Introduction to Parallel Algorithms
(DRAFT)

Guy E. Blelloch, Laxman Dhulipala and Yihan Sun

February 25, 2024

Contents

1 Introduction

This document is intended an introduction to parallel algorithms. The algorithms and techniques described in this document cover over 40 years of work by hundreds of researchers. The earliest work on parallel algorithms dates back to the 1970s. The key ideas of the parallel merging algorithm described in Section 4.4, for example, appear in a 1975 paper by Leslie Valiant, a Turing Award winner. Many other Turing award winners have contributed to ideas in this document including Richard Karp, Robert Tarjan, and John Hopcroft.

The focus of this document is on key ideas which have survived or are likely to survive the test of time, and are likely to be useful in designing parallel algorithms now and far into the future.

Although this document is focused on the theory of parallel algorithms, many, if not most, of the algorithms and algorithmic techniques in this document have been implemented on modern multicore machines (e.g., your laptop, iphone, or server). The algorithms most often, sometimes with tweaks, far outperform the best sequential algorithms on the same machine, even ones with a modest number of cores.

This document is a work in progress. Please tell us about any bugs you find and give us feedback on the presentation.

2 Models

To analyze the cost of algorithms, it is important to have a concrete model with a well-defined notion of costs. Sequentially, it is well-known that the Random Access Machine (RAM) model has served well for many years. The RAM model is meant to approximate how real sequential machines work. It consists of a single processor with some constant number of registers, an instruction counter and an arbitrarily large memory. The instructions include register-to-register instructions (e.g. adding the contents of two registers and putting the result in a third), control-instructions (e.g. jumping), and the ability to read from and write to arbitrary locations in memory. For the purpose of analyzing cost, the RAM model assumes that all instructions take unit time. The "time" (cost) of a computation is then just the total number of instructions it performs from the start until a final END instruction. To allow storing a pointer to memory in a register or memory location, but disallow playing games by storing arbitrary large values, most often it is assumed that for an input of size n, each register and memory location can store $\Theta(\log n)$ bits.

The RAM is by no stretch meant to model the runtime on a real machine with cycle-by-cycle level accuracy. It does not model, for example, that modern-day machines have cache hierarchies and therefore not all memory accesses are equally expensive. Modeling all features of modern-day machines would lead to very complicated models that could be hard to use and hard to gain intuition from. Instead the RAM tries to be simple. As such the RAM does not precisely model the performance of any particular real machine, it can, and has, effectively served to compare different algorithms, and understand how the performance of the algorithms will scale with size. For these reasons the RAM model should really only be used for asymptotic (i.e. big-O) analysis. The RAM has also served as a tool to better understand algorithmic techniques such as divide-and-conquer, and dynamic programming, among many others. Finally, and importantly, it is natural to write pseudocode or real code that is naturally translated to the RAM.

In the context of parallel algorithms, we would like to use a cost model that satisfies the same important features—i.e., simple, guides the user to the most efficient algorithms, helps the user understand how an algorithm scales with input size, robust across a variety of machines, helps the user understand algorithmic techniques, and can be naturally expressed with simple pseudocode and real code. Early work on parallel algorithms largely used the Parallel RAM (PRAM) model [23]. The model consists of p fully synchronous processors accessing a shared memory. Costs are measured in terms of the number of processors and the number of time steps. Unfortunately the PRAM is not particularly well suited for simple pseudocode, and consequently real code, due to its assumption of a fixed number of processors—the user needs to worry about allocating tasks to processors.

More recently researchers have used so-called *work-span* (or work-depth) models in which algorithms still assume a shared random access memory, but allow dynamically creating tasks (or processes). Costs are measured in terms of the total number of operations, the *work* and the longest chain of dependences (the depth or the *span*). This simplifies the description and analysis of algorithms—avoiding the need to talk about scheduling tasks to processors.

In this document we use a work-span model, the MP-RAM, which we feel perhaps best satisfies the features for a good parallel algorithmic model. It is based on the RAM, but allows the dynamic forking of new processes. Fortunately costs are robust whether analyzed in the PRAM and MP-RAM (and other models). If we define work in the PRAM as the product of the number of processors and the time, almost all algorithms in this document have the same work in the PRAM or MP-RAM. If we define span in the PRAM as time, then all algorithms in this document have a span in the PRAM that differs by at most a logarithmic factor from the MP-RAM.

It may not be obvious how to map these dynamic processes onto a physical machine which will only have a fixed number of processors. To convince ourselves that it is possible, later we show how to design schedulers that map the processes onto processors, and prove bounds that relate costs. In particular we show various forms of the following work-span, processor-time relationship:

$$\max\left(\frac{W}{P}, D\right) \leq T \leq O\left(\frac{W}{P} + D\right) \tag{1}$$

where W is the work, D the span, P the processors, and T the running time using P processors. More details on other models and simulations among them is given in Section **??**

When evaluating parallel computations, we are especially interested in the two quantities work and span, because they each reflects an aspect of the parallel computation. The *work*, as indicated by its definition, is equivalent to the running time of the same computation on one processor (or, sequentially). One might be curious about why we need to care about the sequential time complexity for a parallel algorithm. First of all, the total work still indicates how much resource the computation needs, include the computational ability, power, etc. More importantly, from the running time point of view, the total work affects the time in the W/P term (see equation 1). In modern multicore machines, the number of processors P is usually small compared to the work W, as it is usually $\Omega(n)$ for input size n. Meanwhile, often D is asymptotically much smaller than W (see more details below). This makes the running time bound to be dominated by the total work W. We call a parallel algorithm *work-efficient*, if its work is work asymptotically the same as its best-known sequential counterpart. When design parallel algorithms, our goal is usually to make the algorithm work-efficient, or at least close to work-efficient.

The span for a parallel algorithm reflects another aspect of the computation, that is, what is the running time when you have an infinite number of processors. In other words, this is a "lower bound" of running time whatever the number of processors. Although as mentioned above, the running time is usually bounded by work W, the span indicates an algorithm's ability to gain better performance on more processors. This is also referred to as the *scalability* of the algorithm [1]. Intuitively, when designing parallel algorithms, the goal in terms of span is to not let it dominate the cost. Generally speaking, we would like to bound the span to be polylogarithmic in n.

Another measure that can be derived from the work and span is *parallelism*, which is defined simply as the work divided by the span. It indicates, asymptotically speaking, how many processors can be effectively used by the computation. If the work equals span then the parallelism is 1 and the computation is sequential. If the work is $O(n \log n)$ and the span is $O(\log^2 n)$ then the parallelism is $O(n/\log n)$ which is actually quite high, and unlikely to be a bottleneck on most machines in the next 25 years. If the work is $O(n^2)$ and the span is $O(n)$ then the parallelism is $O(n)$, which is even higher, even though the span is not polylogarithmic.

In summary, the primary focus should be on designing work-efficient algorithms with good parallelism, and, ideally but not necessarily, with polylogarithmic span.

The work-span model is a simple cost model, which specifies how the total cost of the algorithm should be evaluated. It does not specify, however, what operations are allowed or disallowed in a parallel computation, e.g., what type of concurrency or synchronization is supported. In the following, we introduce the MP-RAM model, which will be the main model used in this book.

[1] In many other settings, scalability means the ability to gain good performance when the *input size* increases. In parallel algorithms, especially in this book, we use this term to mean the ability to gain performance when the number of processors increases.

MP-RAM

The *Multi-Process Random-Access Machine (MP-RAM)* consists of a set of processes that share an unbounded memory. Each process runs the instructions of a RAM—it works on a program stored in memory, has its own program counter, a constant number of its own registers, and runs standard RAM instructions. The MP-RAM extends the RAM with a FORK instruction that takes a positive integer k and forks k new child processes. Each child process receives a unique integer in the range $[1, \ldots, k]$ in its first register and otherwise has the identical state as the parent (forking process), which has that register set to 0. All children start by running the next instruction, and the parent suspends until all the children terminate (execute an END instruction). The first instruction of the parent after all children terminate is called the *join* instruction. A *computation* starts with a single root process and finishes when that root process ends. This model supports *nested parallelism*—the ability to fork processes in a nested fashion. If the root process never does a fork, it is a standard sequential program.

A computation in the MP-RAM defines a partial order on the instructions. In particular (1) every instruction depends on its previous instruction in the same thread (if any), (2) every first instruction in a process depends on the fork instruction of the parent that generated it, and (3) every join instruction depends on the END instruction of all child processes of the corresponding fork generated. These dependences define the partial order. The *work* of a computation is the total number of instructions, and the *span* is the longest sequences of dependent instructions. As usual, the partial order can be viewed as a DAG. For a fork of a set of child processes and corresponding join the span of the subcomputation is the maximum of the span of the child processes, and the work is the sum. This property is useful for analyzing algorithms, and specifically for writing recurrences for determining work and span.

We assume that the results of memory operations are consistent with some total order (linearization) on the instructions that preserves the partial order—i.e., a read will return the value of the previous write to the same location in the total order. The choice of total order can affect the results of a program since processes can communicate through the shared memory. In general, therefore computations can be nondeterministic. Two instructions are said to be *concurrent* if they are unordered, and *ordered* otherwise. Two instructions *conflict* if one writes to a memory location that the other reads or writes the same location. We say two instructions *race* if they are concurrent and conflict. If there are no races in a computation, and no other sources of non-determinism, then all linearized orders will return the same result. This is because all pairs of conflicting instructions are ordered by the partial order (otherwise it would be a race) and hence must appear in the same relative order in all linearizations. A particular linearized order is to iterate sequentially from the first to last child in each fork. We call this the *sequential ordering*.

Pseudocode. Our pseudocode will look like standard sequential code, except for the addition of two constructs for expressing parallelism. The first construct is a *parallel* *loop* indicated with parFor. For example the following loop applies a function f to each element of an array A, writing the result into an array B:

```
parfor $i$ in $[0 : |A|]$
  $B[i] \leftarrow f(A[i])$
```

In pseudocode $[s : e]$ means the sequence of integers from s (inclusive) to e (exclusive), and ← means assignment. Our arrays are zero based. A parallel loop over n iterations can easily be implemented in the MP-RAM by forking n children applying the loop body in each child and then ending each child. The work of a parallel loop is the sum of the work of the loop bodies. The span is the maximum of the span of the loop bodies.

The second construct is a *parallel do*, which just runs some number of statements in parallel. In pseudocode we use a semicolon to express sequential ordering of statements and double bars (||) to express parallel statements. For example the following code will sum the elements of an array.

```
sum$(A)$ =
  if $(|A| = 1)$ then return $A[0]$
  else
    $l \leftarrow$ sum$(A[0 : |A|/2])$ ||
    $r \leftarrow$ sum$(A[|A|/2 : |A|])$;
    return $l + r$
```

The || construct in the code indicates that the two statements with recursive calls to `sum` should be done in parallel. The semicolon before the return indicates that the two parallel recursive calls have to complete before adding the results. In our pseudocode we use the $A[s : e]$ notation to indicate the slice of an array between location s (inclusive) and e (exclusive). If s (or e) is empty it indicates the slice starts (finishes) at the beginning (end) of the array. Taking a slices takes $O(1)$ work and span since it need only keep track of the offsets.

