices $\pi(1)$, $\pi(2)$, ..., $\pi(n)$, one by one, each time adding the vertex $\pi(i)$ to our independent set S if and only if none of the previous vertices $\pi(j)$ adjacent to $\pi(i)$, for any $j < i$, has been added to S.

The above process will clearly build a maximal independent set. However, as is, it seems like we need to perform n steps, one after the other, and the process is not suitable for parallelism. Indeed, if π was an arbitrary permutation rather than a random permutation, it seems quite hard to find a fast parallel method to implement this idea[6]. However, we will see that for a random permutation π, the story is very different: indeed, we will be able to emulate this process in just $O(\log \log n)$ rounds of MPC, with memory of $S = \Omega(n \log^3 n)$ per machine.

Lemma 6.22. *Fix an arbitrary $t \in \{1, \ldots, n\}$. Consider the graph G_t resulting from removing all vertices $\pi(1)$, $\pi(2)$, ..., $\pi(t)$ as well as all vertices that are adjacent to the independent set S computed after processing $\pi(1)$, $\pi(2)$, ..., $\pi(t)$. With probability at least $1 - 1/n^5$, each node in G_t has degree at most $O(\frac{n \log n}{t})$.*

Proof. Consider a node v and suppose that $d_{t'}$ is the remaining degree of

[6] In fact, this is exactly the problem of Lexicographically-First MIS (LFMIS) which is known to be P-complete. That means, if we find a polylogarithmic depth PRAM algorithm for LFMIS, we have shown that all decision problems in P admit polylogarithmic depth PRAM algorithms.

v after processing $\pi(1), \pi(2), \ldots, \pi(t')$ for $t' \leq t$. If for any $t' \leq t$, we have $d_{t'} \leq \frac{10n \log n}{t}$, then we do not need to prove anything for this node. In the contrary, consider any step $t' < t$ where we have $d_{t'} \geq \frac{10n \log n}{t}$. Then, the probability that $\pi(t')$ is either node v and or one of its $d'_t \geq \frac{10n \log n}{t}$ neighbors is at least $\frac{10n \log n}{tn}$. If that happens, v would be removed. Thus, the only case where v remains with a high degree is if this does not happen, for any $t' < t$. The probability of that is at most $(1 - \frac{10n \log n}{tn})^t \leq e^{-10 \log n}$. By a union bound over all vertices, we conclude that the probability that there is any vertex v who remains with a degree higher than $\frac{10n \log n}{t}$ is at most $n \cdot e^{-10 \log n} < 1/n^5$. □

Preparation for MIS in MPC: We first move all the edges of each node to one machine, using the MPC sorting algorithm that we saw in Section 6.7.2, in constant rounds (how?).

Round Compression for Randomized Greedy MIS in MPC: We now discuss how we can compress the seemingly n rounds of computation in the randomized greedy procedure described above to just $O(\log \log n)$ rounds, in MPC. The algorithm has $O(\log \log n)$ iterations.

For the first iteration, we pick the $\sqrt{n}$ first vertices of the permutation π, and deliver the subgraph induced by them to one machine. Notice that this can be done in $O(1)$ rounds (why?). That machine can then run the process over $\pi(1)$ to $\pi(\sqrt{n})$ and report the resulting independent set S to all other machines. Each machine then discards all of its vertices that are in S or adjacent to S.

For the second iteration, we pick the first $n^{3/4}$ vertices of the permutation π and deliver the subgraph induced by the remaining ones of them to one machine. The subgraph induced by these nodes has $O(n \log n)$ edges, with high probability. The reason is that each of these $n^{3/4}$ vertices has degree at most $10\sqrt{n} \log n$, after we finished the first iteration, as Lemma 6.22 shows. Hence, we expect it to have at most $10\sqrt{n} \log n \cdot \frac{n^{3/4}}{n} = 10n^{1/4} \log n$ neighbors among the first $n^{3/4}$ vertices of π. Using a Chernoff bound, we can conclude that with high probability, the node has at most $20n^{1/4} \log n$ neighbors there. Now, summing up over all the first $n^{3/4}$ vertices of π, the number of edges is $n^{3/4} \cdot 20n^{1/4} \log n = 20n \log n$ edges. We have the capacity to deliver all these edges to one machine, in one round. That machine then simulates the greedy MIS process for vertices $\pi(\sqrt{n}+1), \ldots, \pi(n^{3/4})$,

and reports the resulting updated independent set S to all machines. They then remove any vertex that is in S or adjacent to S.

We then proceed to the next iteration. In general, in iteration i, we will continue the process over $\pi(\ell_{i-1}+1)$ to $\pi(\ell_i)$ where $\ell_i = n^{1-1/2^i}$, but in a way that it is compressed to just $O(1)$ rounds of MPC. It can be seen similar to the above paragraph that, thanks to the *degree drop behavior* stated in Lemma 6.22, the total number of the edges induced by these vertices is at most $O(n \log n)$, with high probability. Hence, all these edges can be delivered to one machine, thus allowing us to process all vertices $\pi(\ell_{i-1}+1)$ to $\pi(\ell_i)$ there. After $\log\log n$ iterations, we have processed vertices up to rank $\ell_{\log\log n} = n^{1-1/2^{\log\log n}} = n \cdot n^{-1/\log n} = n/2$. Then, by lemma Lemma 6.22, all remaining vertices have degree at most $20 \log n$ and thus we can move them to one machine and finish the process there.

Bibliography

[ASS+18] Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *Foundations of Computer Science (FOCS), arXiv:1805.03055*, 2018.

[Col88] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CV89] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, June 1989.

[FGT16] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 754–763, 2016.

[FGT19] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. *SIAM Journal on Computing*, (0):STOC16–218, 2019.

[GG17] Shafi Goldwasser and Ofer Grossman. Bipartite perfect matching in pseudo-deterministic nc. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[GGK+18] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 129–138, 2018.

[GU19] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Symposium on Discrete Algorithms (SODA), arXiv:1807.06251*, 2019.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[JN18] Tomasz Jurdziński and Krzysztof Nowicki. MST in O(1) rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2620–2632. SIAM, 2018.

[KR90] Richard M. Karp and Vijaya Ramachandran. Handbook of theoretical computer science (vol. a). chapter Parallel Algorithms for Shared-memory Machines, pages 869–941. MIT Press, Cambridge, MA, USA, 1990.

[Lei14] F Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes*. Elsevier, 2014.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *the Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 85–94, 2011.

[MVV87] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

[Nai82] Mohan Nair. On chebyshev-type inequalities for primes. *The American Mathematical Monthly*, 89(2):126–129, 1982.

[ST17] Ola Svensson and Jakub Tarnawski. The matching problem in general graphs is in quasi-nc. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 696–707, 2017.