The || construct directly maps to a fork in the MP-RAM, in which the first and second child run the two statements. Analogously to parFor, the work of a || is the sum of the work of the statements, and the span is the maximum of the spans of the statements. For the `sum` example the overall work can be written as the recurrence:

$$W(n) = W(n/2) + W(n/2) + O(1) = 2W(n/2) + O(1)$$

which solves to $O(n)$, and the span as

$$D(n) = \max(D(n/2), D(n/2) + O(1) = D(n/2) + O(1)$$

which solves to $O(\log n)$.

It is important to note that parallel-for loops and parallel-do (||) instructions can be nested in an arbitrary fashion.

Binary or Arbitrary Forking. Some of our algorithms use just binary forking while others use arbitrary n-way forking. This makes some difference when we discuss scheduling the MP-RAM onto a fixed number of processors, and can make a difference in the span bounds. We will use the term the *binary-forking* MP-RAM, or *binary-forking model* for short, to indicate the version that only allows binary forking. In some cases we give separate bounds for MP-RAM and binary forking model. It is always possible to implement n-way forking using binary forking by creating a tree of binary forks of span $\log n$. Thus in general this can increase the span, but not always does so. In these cases we will use `parfor2` to indicate we are using a tree to fork the n parallel calls.

Scheduling. Clearly the MP-RAM is an abstract model since one cannot just create new processors by calling a fork instruction. In reality the processes of the MP-RAM have to be scheduled onto actual processors on the machine. This is the job of a scheduler. The scheduler needs to keep some kind of work queue and then distribute the work to processors. When forking, new tasks (processes) are added to the queue, and when a processor finishes its task, it goes to the queue for more work. Here we use "queue" in a generic sence since it could implemented in a distributed fashion, and it might be first-in first-out, or last-in last out, or an arbitrary combination.

Implementing schedulers will be a topic of later chapters, and will be used to show bounds such as given in Equation 1.

Additional Instructions. In the parallel context it is useful to add some additional instructions that manipulate memory. Some commonly-used instructions are a test-and-set (TS), compare-and-swap (CAS), fetch-and-add (FA), and priority-write (PW). These primitives are atomic, meaning that they are indivisible. Even when executed concurrently with other instructions, these primitives will either be correctly completed, or as if not executed (does not affect the shared memory state). These instructions are supported by most modern architecture, either directly or indirectly. We discuss our model with one of these operations enabled as the TS-, FA-, and PW-variants of the MP-RAM.

A TESTANDSET(x) (TS) instruction takes a reference to a memory location x, checks if the value of x is `false` and if so atomically sets it to `true` and returns `true`; if already `true` it returns `false`.

A COMPAREANDSWAP(x, o, n) (CAS) instruction takes a reference to a memory location x, checks if the value of x equals o. If so, the instruction will change the value to n and return `true`. If not, the instruction does nothing and simply returns `false`.

A FETCHANDADD(x, y) (FA) instruction takes a reference to a memory location x, and a value y, and it adds y to the value of x, returning the old value. Different from a TS or a CAS, an FA instruction always successfully adds y to the value stored in x.

A PRIORITYWRITE(x, y) (PW) instruction takes a reference to a memory location x, and checks if the value y is less than the current value in x. If so, it changes the value stored in x to y, and return `true`. If not, it does nothing and return `false`.

For the operations above returning a boolean value (TS, CAS and PW), we say it succeeds (or is successful) if it returns a `true`, and otherwise it fails (or is unsuccessful).

It is worth noting that some of these instructions can be used to implement some other instructions, and thus more powerful, while some of them cannot. Usually, TS is considered to be the weakest instruction, as it cannot be used to simulate any of the other instructions. On the other hand, the other three can all be used to simulate TS.

Memory Allocation. To simplify issues of parallel memory allocation we assume there is an ALLOCATE instruction that takes a positive integer n and allocates a contiguous block of n memory locations, returning a pointer to the block, and a FREE instruction that given a pointer to an allocated block, frees it.

3 Preliminaries

We will make significant use of randomized algorithms in this document and assume some reasonable understanding of probability theory (e.g., random variables, expectations, independence, conditional probability, union bound, Markov's inequality). We are often interested in tail bounds of probability distributions, and showing that the probability of some tail is very small—i.e. the cumulative probability above some threshold value in a probability density function is very small. Hence the cumulative probability not in this tail is very high, approaching 1.

These tail bounds are often asymptotic as a function of some parameter n. In particular we will use the following definiton for "high probability".

Definition 3.1 (w.h.p.)**.** *$g(n) \in O(f(n))$ with high probability (w.h.p.) if $g(n) \in O(cf(n))$ with probability at least $1 - \left(\frac{1}{n}\right)^c$, for some constant c_0 and all $c \geq c_0$.*

This definition allows us to make the probability arbitrarily close to 1 by increasing c, and even for a modest constant c (e.g. 3), the probability is very close to 1.

Such high-probability, or tail, bounds are particularly important in parallel algorithms since we often cannot effectively use expectations. In a sequential algorithm we are just adding times so we can compose the times taken by a bunch of components by adding their expected times and relying on the linearity of expectations to get the overall expected time. In parallel we often run multiple components in parallel and wait for the last to finish. Therefore the overal span is the maximum of the spans of the components, not the sum.

Unfortunately there is no equivalent to linearity of expectations for maximums. Instead a common approach is to show that the probability that each component will not finish within some span is very small. The union bound tells us that if we sum these probabilities, this is an upper bound on the probability that any will not finish within that bound. The goals is therefore to then show that this sum is still sufficiently small. The polynomial $(1/n)^c$ term in the high probability bound is important since when we multiply this possibility of not being within the bounds by n parallel components, we have that the probability of them all being within our bounds is at least $1 + (1/n)^{c-1}$. This is sufficient since we can use any constant c.

There are many ways to derive tail bounds, many based on Chernoff or Hoeffding bounds. In this document we will mostly use the following rather simple inequality for tail bounds.

Theorem 3.1. *Consider a set of indicator random variables $X_1, \ldots X_n$ for which $p(X_i = 1) \le \bar{p}_i$ conditioned on all possible events $X_j = \{0, 1\}, i \ne j$. Let $X = \sum_{i=1}^{n} X_i$ and $\bar{E}[X] = \sum_{i=1}^{n} \bar{p}_i$, then:*

$$\mathbf{Pr}[X \ge k] \le \left(\frac{e\bar{E}[X]}{k}\right)^k .$$

Proof. Let's first consider the special case that the $\bar{p}_i$ are all equal and have value p. If $X \ge k$ then we have that at least k of the random variables are 1. The probability of any particular k variables all being 1, and the others being anything, is upper bounded by p^k. Now we use the union bound over the $\binom{n}{k}$ possible choices of k variables, yielding:

$$\mathbf{Pr}[X \ge k] \le p^k \binom{n}{k} < p^k \left(\frac{ne}{k}\right)^k = \left(\frac{pne}{k}\right)^k = \left(\frac{e\bar{E}[X]}{k}\right)^k$$

Here we used a standard upper bound on the binomial coefficients: $\binom{n}{m} < \left(\frac{ne}{m}\right)^m$.

Now consider the case when the $\bar{p}_i$ are not equal. The claim is that making them unequal can only decrease the total bound on probability. This can be seen by starting with all equal probabilities and then moving some probability from one variable to another. Some of the k subsets will include both variables, and these probabilities will go down since if a set of non-negative values sums to a fixed value, their product is maximized when all are equal and decreases when two are further separated. For every subset than includes just the first there is an equivalent subset containing just the second (i.e. all the same except swapping the first for the second). The sum of the product of each of these pairs will not change. Any k including neither will clearly not change. Hence the sum of the probabilities of all subsets of size k is maximized when all probabilities are the same. □

It may seem that instead of bounding the conditional probabilities we could just require that X_i are independent. For many cases this will suffice, and then we can use exact probabilities and expectations. For some useful cases, however, the random variables will not be independent, but we will be able to show an upper bound on the conditional probabilities. In these cases the lack of independence could make the actual probabilities better, but not worse than this bound. The theorem as stated is therefore more powerful than one that requires independence—it fully captures the independent case but also captures more.

One way to remember this tail bound is by noting its similarity to Markov's inequality:

$$\mathbf{Pr}[X \geq k] \leq \left(\frac{E[X]}{k}\right)$$

The only difference is the added power by k and added constant e. Markov's inequality does not require any kind of independence but due to the lack of the power of k can be a much weaker bound. Therefore perhaps you can remember this as the "power" of independence.

We are often interested in asymptotics and will often use the following Corollary:

Corollary 3.1. *Consider a set of indicator random variables $X_1, \ldots X_n$ with varying n for which $p(X_i = 1) \leq \bar{p}_i$ conditioned on all possible events $X_j = \{0, 1\}, i \neq j$. Let $X(n) = \sum_{i=1}^{n} X_i$ and $f(n) = \sum_{i=1}^{n} \bar{p}_i$, then $X(n) \in O(f(n) + \log n)$ w.h.p.*

Proof. Based on Theorem 3.1 we have that

$$\mathbf{Pr}[X(n) \geq g(n)] \leq \left(\frac{ef(n)}{g(n)}\right)^{g(n)}$$

Choosing $g(n) = 2ec(f(n) + \lg n)$ we have that

$$\begin{aligned} \mathbf{Pr}[X(n) \geq g(n)] &\leq \left(\tfrac{f(n)}{2f(n)+\ln n}\right)^{g(n)} \\ &\leq \left(\tfrac{1}{2}\right)^{g(n)} \\ &\leq \left(\tfrac{1}{2}\right)^{2ec\lg n} \\ &= \left(\tfrac{1}{n}\right)^{2ec} \\ &\leq \left(\tfrac{1}{n}\right)^{c} \end{aligned}$$

Now clearly $g(n) \in O(c(f(n) + \log n)$ and the probability of $X(n) \leq g(n)$ is at least $1 - (1/n)^c$, satisfying our definition for high probability. □

As an example, which we will see several times, let's consider the case of a set of independent indicator random variables $X_1, \ldots X_n$ each with $p(X_i = 1) = 1/i$. Here $f(n) = \sum_{i=1}^{n} 1/i < \ln n + 1$ so the corollary tells us that the sum of the indicator random variables is $X(n) \in O(\lg n)$ w.h.p.

Corollary 3.1 cannot always be applied where Theorem 3.1 can. As an example consider throwing n balls into n bins randomly, and bounding the size of the maximum sized bin. Here we first analyze the size of any one bin, and the X_i indicate whether the i-th ball ends up in that one bin. We have $p_i = 1/n$, and let the total number in that bin be X. We have that $E[X] = 1$, giving

$$\mathbf{Pr}[X \geq k] \leq \left(\frac{e}{k}\right)^{k}$$

Setting $k = (c+1)e \ln n / \ln\ln n$ gives:

$$\begin{aligned}
\mathbf{Pr}[X \geq k] &\leq \left(\frac{\ln\ln n}{(c+1)\ln n}\right)^{(c+1)e\ln n/\ln\ln n} \\
&\leq \left(\frac{\ln\ln n}{\ln n}\right)^{(c+1)e\ln n/\ln\ln n} \\
&= \left(e^{-\ln\ln n + \ln\ln\ln n}\right)^{(c+1)e\ln n/\ln\ln n} \\
&\leq \left(e^{-.5\ln\ln n}\right)^{(c+1)e\ln n/\ln\ln n} \\
&= \left(e^{-.5}\right)^{(c+1)e\ln n} \\
&= n^{-.5e(c+1)} \\
&\leq \left(\frac{1}{n}\right)^{c+1}
\end{aligned}$$

This tells us that $X(n) \in O(\ln n/\ln\ln n)$ w.h.p. which is is tighter by a $\ln\ln n$ factor than what Corollary 3.1 gives. This just bounds the size of one bin, but we can take the union bound over n bins by multiplying the probability by n upper bounding the probability that any bin is larger than $k = (c+1)e\ln n/\ln\ln n$ by $n\left(\frac{1}{n}\right)^{c+1} = \left(\frac{1}{n}\right)^{c}$. Hence the probability that no bin is larger than $O(c\ln n/ln\ln n)$ is at least $1 - \left(\frac{1}{n}\right)^{c}$, and the largest bin is of size $O(\ln n/\ln\ln n)$ w.h.p.

4 Some Building Blocks

Several problems, like computing prefix-sums, merging sorted sequences and filtering frequently arise as subproblems when designing other parallel algorithms. Here we cover some of these basic building blocks. In some cases we will see there are tradeoffs between work and span. In some cases we will see that arbitrary forking gives some additional power in reducing the span.

4.1 Scan

A *scan* or *prefix-sum* function takes a sequence A, an associative function f, and a left identity element $\perp$ and computes the values:

$$r_i = \begin{cases} \perp & i = 0 \\ f(r_{i-1}, A_i) & 0 < i \leq |A| \end{cases}$$

Each r_i is the "sum" of the prefix $A[0,i]$ of A with respect to the function f. Often it is returned as an array of length $|A|$, where the value at index i is the sum of all elements preceding the i-th element, and the total sum, as in:

$$[r_0, \ldots, r_{|A|-1}], r_{|A|} \ .$$

The scan function is useful because it lets us compute a value for each element in an array that depends on all previous values, something that seems inherently sequential, but can in fact be performed efficiently in parallel. We often refer to the plusScan operation, which is a scan where f is addition, and $\perp = 0$.

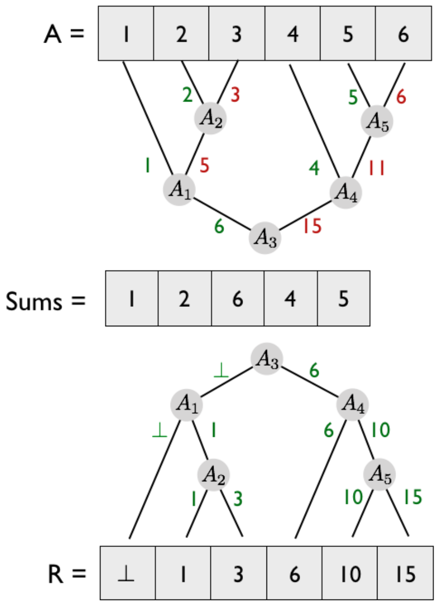


Figure 1: Recursive implementation of scan. *Sorry,* sums *in the figure corresponds to L in the code and description.*

Pseudocode for a recursive implementation of scan is given in Figure 2. The code works with an arbitrary associative function f. Conceptually, the implementation performs two traversals of a balanced binary tree built on the array. Both traversals traverse the tree in exactly the same way recursively, although the first does its work from from the leaves to root (i.e., applies f after the recursive calls), while the second does its work from root to leaves (i.e., applies f before the recursive calls). Figure 2 visually illustrates both traversals. Each node of the tree corresponds to a recursive call, and the interior nodes are labeled by the element in A that corresponds to the position of of the midpoint m for that call.

The first traversal, scanUp computes partial sums of the left subtrees storing them in L. It does this bottom-up: each call splits A at the middle m, and splits L into two ranges ($L[0 : m-1]$ and $L[m : n-1]$) and one element left out in the middle (i.e. $L[m-1]$) for writing its result. It recurses on the splits of A and L. Each call returns the sum of its range of A with respect to f, and each internal call corresponding to an internal tree node (i.e., not the base cases) writes the result it receives from the left call into $L[m-1]$. The array L of partial sums has size $|A|-1$ since there are $n-1$ internal nodes for a tree with n leaves.

The second traversal, scanDown, performs a top-down traversal that receives a value s from its parent (the identify I for the top-level call) and again makes two recursive calls in parallel on its left and right children. Each call passes the s it received from the parent, directly to its left child, and passes $f(s, L[m])$ to its right child. Recall that $L[m]$ is the sum of the left child with respect to the combining function f. Leafs

```
scanUp$(A, L, f)$ =
  if $(|A| = 1)$ then return $A[0]$
  else
    $n \leftarrow |A|$;
    $m \leftarrow n/2$;
    $l \leftarrow$ scanUp$(A[0:m], L[0:m-1], f)$ ||
    $r \leftarrow$ scanUp$(A[m:n], L[m:n-1], f)$;
    $L[m-1] \leftarrow l$;
    return $f(l, r)$
```

```
scanDown$(R, L, f, s)$ =
  if $(|R| = 1)$ then $R[0] = s$; return;
  else
    $n \leftarrow |A|$;
    $m \leftarrow |R|/2$;
    scanDown$(R[0:m], L[0:m-1], s)$ ||
    scanDown$(R[m:n], L[m:n-1], f(s, L[m-1]))$;
    return

scan$(A, f, I)$ =
  $L \leftarrow$ array$[|A|-1]$;
  $R \leftarrow$ array$[|A|]$;
  total $\leftarrow$ scanUp$(A, L, f)$;
  scanDown$(R, L, f, I)$;
  return $\langle R,$ total$\rangle$;
```

Figure 2: The scan function.

in the tree write the value passed to them by the parent into the result array R.

We now consider why this algorithm is correct. Firstly for scanUp it should be clear that the values written into L are indeed the sums of the left subtrees. For scanDown consider a node v in the tree and the value s passed to it. The invariant the algorithm maintains it that the value s is the sum of all values to the left of the subtree rooted at v. The claim is that if this is true at the parent then it will be true at the two children. It is true for the root since there is nothing to the left and the identify is passed in. It is true for each left child of a node since it is passed s directly and it has the same values to the left of it as the parent. It is also true on the right child since the values to the left of the right subtree are exactly the disjoint union of the values to the left of the subtree of v as a whole and the values in the left subtree of v. So combining the sums of these two parts using f, gives the overall sum to the right subtree. Finally, when the algorithm gets to the base case (a leaf) it writes in the appropriate position of R exactly the sum of everything to its left.

Our algorithm does not require that f is commutative. Whenever it combines values, it combines them in the correct order, with values on the left used as the left argument to f and on the right as the right argument.

Assuming that our function f taked $O(1)$ work, The work of scanUp and scanDown is given by the following recurrence

$$W(n) = 2W(n/2) + O(1)$$

which solves to $O(n)$, and the span as

$$D(n) = D(n/2) + O(1)$$

which solves to $O(\log n)$. For functions f that take other costs we would have to plug their costs into the recurrences to determine the overall cost. The scan algorithm works in the binary-forking model since we only make pairwise recursive calls.

```
FILTER$(A, p)$ =
    $n \leftarrow |A|$;
    $F \leftarrow$ ARRAY$[n]$;
    parfor $i$ in $[0 : n]$
        $F[i] \leftarrow p(A[i])$
    $\langle X, m\rangle \leftarrow$ PLUSSCAN$(F)$;
    $R \leftarrow$ ARRAY$[m]$;
    parfor $i$ in $[0 : n]$
        if $(F[i])$ then $R[X[i]] \leftarrow A[i]$
    return $R$
```

Figure 3: The FILTER function.

```
FLATTEN$(A)$ =
    sizes $\leftarrow$ ARRAY$(|A|)$;
    parfor $i$ in $[0 : |A|]$
        sizes$[i] \leftarrow |A[i]|$
    $\langle X, m\rangle \leftarrow$ PLUSSCAN(sizes);
    $R \leftarrow$ ARRAY$(m)$;
    parfor $i$ in $[0 : |A|]$
        $o \leftarrow X[i]$;
        parfor $j$ in $[0 : |A[i]|]$
            $R[o + j] \leftarrow A[i][j]$
    return $R$
```

Figure 4: The FLATTEN function.

In the special case of a PLUSSCAN of n integers in the range $[0 : n^c]$, it is possible to improve the bounds to $O(n)$ work and $O(\log n / \log \log n)$ span with the multi-way MP-RAM [12]. The approach is quite complicated, breaking the words into parts which are then scanned independently, and using table lookup as a building block.

4.2 Filter and Flatten

The *filter* primitive takes as input a sequence A and a predicate p and returns an array containing $a \in A$ s.t. $p(a)$ is true, in the same order as in A. Pseudocode for the filter function is given in Figure 3. We first compute an array of flags, F, where $F[i] = p(A[i])$, i.e. $F[i] = 1$ iff $A[i]$ is a live element that should be returned in the output array. Next, we PLUSSCAN the flags to map each live element to a unique index between 0 and m, the total number of live elements. Finally, we allocate the result array, R, and map over the flags, writing a live element at index i to $R[X[i]]$. We perform a constant number of steps that map over n elements, so the work of filter is $O(n)$, and the span is $O(\log n)$ because of the PLUSSCAN.

The *flatten* primitive takes as input a nested sequence A (a sequence of sequences) and returns a flat sequence R that contains the sequences in A appended together. For example, FLATTEN$([[3, 1, 2], [5, 1], [7, 8]])$ returns the sequence $[3, 1, 2, 5, 1, 7, 8]$.

Pseudocode for the flatten function is given in Figure 4. We first write the size of each array in A, and PLUSSCAN to compute the size of the output. The last step is to map over the $A[i]$'s in parallel, and copy each sequence to its unique position in the output using the offset produced by PLUSSCAN.

4.3 Search

The sorted search problem is given a sorted sequence A and a key v, to find the position of the greatest element in A that is less than v. It can be solved using binary search in $O(\log |A|)$ work and span. In parallel it is possible to reduce the span, at the cost of increasing the work. The idea is to use a k-way search instead of binary search.

```
// finds which of $k$ blocks contains v, returning block and offset
findBlock$(A, v, k)$ =
  $s \leftarrow |A|/k$;
  $r \leftarrow k$;
  parfor $i$ in $[0 : k]$
    if $(A[i \times s] < v$ and $A[(i+1) \times s] > v)$
    then $r \leftarrow i$
  return $(A[r \times s, (r+1) \times s], i \times s)$

search$(A, v, k)$ =
  $(B, o)$ = findBlock$(A, v, \min(|A|, k))$;
  if $(|A| \leq k)$ then return $o$
  else return $o$ + search$(B, v, k)$;
```

Figure 5: The search function.

```
kthHelp$(A, a_o, B, b_o, k)$ =
  if $(|A| + |B| = 0)$ then return $(a_o, b_o)$;
  else if $(|A| = 0)$ then return $(a_o, b_o + k)$;
  else if $(|B| = 0)$ then return $(a_o + k, b_o)$;
  else
    $m_a \leftarrow |A|/2; m_b \leftarrow |B|/2$;
    case $(A[m_a] < B[m_b], k > m_a + m_b)$ of
      $(T, T)$ ⇒ return kthHelp$(A[m_a + 1 : |A|], a_o + m_a + 1, B, b_o, k - m_a - 1)$
      $(T, F)$ ⇒ return kthHelp$(A, a_o, B[0 : m_b], b_o, k)$
      $(F, T)$ ⇒ return kthHelp$(A, a_o, B[m_b + 1 : |B|], b_o + m_b + 1, k - m_b - 1)$
      $(F, F)$ ⇒ return kthHelp$(A[0 : m_a], a_o, B, b_o, k)$

kth$(A, B, k)$ = return kthHelp$(A, 0, B, 0, k)$
```

Figure 6: The kth function.

This allows us to find the position in $O(\log_k |A|)$ rounds each requiring k comparisons. Figure 5 shows the pseudocode. Each round, given by findBlock, runs in constant span. By picking $k = n_0^\alpha$, where n_0 is the original input size, and for $0 < \alpha \leq 1$, the algorithm will make $O(1/\alpha)$ calls until it is of size 1 or less. The work on each recursive call is $O(k) = O(n_0^\alpha)$ so the total work is $W(n) = O((1/\alpha)n^\alpha)$. On the MP-RAM the span on each level is $O(1)$ so the overall span is $O(1/\alpha)$. If we pick $\alpha = 1/2$, for example, we get a constant span algorithm with $O(n^{1/2})$ work. If we pick $\alpha = 1/\log(n)$ we get $W(n) = S(n) = O(\log n)$ and we are back to a standard binary search. The multiway branching algorithm requires a k-way fork and is no better than binary search for the binary-forking model.

Another related problem is given two sorted sequences A and B, and an integer k, to find the k smallest elements. More specifically kth(A, B, k) returns locations (l_a, l_b) in A and B such that $l_a + l_b = k$, and all elements in $A[0 : l_a] \cup B[0 : l_b]$ are less than all elements in $A[l_a : |A|] \cup B[l_b : |B|]$. This can be solved using a dual binary search as shown in Figure 6. Each recursive call either halves the size of A or halves the size of B and therefore runs in in $O(\log |A| + \log |B|)$ work and span.

The dual binary search in Figure 6 is not parallel, but as with the sorted search problem it is possible to trade off work for span. Again the idea is to do a k-way search. By picking k evenly spaced positions in one array it is possible to find them in the other array using the sorted search problem. This can be used to find the sublock of A and B that contain the locations (l_a, l_b). By doing this again from the other array, both subblocks can be reduced in size by a factor of k. This is repeated for $\log_k |A| + \log_k |B|$ levels. By picking $k = n_0^\alpha$ this will result in an algorithm taking $O(n^{2\alpha})$ work and $O(1/\alpha^2)$ span. As with the constant span sorted array search problem, this does not work on the binary-forking model.

4.4 Merge

The merging problem is to take two sorted sequences A and B and produces as output a sequence R containing all elements of A and B in a stable, sorted order. Sequentially this is typically implemented by starting at the start of each sequence repeatedly pulling off the lesser element of the two remaining sequences, This algorithm is not parallel, but here we describe a few different parallel algorithms for the problem, which are also good sequential algorithms (i.e., they are work efficient).

Using the KTH function, merging can be implemented using divide-and-conquer as shown in Figure 7. The call to KTH splits the output size in half (within one), and then the merge recurses on the lower parts of A and B and in parallel on the higher parts. The updates to the output R are made in the base case of the recursion and hence the MERGE does not return anything. Letting $m = |A| + |B|$, and using the dual binary search for KTH the cost recurrences for MERGE are:

$$\begin{aligned} W(m) &= 2W(m/2) + O(\log m) \\ D(m) &= D(m/2) + O(\log m) \end{aligned}$$

solving to $W(m) = O(m)$ and $D(m) = O(\log^2 m)$. This works on the binary-forking model. By using the parallel version of KTH with $\alpha = 1/4$, the recurrences are:

$$\begin{aligned} W(m) &= 2W(m/2) + O(n^{1/2}) \\ D(m) &= D(m/2) + O(1) \end{aligned}$$

solving to $W(m) = O(m)$ and $D(m) = O(\log m)$. This does not work on the binary-forking model.

The span of parallel merge can be improved by using a multi-way divide-and-conquer instead of two-way, as showin in Figure 8. The code makes $f(n)$ recursive calls each responsible for a region of the output of size l. If we use $f(n) = \sqrt{n}$, and

```
MERGE$(A, B, R)$ =
  case $(|A|, |B|)$ of
    $(0, \_)$ ⇒ copy $B$ to $R$; return;
    $(\_, 0)$ ⇒ copy $A$ to $R$; return;
    otherwise ⇒
      $m \leftarrow |R|/2$;
      $(m_a, m_b) = $ KTH$(A, B, m)$;
      MERGE$(A[0 : m_a], B[0 : m_b], R[0 : m])$ ||
      MERGE$(A[m_a : |A|], B[m_b : |B|], R[m : |R|])$;
      return
```

Figure 7: 2-way D&C MERGE.

```
MERGEFWAY$(A, B, R, f)$ =
  // Same base cases
  otherwise ⇒
    $l \leftarrow (|R| - 1)/f(|R|) + 1$;
    parfor $i$ in $[0 : f(|R|)]$
      $s \leftarrow \min(i \times l, |R|)$;
      $e \leftarrow \min((i + 1) \times l, |R|)$;
      $(s_a, s_b) \leftarrow$ KTH$(A, B, s)$;
      $(e_a, e_b) \leftarrow$ KTH$(A, B, e)$;
      MERGEFWAY$(A[s_a : e_a], B[s_b : e_b], R[s : e], f)$
    return
```

Figure 8: $f(n)$-way D&C merge.

using dual binary search for KTH, the cost recurrences are:

$$\begin{aligned} W(m) &= \sqrt{m}\, W(\sqrt{m}) + O(\sqrt{m} \log m) \\ D(m) &= D(\sqrt{m}) + O(\log m) \end{aligned}$$

solving to $W(n) = O(n)$ and $D(n) = O(\log m)$. This version works on the binary-forking model since the PARFOR can be done with binary-forking in the required span. By using $f(n) = \sqrt{n}$ and the parallel version of KTH with $\alpha = 1/8$, the cost recurrences are:

$$\begin{aligned} W(m) &= \sqrt{m}\, W(\sqrt{m}) + O(m^{3/4}) \\ D(m) &= D(\sqrt{m}) + O(1) \end{aligned}$$

solving to $W(n) = O(n)$ and $D(n) = O(\log \log m)$.

Bound 4.1. *Merging can be solved in $O(n)$ work and $O(\log n)$ span in the binary-forking model and $O(n)$ work and $O(\log \log n)$ span on the MP-RAM.*

We note that by using $f(n) = n/\log(n)$, and using a sequential merge on the recursive calls gives another variant that runs with $O(n)$ work and $O(\log n)$ span on the binary-forking model. When used with a small constant, e.g. $f(n) = .1 \times n/\log n$, this version works well in practice.

4.5 K-th Smallest

The k-th smallest problem is to find the k-smallest element in an sequences. Figure 9 gives an algorithm for the problem. The performance depends on how the pivot is selected. If it is selected uniformly at random among the element of A then the algorithm will make $O(\log |A| + \log(1/\epsilon))$ recursive calls with probability $1 - \epsilon$. One way to analyze this is to note that with probability 1/2 the pivot will be picked in the middle half (between 1/4 and 3/4), and in that case the size of the array to the recursive call

```
KTHSMALLEST$(A, k)$ =
  $p \leftarrow$ SELECTPIVOT$A)$;
  $L \leftarrow$ FILTER$(A, \lambda x.(x < p))$;
  $G \leftarrow$ FILTER$(A, \lambda x.(x > p))$;
  if $(k < |L|)$ then
    return KTHSMALLEST$(L, k)$
  else if $(k > |A| - |G|)$ then
    return KTHSMALLEST$(G, k - (|A| - |G|))$
  else return $p$
```

Figure 9: KTHSMALLEST.

```
SELECTPIVOTR$(A) = A[$RAND$(n)]$;

SELECTPIVOTD$(A, l)$ =
  $l \leftarrow f(|A|)$;
  $m \leftarrow (|A| - 1)/l + 1$;
  $B \leftarrow$ ARRAY$[m]$;
  parfor $i$ in $[0 : m]$
    $s \leftarrow i \times l$;
    $B[i] \leftarrow$ KTHSMALLEST$(A[s : s + l], l/2)$;
  return KTHSMALLEST$(B, m/2)$;
```

Figure 10: Randomized and deterministic pivot selection.

be at most $3/4|A|$. We call such a call *good*. After at most $\log_{4/3}|A|$ good calls the size will be 1 and the algorithm will complete. Analyzing the number of recursive calls is the same as asking how many unbiased, independent, coin flips does it take to get $\log_{4/3}|A|$ heads, which is bounded as stated above.

In general we say an algorithm has some property *with high probability* (w.h.p.) if for input size n and any constant k the probability is at least $1 - 1/n^k$. Therefore the randomized version of KTHSMALLEST makes $O(\log|A|)$ recursive calls w.h.p. (picking $\epsilon = 1/|A|^k$). Since filter has span $O(\log n)$ for an array of size n, the overall span is $O(\log|A|^2)$ w.h.p.. The work is $O(|A|)$ in expectation. The algorithm runs on the binary-forking model.

It is also possible to make a deterministic version of KTHSMALLEST by picking the pivot more carefully. In particular we can use the median of median method shown in Figure 10. It partitions the array into blocks of size $f(|A|)$, finds the median of each, and then finds the median of the results. The resulting median must be in the middle half of values of A. Setting $f(n) = 5$ gives a parallel version of the standard deterministic sequential algorithm for KTHSMALLEST. Since the blocks are constant size we don't have to make recursive calls for each block and instead can compute each median of five by sorting. Also in this case the recursive call cannot be larger than $7/10|A|$. The parallel version therefore satisfies the cost recurrences:

$$\begin{aligned} W(n) &= W(7/10n) + W(1/5n) + O(n) \\ D(m) &= D(7/10n) + D(1/5n) + O(1) \end{aligned}$$

which solve to $W(n) = O(n)$ and $D(n) = O(n^\alpha)$ where $\alpha \approx .84$ satisfies the equation $\left(\frac{7}{10}\right)^\alpha + \left(\frac{1}{5}\right)^\alpha = 1$.

The span can be improved by setting $f(n) = \log n$, using a sequential median for each block, and using a sort to find the median of medians. Assuming the sort does $O(n \log n)$ work and has span $D_{sort}(n)$ this gives the recurrences:

$$\begin{aligned} W(n) &= W(3/4n) + O((n/\log n)\log(n/\log n)) + O(n) \\ D(m) &= D(3/4n) + O(\log n) + D_{sort}(n) \end{aligned}$$

	MP-RAM		Binary-Forking	
Problem	Work	Span	Work	Span
SCAN	$O(n)$	$O(\log n)$	—	—
FILTER	$O(n)$	$O(\log n)$	—	—
FLATTEN	$O(m+n)$	$O(\log n)$	—	$O(\log m)$
SEARCH	$O(n^\alpha/\alpha)$	$O(1/\alpha)$	$O(\log n)$	$O(\log n)$
MERGE	$O(n)$	$O(\log\log n)$	—	$O(\log n)$
KTHSMALLEST	$O(n)$	$O(D_s(n)\log\log n)$	—	$O(D_s(n)\log\log n)$

Table 1: Summary of costs. A — indicates that the cost is the same as on the MP-RAM. For FLATTEN, m is the size of the result. For SEARCH, $0 < \alpha \leq 1$. For KTHSMALLEST, $D_s(n)$ is the span for sorting a sequence of length n on the given model.

which solve to $W(n) = O(n)$ and $D(n) = O(D_{sort}(n)\log n)$. By stopping the recursion of KTHSMALLEST when the input reaches size $n/\log n$ (after $O(\log\log n)$ recursive calls) and applying a sort to the remaining elements improves the span to $D(n) = O(D_{sort}(n)\log\log n)$.

4.6 Summary

Table 1 summarizes the costs for the primitives we have covered so far.

5 Sorting

A large body of work exists on parallel sorting under different parallel models of computation. In this section, we present several classic parallel sorting algorithms such as mergesort, quicksort, samplesort and radix-sort. We also discuss related problems like semisorting and parallel integer sorting.

5.1 Mergesort

Parallel mergesort is a classic parallel divide-and-conquer algorithm. Pseudocode for a parallel divide-and-conquer mergesort is given in Figure 11. The algorithm takes an input array A, recursively sorts $A[0:m]$ and $A[m:|A|]$ and merges the two sorted sequences together into a sorted result sequence R. As both the divide and merge steps are stable, the output is stably sorted. We compute both recursive calls in parallel, and use the parallel merge described in Section 4.4 to merge the results of the two recursive calls. This gives the following recurrences for work and span:

$$W(n) = 2W(n/2) + W_{\text{MERGE}}(n)$$

$$D(n) = D(n/2) + D_{\text{MERGE}}O(n)\ .$$

```
MERGESORT($A$)
  if ($|A| = 1$) then return $A$;
  else
    $m \leftarrow |A|/2$
    $l \leftarrow$ MERGESORT($A[0 : m]$) ||
    $r \leftarrow$ MERGESORT($A[m : |A|]$)
    return MERGE($l, r$)
```

Figure 11: Parallel mergesort.

```
QUICKSORT($S$) =
  if ($|S| = 0$) then return $S$;
  else
    $p \leftarrow$ SELECTPIVOT($S$);
    $e \leftarrow$ FILTER($S, \lambda x.(x = p)$);
    $l \leftarrow$ QUICKSORT(FILTER($S, \lambda x.(x < p)$)) ||
    $r \leftarrow$ QUICKSORT(FILTER($S, \lambda x.(x > p)$))
    return FLATTEN($\langle l, e, r \rangle$)
```

Figure 12: Parallel quicksort.

Given the merging results from Section 4.4, these solve to $O(n \log n)$ work and $O(\log n \log \log n)$ span on the MP-RAM, and the same, optimal, work but $O(\log^2 n)$ span on the binary-forking model.

5.2 Quicksort

Pseudocode for a parallel divide-and-conquer quicksort is given in Figure 12. It is well known that for a random choice of pivots, the expected time for randomized quicksort is $O(n \log n)$. As the parallel version of quicksort performs the exact same calls, the total work of this algorithm is also $O(n \log n)$ in expectation. Here we briefly analyze the work and span of the algorithm. We assume pivots are selected at random with equal probability. The analyses for work is the same as in analyzing the sequential algorithm. To analyze span we note that the span of each call to quicksort (not including the recursive calls) is $O(\log n)$, so if we can bound the recursion depth, we can bound the span.

For an input sequence of length n let A_{ij} be a random variable indicating that when the element of rank i is selected as a pivot, the element with rank j is in the sequence S for that call to quicksort. Note that as the code is written, all elements are eventually selected as a pivot. Lets consider the probability of the event, i.e., $p(A_{ij} = 1)$. The event happens exactly when the element of rank i is selected first among the elements from rank i to rank j (inclusive). If any of the other elements between the elements ranked i and j are picked first they would split the two. If the element ranked j is picked first then clearly it cannot be in S when i is picked. If the element i is picked first, then j will be in S since no other element separated the two. There are $1 + |j - i|$ elements from i to j inclusive of both i and j, so the probability of the even is $1/(1 + |j - i|)$.

We first Let B be a random variable that specifies how many comparisons the sort does. We have:

$$B = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij}$$

By linearity of expectations we then have

$$
\begin{aligned}
E[B] &= \sum_{i=1}^{n}\sum_{j=1}^{n} E[A_{ij}] \\
&= \sum_{i=1}^{n}(\sum_{k=1}^{i-1}\frac{1}{k} + \sum_{k=1}^{n-i}\frac{1}{k}) \\
&< \sum_{i=1}^{n}(H_{i-1} + H_{n-i}) \\
&< \sum_{i=1}^{n} 2\ln n \\
&< 2n\ln n
\end{aligned}
$$

To analyze the the span we need to determine the depth of recursion. The depth of a rank i element is going to be the number of pivots selected above it. Let D_i be the number of such pivots, so we have that $D_i = \sum_{j=0}^{n} A_{ji}$ and

$$E[D_i] = \sum_{j=0}^{n} \frac{1}{1+|i-j|} < 2\ln n$$

This tells us the expected depth of an individual element but we care about the maximum depth over all elements. Unfortunately there is no equivalent of linearity-of-expectations for maximums. Instead we will use tail bounds to bound the probability that any element has depth higher than $O(\log n)$. In particular we will use the following Chernoff bounds.

Theorem 5.1 (Chernoff Bounds)**.** *Let $X_1, \ldots, X_n$ be independent random indicator variables, then:*

$$P[X \geq c] \leq \left(\frac{eE[X]}{c}\right)^c$$

Importantly note that our A_{ij} are independent across different i for any fixed j. This is because A_{ij} tells us a lot about i (it is the first picked in the range), but nothing about j other than it is picked after i. In particular, after i is picked it remains equally likely that any elements with ranks $i+1$ to j are picked next among them. We can therefore apply our Chernoff, setting $c = 2a\ln n$ for some constant a, giving

$$\begin{aligned} P[D_i \geq 2a \ln n] &\leq \left(\frac{eE[X]}{2a \ln n}\right)^{2a \ln n} \\ &\leq \left(\frac{e}{a}\right)^{2a \ln n} \\ &= n^{2a \ln(e/a)} \\ &= n^{-2a \ln(a/e)} \end{aligned}$$

For $a = e^2$, for example, we would have the probability less than n^{-2e^2}, which is less than n^{-14}. Even if we take a union bound over all n elements, we have the probability that any is deeper than $2e^2 \ln n$ is at most n^{-13}.

We therefore say randomized quicksort has recursion depth $O(\log n)$ with high probability, and hence span $O(\log^2 n)$ with high probability.

5.3 Sample Sort

Work in progress.

Practically, quicksort has high variance in its running time—if the choice of pivot results in subcalls that have highly skewed amounts of work the overall running time of the algorithm can suffer due to work imbalance. A practical algorithm known as *samplesort* deals with skew by simply sampling many pivots, called *splitters* ($c \cdot p$ or $\sqrt{n}$ splitters are common choices), and partitioning the input sequence into buckets based on the splitters. Assuming that we pick more splitters than the number of processors we are likely to assign a similar amount of work to each processor. One of the key substeps in samplesort is shuffling elements in the input subsequence into buckets. Either the samplesort or the radix-sort that we describe in the next section can be used to perform this step work-efficiently (that is in $O(n)$ work).

5.4 Counting and Radix Sort

Counting and radix sort, are integer-based sorts that are not comparison based and therefore are not constrained by the $O(n \log n)$ lower bound that comparison sorts have. Indeed, sequentially, assuming the integers are polynomially bounded in the number of keys n, radix sort based on counting sort takes $O(n)$ time.

For keys in a fixed range $[0 : m]$ counting sort works in three steps: first counting the number of keys have each possible value, then calculating the offsets at which each key appears in the ouput, and finally passing over the keys a second time to place them in their correct position. These corresponds to lines 13, 14, and 16 in Figure 13. For n integer keys in the range $[0 : m]$ counting sort requires $O(n + m)$ work (time). It is therefore not efficient when $m \gg n$. An important property of counting sort is that it is stable: equal keys maintain their order.

When $m > n$ we can apply radix sort, which simply makes repeated calls to counting sort. For a radix of size m, round i sorts based on extracting from each key k, a round key $(k/n^i) \mod n$. Since the round key is bounded by n, we can apply counting sort for the round in linear time. Since counting sort is stable, each round maintains the ordering with respect to previous rounds. If the input integers are bounded by n^c, radix sort takes c rounds and hence $O(cn)$ time. Assuming c is a constant, this gives $O(n)$ time.

Interestingly whether it is possible to sort integers in the range $[0 : n^c], c > 1$ in $O(n)$ work and polylogarithmic span is an open problem. What we will show now is that it is reasonably easy to do radix sort in parallel with $O\left(\frac{n}{\alpha}\right)$ work and $O\left(\frac{n^\alpha}{\alpha}\right)$ span for any $\alpha, 0 < \alpha \leq 1$. The span is not polylogarithmic.

We start by considering a parallel counting sort that sorts n integers in the range $[0 : m]$. The sort is given by PARALLELCOUNTSORT in Figure 13. It is similar to the sequential version, but works by breaking the input into blocks of size $b = n/m$. Each block does its own count of its keys. We then need to calculate the offset in the output for each block and each key for where the results should be placed. Like the sequential case this involves a PLUSSCAN, but it is across all blocks and buckets. Finally given the offsets, we again work across the blocks in parallel to place each value in the correct position.

To analyzing the costs we note that the functions COUNTS and PLACE are fully sequential. When called on m keys they both take $O(m)$ work and span. Across the $|K|/m$ parallel calls to them this comes to adds to $O(|K|)$ work and $O(m)$ span. The cost of the PLUSSCAN, flattening, and transposing is $O(|K|)$ work and $O(\log |K|)$ span, for a total of $O(|K|)$ work and $O(m + \log |K|)$ span. For integers in the range $[0 : \log |K|]$ this is great, with linear work and logarithmic span. However for a range $[0 : n]$, it is completely sequential with $O(n)$ work and span.

To uses this parallel counting sort in a radix sort on n integers, we use subkeys in the range $[0 : n^\alpha], 0 < \alpha \leq 1$. This leads to $O(n)$ work and $O(n^\alpha)$ span per call to counting sort. For integers in the range $[0 : n^c]$ there will be c/α calls to counting sort for a total of $O(n/\alpha)$ work, and $O((n^\alpha + \log n)/\alpha) \subset O(n^\alpha/\alpha)$ span, assuming c is constant.

For integers $[0 : n \log^k n)]$ the best existing work-efficient integer sorting algorithm can unstably sort integers in $O(kn)$ work in expectation and $O(k \log n)$ span with high probability [21]. However, since it is unstable it cannot be used in a radix sort.

5.5 Semisort

Given an array of keys and associated records, the semisorting problem is to compute a reordered array where records with identical keys are contiguous. Unlike the output of a sorting algorithm, records with distinct keys are not required to be in sorted order. Semisorting is a widely useful parallel primitive, and can be used to implement the shuffle-step in MapReduce, compute relational joins and efficiently implement parallel graph algorithms that dynamically store frontiers in buckets, to give a few applications.

```
counts($K, m$)
  for $j \in [0 : m]$ : $C_j \leftarrow 0$
  for $j \in [0 : m]$
    $k \leftarrow K_j$
    $C_k \leftarrow C_k + 1$
  return $C$

place($R, O, K, V, m$)
  for $j \in [0 : m]$
    $k \leftarrow K_j$
    $R[O_k] \leftarrow V_j$
    $O_k \leftarrow O_k + 1$

sequentialCountSort($K, V, m$)
  $C \leftarrow$ counts($K, m$)
  $O \leftarrow$ plusScan($C$)
  $R \leftarrow$ an array of size $|K|$
  place($R, O, K, V, m$)
  return $R$

parallelCountSort($K, V, m$)
  $b \leftarrow |K|/m$
  parfor $i \in [0 : b]$
    $C_i =$ counts($K[mi : m(i+1)], m$)
  $O \leftarrow$ plusScan(flatten(transpose($C$)))
  $O' \leftarrow$ transpose(partition $O$ into blocks of size $b$)
  $R \leftarrow$ an array of size $|K|$
  parfor $i \in [0 : b]$
    place($R, O'_i, K[mi : m(i+1)], V[mi : m(i+1)], m$)
  return $R$
```

Figure 13: A parallel counting sort. K is a sequence of keys in the range $[0 : m]$ and V is a sequence of values of the same length as K. The output are the values from V sorted by their corresponding keys in K.

Gu, Shun, Sun and Blelloch [14] give a recent algorithm for performing a top-down parallel semisort. The specific formulation of semisort is as follows: given an array of n records, each containing a key from a universe U and a family of hash functions $h : U \rightarrow [1, \ldots, n^k]$ for some constant k, and an equality function on keys, $f : U \times U \rightarrow$ bool, return an array of the same records s.t. all records between two equal records are other equal records. Their algorithms run in $O(n)$ expected work and space and $O(\log n)$ span w.h.p. on the TS-MP-RAM.

6 Graph Algorithms

In this section, we present parallel graph algorithms for breadth-first search, low-diameter decomposition, connectivity, maximal independent set and minimum span-

```
edgeMap$(G, U, f_u) =$
    parfor $i \in [0 : |U|]$
        $N[i] \leftarrow \{v \in N^+(G, v) \mid f_u(u, v)\}$
    return flatten$(N)$
```

Figure 14: edgeMap.

ning tree which illustrate useful techniques in parallel algorithms such as randomization, pointer-jumping, and contraction. Unless otherwise specified, all graphs are assumed to be directed and unweighted. We use $deg_-(u)$ and $deg_+(u)$ to denote the in and out-degree of a vertex u for directed graphs, and $deg(u)$ to denote the degree for undirected graphs.

6.1 Graph primitives

Many of our algorithms map over the edges incident to a subset of vertices, and return neighbors that satisfy some predicate. Instead of repeatedly writing code performing this operation, we express it using an operation called edgeMap in the style of Ligra [24].

edgeMap takes as input U, a subset of vertices and update, an update function and returns an array containing all vertices $v \in V$ s.t. $(u, v) \in E, u \in U$ and update$(u, v) =$ `true`. We will usually ensure that the output of edgeMap is a set by ensuring that a vertex $v \in N(U)$ is atomically acquired by only one vertex in U. We give a simple implementation for edgeMap based on flatten in Figure 14. The code processes all $u \in U$ in parallel. For each u we filter its out-neighbors and store the neighbors v s.t. update$(u, v) =$ `true` in a sequence of sequences, `nghs`. We return a flat array by calling flatten on `nghs`. It is easy to check that the work of this implementation is $O(|U| + \sum_{u \in U} deg_+(u))$ and the depth is $O(\log n)$.

We note that the flatten-based implementation given here is probably not very practical; several papers [7, 24] discuss theoretically efficient and practically efficient implementations of edgeMap.

6.2 Parallel breadth-first search

One of the classic graph search algorithms is breadth-first search (BFS). Given a graph $G(V, E)$ and a vertex $v \in V$, the *BFS* problem is to assign each vertex reachable from v a parent s.t. the tree formed by all $(u, \texttt{parent}[u])$ edges is a valid BFS tree (i.e. any non-tree edge $(u, v) \in E$ is either within the same level of the tree or between consecutive levels). BFS can be computed sequentially in $O(m)$ work [13].

We give pseudocode for a parallel algorithm for BFS which runs in $O(m)$ work and $O(\mathsf{diam}(G) \log n)$ depth on the TS-MP-RAM in Figure 15. The algorithm first creates an initial frontier which just consists of v, initializes a visited array to all 0, and a parents array to all -1 and marks v as visited. We perform a BFS by looping while

```
BFS($G(V,E), v$) =
  $F \leftarrow \{v\}$;
  parfor $v \in V$ :
    $X_v \leftarrow$ false
    $P_v \leftarrow v$
  $X_v \leftarrow$ true
  while ($|F| > 0$)
    $F \leftarrow$ edgeMap($G, F, \lambda(u,v)$.
          if (testAndSet($X_v$))
            $P_v \leftarrow u$;
            return true
          return false
  return $P$
```

Figure 15: Parallel breadth-first search.

the frontier is not empty and applying edgeMap on each iteration to compute the next frontier. The update function supplied to edgeMap checks whether a neighbor v is not yet visited, and if not applies a test-and-set. If the test-and-set succeeds, then we know that u is the unique vertex in the current frontier that acquired v, and so we set u to be the parent of v and return true, and otherwise return false.

6.3 Low-diameter decomposition

Many useful problems, like connectivity and spanning forest can be solved sequentially using breadth-first search. Unfortunately, it is currently not known how to efficiently construct a breadth-first search tree rooted at a vertex in $\mathsf{polylog}(n)$ depth on general graphs. Instead of searching a graph from a single vertex, like BFS, a low-diameter decomposition (LDD) breaks up the graph into some number of connected clusters s.t. few edges are cut, and the internal diameters of each cluster are bounded (each cluster can be explored efficiently in parallel). Unlike BFS, low-diameter decompositions can be computed efficiently in parallel, and lead to simple algorithms for a number of other graph problems like connectivity, spanners, hop-sets, and low stretch spanning trees.

A (β, d)-*decomposition* partitions V into clusters, $V_1, \ldots, V_k$ s.t. the shortest path between two vertices in V_i using only vertices in V_i is at most d (strong diameter) and the number of edges (u, v) where $u \in V_i, v \in V_j, j \neq i$ is at most βm. Low-diameter decompositions (LDD) were first introduced in the context of distributed computing [5], and were later used in metric embedding, linear-system solvers, and parallel algorithms.

Sequentially, LDDs can be found using a simple sequential ball-growing technique [5]. The algorithm repeatedly picks an arbitrary uncovered vertex v and grows a ball around it using BFS until the number of edges incident to the current frontier is at most a β fraction of the number of internal edges. It then removes the ball. Because of the stopping condition at most a β fraction of the edges will leave a ball. Also since each level of the BFS will grow the number of edges by a factor of $1 + \beta$, the number of levels of the BFS is bounded by $O(\log_{(1+\beta)} m) = O(\log n/\beta)$. Therefore the algorithm gives a $(\beta, O(\log n/\beta)$-decomposition. As each edge is examined once, the algorithm does $O(n+m)$ work, but since the balls are visited one by one, in the worst case the algorithm is fully sequential.

We will now discuss a work-efficient parallel algorithm [18]. As with the sequential algorithm it involves growing balls around vertices, but in this case in parallel. Starting to grow them all at the same time does not work since it would make every vertex its own cluster. Instead the algorithm grows the balls from vertices based on start times that are randomly shifted based on the exponential distribution. The balls grow at the rate of one level (edge) per unit time, and each vertex is assigned to the first ball that hits it. As in the sequential case, we can grow the balls using BFS.

Recall that a non-negative continuous random variable X has an exponential distribution with rate parameter $\beta > 0$ if

$$\mathbf{Pr}[X = x] = \beta e^{-\beta x} \ .$$

By integration, the cumulative probability is:

$$\mathbf{Pr}[X \leq x] = 1 - e^{-\beta x} \ .$$

An important property of the exponential distribution is that it is *memoryless*, i.e.,

$$\mathbf{Pr}[X > a + b \mid X > a] = \mathbf{Pr}[X > b]$$

In words this says that if we take the tail of the distribution past a point a, and scale it so the the total remaining probability is 1 (i.e., conditioned on $X > a$), then the remaining distribution is the same as the original (it has forgotten that anything happened).

To select the start times each vertex v selects δ_v from an exponential distribution with parameter β, and then each uses a start time $T_v = \delta_{\max} - \delta_v$, where $\delta_{\max} = \max_{v \in V} \delta_v$. Note that the exponential distribution is subtracted, so the start times have a backwards exponential distribution, increasing instead of decreasing over time. In particular very few vertices will start in the first unit of time (often just one), and an exponentially growing number will start on each later unit of time. Based on these start times the first ball that hits a vertex u, and hence will be the cluster u is assigned to, is

$$C_u = \arg\min_{v \in V}(T_v + d(u, v))$$

This algorithm can be implemented efficiently using simultaneous parallel breadth-first searches. The initial breadth-first search starts at the vertex with the largest start time, $\delta_{\max}$. Each $v \in V$ "wakes up" and starts its BFS if $\lfloor T_v \rfloor$ steps have elapsed and it is not yet covered by another vertex's BFS. Figure 16 gives pseudocode for the algorithm. The clusters assignment for each vertex, C_v, is initially set to *unvisited*, and

```
1   parfor $v \in V$ : $\delta_v \leftarrow Exp(\beta)$;
2   $\delta_{\max} = \max_{v \in V} \delta_v$;
3   parfor $v \in V$ :
4         $T_v \leftarrow \delta_{\max} - \delta_v$;
5         $C_v \leftarrow$ unvisited;
6         $\gamma_v \leftarrow \infty$
7   $l \leftarrow 0$;
8   $r \leftarrow 1$;
9   while $(l < |V|)$
10     $F \leftarrow F \cup \{v \in V \mid (T_v < r) \wedge (C_v =$ unvisited$)\}$;
11     $l \leftarrow l + |F|$;
12     edgeMap$(G, F, \lambda(u, v)$.
13        if $(C_v =$ unvisited$)$
14           $c \leftarrow C_u$;
15           writeMin$(\gamma_v, T_c - \lfloor T_c \rfloor))$
16     $F \leftarrow$ edgeMap$(G, F, \lambda(u, v)$.
17           $c \leftarrow C_u$;
18           if $(\gamma_v = T_c - \lfloor T_c \rfloor)$
19              $C_v \leftarrow c$;
20              return true
21           return false)
22     $r \leftarrow r + 1$
23  return $C$
```

Figure 16: Low-diameter decomposition.

once set by the first arriving ball, it does not change. The γ_v variables are where the balls compete to see which has the earliest T_v—we only need the fractional part of T_v since only those that arrive on the same round compete. A simpler implementation is to arbitrarily take one of the balls that arrive on the first round on which any arrive. It turns out this maintains the desired properties within a constant factor [25].

To analyze the work and span of the algorithm, we note the number of rounds of the algorithm is bounded by $\lceil \delta_{\max} \rceil$ since every vertex will have been processed by then. Line 10 can be implemented efficiently by pre-sorting the vertices by the round they start on, just pulling from the appropriate bucket on each round. A counting sort can be used giving linear work and $O(\log |V| + \delta_{\max})$ span. Every vertex is in the frontier at most once, and hence every edge is visited at most once, so the total work is $O(|V| + |E|)$. Each round of the the algorithm has span at most $O(\log |V|)$, so the total span is $O(\delta_{\max} \log |V|)$.

We now analyze the radius of clusters and number of edges between them. We first argue that the maximum radius of each ball is $O(\log n/\beta)$ w.h.p. We already mentioned that the number of rounds of the LDD algorithm is bounded by $\delta_{\max}$. This also bounds the radius of any BFS and therefore the diameter is bounded by $2\delta_{\max}$. To bound $\delta_{\max}$, consider the probability that a single vertex picks a shift larger than

$\frac{c \log n}{\beta}$:

$$\mathbf{Pr}\left[\delta_v > \frac{c \log n}{\beta}\right] = 1 - \mathbf{Pr}\left[\delta_v \leq \frac{c \log n}{\beta}\right] = 1 - (1 - e^{-c \log n}) = \frac{1}{n^c}$$

Now, taking the union bound over all n vertices, we have that the probability of any vertex picking a shift larger than $\frac{c \log n}{\beta}$ is:

$$\mathbf{Pr}\left[\delta_{\max} > \frac{c \log n}{\beta}\right] \leq \frac{1}{n^{c-1}}$$

which implies that $\delta_{\max} \in O(\log n/\beta)$ w.h.p.

We now argue that at most βm edges are cut in expectation. Our first step will be to show that it is unlikely that the first k balls that cover a vertex or the midpoint of an edge all arrive in a small interval of time. By midpoint of the edge we mean when the ball covers half the edge from one side—imagine a vertex in the middle of the edge with distance .5 to each endpoint.

Lemma 6.1. *For any vertex or midpoint of an edge u, the probability that the smallest and k-th smallest value from $\{T_v + d(v,u) : v \in V\}$ differ by less than a is at most $\left(1 - e^{-\beta a}\right)^{k-1}$.*

Proof. We consider the ball growing running backwards in time, and hence balls will shrink and uncover points over time. Recall that since we subtract the exponential distributions δ_v to get the start time, they are proper exponential distributions going backwards in time. Consider the time t when the k-th latest ball (in backwards time) uncovers u. At this point there are $k-1$ balls still covering u. Due to the memoryless property of exponentials, they each have an exponential distribution starting at t. The probability that any one of them is removed in the next a time, again backwards, is given by the cumulative probability distribution $1 - e^{-\beta a}$.

We thus have that the probability that any of the remaining balls is removed in a time is less than $a\beta$. Now each of the $k-1$ remaining balls chose its start time independently, so the probability of them all being removed in a time is the product of these probabilities, giving the claimed result. □

We can bound $\left(1 - e^{-\beta a}\right)^{k-1} < (\beta a)^{k-1}$ using the inequality $1 - e^{-\alpha} < \alpha$ for $\alpha > 0$, which can be derived by taking the first two terms of the Taylor series of e^{-x} and noting that the rest of the terms sum to a positive value.

Finally we argue that an edge only bridges between clusters if the first two balls (i.e., $k = 2$) arrive at its midpoint within one unit of time (i.e., $a = 1$). In particular if the first ball arrives at time t (we are talking forward time now), and the next at time greater than $t + 1$, then no other ball can reach the edge's endpoints until after time $t + 1/2$. At this point, the first ball has already claimed both endpoints so they will belong to the same cluster. Thus by Lemma 6.1 the probability that the edge is bridges

```
Connectivity($G(V,E),\beta$) =
  $L \leftarrow$ LDD($G,\beta$);
  $G'(V',E')$ = Contract($G,L$);
  if ($|E'| = 0$)
    return $L$
  $L' \leftarrow$ Connectivity($G',\beta$)
  parfor $v \in V$ :
    $u \leftarrow L_v$;
    $L''_v \leftarrow L'_u$
  return L''
```

Figure 17: Parallel connectivity.

between clusters is upper bounded by $(1\beta)^{2-1} = \beta$. Note that we did not use the full generality of the Lemma, but will when discussing graph spanners. All together, we therefore have the following theorem.

Theorem 6.1. *For an undirected graph $G = (V,E)$ and for for any $\beta > 0$ there is a randomized parallel algorithm for finding a $\left(\beta, O\left(\frac{\log|V|}{\beta}\right)\right)$-decomposition of the graph in $O(|V| + |E|)$ work, and $O\left(\frac{\log^2|V|}{\beta}\right)$ span.*

Proof. We just showed that the probability that an edge goes between clusters is at most β. By linearity of expectation across edges this means the expected number of edges between components is at most $\beta|E|$. We showed above that $\delta_{\max}$ and hence the diameter of the components are bounded by $O\left(\frac{\log|V|}{\beta}\right)$ w.h.p.. Plugging $\delta_{\max}$ into the cost analysis of the LDD algorithm gives the work and span bounds. □

6.4 Connectivity

6.5 Spanners

A subgraph H of $G = (V,E)$ is a k-spanner of G if for all pairs of vertices $u, v \in V$, $d_H(u,v) \leq k d_G(u,v)$. In particular the the spanner preserves distances within a factor of k. There a conjectured lower-bound that states that in general a spanner H must have $\Omega(n^{1+1/\lceil k/2 \rceil})$ edges [20]. The lower bound is based on the Erdos Girth Conjecture. The girth of a graph is the size of its smallest cycle, and the conjecture states that the maximum number of possible edges in a graph of girth k is $\Omega(n^{1+1/\lceil k/2 \rceil})$. An unweighted graph of with smallest cycle $> k+1$ cannot have a k-spanner since cutting any edge would distort the weight on it cycles by a factor greater than k. Therefore the girth conjecture, if true, implies the lower bound.

There are many sequential and parallel that achieve $O(k)$-spanners with $O(n^{1+1/k})$ edges, with various tradeoffs [4]. Here we describe a simple parallel algorithm that uses low-diameter-decomposition (LDD) [17]. We just describe a variant for unweighted graphs, but the ideas can be extended to weighted graphs [17].

The idea is simply to run a low-diameter decomposition with parameter $\beta = \ln n/2k$. For each cluster generated by the LDD we add the edges of the BFS tree for that cluster to our spanner. We also add an edge for every pair of clusters that has at least one edge between them. Note that it is important that if there are multiple edges between a pair of clusters, we only keep one of them—it does not matter which one. The claim is that with high probability the spanner constructed in this way will preserve shortest path distances within $O(k)$ while having at most $O(n^{1+1/k})$ edges in expectation.

First we argue that the graph preserves distances. In particular we show that every edge has a stretch of at most $O(k)$ with high probability (i.e., if we remove it there is another path of length $O(k)$ between its endpoints). Firstly we note that by the analysis of LDD, each cluster have diameter $O(\ln n/\beta)$ with high probability. Substituting our value of β we have the diameter is $O(k)$ with high probability. Now note that edges either are within a cluster or go between clusters. Since the longest path using the BFS tree is bounded by twice the diameter, any edge within a cluster can use the BFS tree and hence have stretch bounded by is $O(k)$. Now consider an edge between clusters, and in particular one we did not keep. Since there is another edge between the clusters we can take a path within the cluster on one endpoint to that edge, then across the edge, and then within the other cluster to the other endpoint. This path also has length $O(k)$.

We now argue that the number of edges is bounded by $O(n^{1+1/k})$ in expectation. In particular Consider a vertex on the boundary of a cluster. It could have edges to some number of adjacent clusters. Note that to have edges to l neighboring clusters, the smallest and l^{th} smallest value to arrive at any of them must differ by at most two. Lemma 6.1 showed that the probability of this is at most $\left(1 - e^{-\beta a}\right)^{l-1} = \left(1 - e^{-2 \lg n/2k}\right)^{l-1} = \left(1 - n^{-1/k}\right)^{l-1}$. Consider the random variable L which is the number of neighboring clusters for the vertex. To calculate the expected value of L we can sum across possible values of l, giving :

$$E[L] = \sum_{l=1}^{\infty} P(L \geq l) \leq \sum_{l=1}^{\infty} \left(1 - n^{-1/k}\right)^{l-1}$$

Using the identity that $\sum_{i=0}^{\infty} a^i = 1/(1-a)$ for $a < 1$ we have that $E[L] \leq n^{1/k}$. If we multiply across all vertices we have that the expected bound on the total number of cross cluster edges is $O(n^{1+1/k})$ as required.

6.6 Maximal Independent Set

In a graph $G = (V, E)$ we say a set of vertices $U \subset V$ is an *independent set* if none of them are neighbors in G (i.e., $(U \times U) \cap E = \emptyset$). A set of vertices $U \subset V$ is a *maximal independent set* (MIS) if it is an independent set and no vertex in $V \setminus U$ can be added

$N(G, v), N^-(G, v), N^+(G, v)$	neighbors of v in G ($+ =$ out, $- =$ in)
$N(G, V), N^-(G, V), N^+(G, V)$	$N(G, V) = \bigcup_{v \in V} N(G, v)$ (similarly for N^- and N^+)

Figure 18: Some notation we use for graphs.

```
$\textsc{LubyMIS}(G = (V, E)) =$
  if $(|V| = 0)$ return $\{\}$
  parfor $v \in V$ : $\rho_v \leftarrow$ random priority
  $I \leftarrow \{v \in V \mid \rho(v) < \min_{u \in N(G,v)} \rho(u)\}$
  $V' \leftarrow V \setminus (I \cup N(G, I))$
  $G' \leftarrow (V', \{(u, v) \in E \mid (u \in V' \wedge v \in V')\})$
  $R \leftarrow \textsc{LubyMIS}(G')$
  return $I \cup R$
```

Figure 19: Luby's algorithm for MIS.

to U such that the result is an independent set. The MIS problem is to find an MIS in a graph. Sequentially it is quite easy. Just put the vertices in arbitrary order, and then add them one by one, such that whenever a vertex is added we remove all its neighbors from consideration. This is a seemingly sequential process.

Here we describe Luby's algorithm for finding an MIS efficiently in parallel. Pseudocode is given in Figure 20. The algorithm works in a sequence of recursive calls. On each call in parallel it assigns the remaining vertices a random priority. Then it identifies the set of vertices I that are a local maximum with respect to the priority (i.e. all of their neighbors have lower priorities). The set I is an independent set since no two neighbors can both be a local maximum, so the algorithm will add it to its result (the last line). However, it is not necessarily a maximal independent set. To identify more vertices to add to the MIS we remove the vertices I and their neighbors from the graph, along with their incident edges. We know the the neighbors $N(G, I)$ cannot be in an independent set with I. Now the algorithms recurses on the remaining graph.

We can argue the algorithm generates an MIS by induction. Inductively assume it is true on the smaller recursive call. The base case is true since an empty graph has no vertices to place in an MIS. For any other call we know I is independent, and we know that given I is in the MIS, then no vertices in $N(G, I)$ can be. Any remaining vertex in V' is independent of I. Since by induction the recursive call returned an MIS, there are no additional vertices we can add to R such that the set will still be independent, and there are also none from $N(G, I)$, showing that $I \cup R$ is independent and maximal.

Proving the overall cost bounds is a bit trickier. What we will show is that we expect to remove at least half the edges in each round (each recursive call). The proof is interesting because when adding a vertex to the MIS we will not be counting the removal of its edges, but rather the removal of some of the edges of its neighbors. The purpose of this approach is to avoid multiple counting. It turns out that the expected fraction of vertices that are kept on a round might be larger than 1/2—the actual

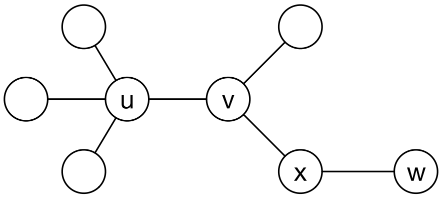


Figure 20: Example for the proof of bounds for Luby's algorithm.

number depends on the particular graph.

Theorem 6.2. *One round of Luby's algorithm removes at least half the edges in expectation.*

Proof. Consider an edge (u, v). Let $A_{u,v}$ be the event that u has the largest priority among u, v, and all their neighbors. Since priorities are selected uniformly at random, to determine the probability of this event we just have to count the number of such vertices, which gives:

$$\mathbf{Pr}[A_{u,v}] \geq \frac{1}{d(u) + d(v)}$$

It is an inequality since u and v might share neighbors. Figure 20 illustrates an example where $\mathbf{Pr}[A_{u,v}] = 1/(d(u) + d(v)) = 1/(4 + 3) = 1/7$. When the event $A_{u,v}$ occurs, u will necessarily be selected to be in the MIS, and all the edges incident on v will be removed since v is a neighbor of u and will itself be removed. Therefore we expect to remove $\mathbf{Pr}[A_{u,v}]d(v)$ neighbors of v due to the event $A_{u,v}$. Note that the edges incident on u will also be removed, but we are not going to count these—except for (u, v) since it is a neighbor of v.

Now note that when we have an event $A_{u,v}$ we cannot on the same round have another event $A_{w,v}$ since that would imply that both $\rho(u) > \rho(w)$ and $\rho(w) > \rho(u)$, which is not possible. Therefore if we count the edges incident on v removed by $A_{u,v}$, we will not double count them due to another simultaneous event $A_{w,v}$. However since an edge has two endpoints, an edge can still be counted twice, once from each of its endpoints. In Figure 20, for example, we could simultaneously have the events $A_{u,v}$ and $A_{w,x}$, which would each count the edge v, x in edges it removes.

Let Y be a random variable giving the number of edges we remove in expectation. We claim the following is an expression for the expectation of Y.

$$E[Y] \geq \frac{1}{2} \sum_{(u,v) \in E} \left(\mathbf{Pr}[A_{u,v}]d(v) + \mathbf{Pr}[A_{v,u}]d(u) \right)$$

We get this by adding up the contribution of removed edges for both directions of the edge (u, v) and accounting for the double counting when removing an edge from each of its endpoints (i.e., the 1/2 factor). Each term of the sum is at least 1, giving a total $E[Y] \geq |E|/2$, which is what we were aiming to prove. □

Now we consider the overall cost. Setting the priorities, taking the minimum of the neighbors to determine I, and generating the new graph can all be done with $O(|V| + |E|)$ work and $O(\log |V|)$ span. Hence one call to LubyMIS without the recursive call, uses $O(|V|+|E|)$ work and $O(\log |V|)$ span. Intuitively since the number of edges goes down by at least 1/2 in expectation, and a vertex will be removed if it has no incident edges, the size of the graph will decrease geometrically. This would give $O(|V| + |E|)$ work, $O(\log |E|)$ rounds and hence $O(\log^2 |V|)$ span. However since it is a random process we need to be more careful.

Let F_i be a random variable for the fraction removed on step i, and let M_i be the number of edges remaining on step i, i.e. $M_i = m\Pi_0^{i-1} F_i$. Lets say that we have that $E[F_i] \leq \beta$ (for Luby's algorithm $\beta = 1/2$). Furthermore since we are selecting priorities at random in each round, this inequality holds even if conditioned on all outcomes of all previous recursive calls. Let $\bar{E}[F_i] = \beta$ be the upper bound on the conditioned expectation. Given that these upper bounds are independent of previous F_i, we can take the product of these bounded expectations to bound the expectation of the product of the F_i, giving:

$$E[M_i] \leq m \prod_{j=0}^{i-1} \bar{E}[F_i] = m\beta^i$$

If the work on one round i is cM_i (assuming $|E_i| \geq |V_i|$), then by linearity of expectations we have:

$$E[W(m)] \leq \sum_{i=0}^{\infty} cE[M_i] \leq cm \sum_{i=0}^{\infty} \beta^i = O(m)$$

The work on the vertices is bounded by the work on the edges.

To bound the span we use Markov's inequality:

$$\mathbf{Pr}[X \geq a] \leq \frac{E[X]}{a}$$

and consider the probability $\mathbf{Pr}[M_i \geq 1]$. Note that if $M_i < 1$ then the algorithm finishes since there are no edges left. So we just want to show that for sufficiently large i the probability is small. We have that:

$$\mathbf{Pr}[M_i \geq 1] \leq \frac{E[M_i]}{1} \leq m\beta^i$$

Setting $i = k \lg m$ for some k, we have:

$$\mathbf{Pr}[M_i \geq 1] \leq m\beta^{k \lg m} = mm^{k \lg \beta} = m^{1+k \log \beta}$$

In our case $\beta = 1/2$, so $\lg \beta = -1$, giving :

$$\mathbf{Pr}[\text{rounds} < k \lg m] \geq 1 - m^{1-k}$$

This implies that the number of rounds is $O(\log m)$ (and hence also $O(\log n)$) w.h.p.

7 Parallel Binary Trees

In this section, we present some parallel algorithms for balanced binary trees. The methodology in this section is based on an algorithmic framework called *join*-based algorithms [8]. *join* is a primitive defined for each balancing scheme. All the other tree algorithms deal with rebalancing and rotations through *join*, and thus can be generic in code across multiple balancing schemes.

The function $join(T_L, e, T_R)$ for a given balancing scheme takes two balanced binary trees T_L, T_R balanced by that balancing scheme, and a single entry e as inputs, and returns a new valid balanced binary tree, that has the same entries and the same in-order traversal as $node(T_L, e, T_R)$, but satisfies the balancing criteria. We call the middle entry e the *pivot* of the *join*.

7.1 Preliminaries

A *binary tree* is either a *nil-node*, or a node consisting of a *left* binary tree T_l, an entry e, and a *right* binary tree T_r, and denoted $node(T_l, e, T_r)$. The entry can be simply a key, or a key-value pair. The *size* of a binary tree, or $|T|$, is 0 for a *nil-node* and $|T_l| + |T_r| + 1$ for a $node(T_l, e, T_r)$. The *weight* of a binary tree, or $w(T)$, is one more than its size (i.e., the number of leaves in the tree). The *height* of a binary tree, or $h(T)$, is 0 for a *nil-node*, and $\max(h(T_l), h(T_r)) + 1$ for a $node(T_l, e, T_r)$. *Parent*, *child*, *ancestor* and *descendant* are defined as usual (ancestor and descendant are inclusive of the node itself). A node is called a *leaf* when its both children are *nil-nodes*. The *left spine* of a binary tree is the path of nodes from the root to a leaf always following the left tree, and the *right spine* the path to a leaf following the right tree. The *in-order values* (also referred to as the symmetric order) of a binary tree is the sequence of values returned by an in-order traversal of the tree. When the context is clear, we use a node u to refer to the subtree T_u rooted at u, and vice versa.

A *balancing scheme* for binary trees is an invariant (or set of invariants) that is true for every node of a tree, and is for the purpose of keeping the tree nearly balanced. In this section we consider four balancing schemes that ensure the height of every tree of size n is bounded by $O(\log n)$.

AVL Trees [3]. AVL trees have the invariant that for every $node(T_l, e, T_r)$, the height of T_l and T_r differ by at most one. This property implies that any AVL tree of size n has height at most $\log_\phi(n + 1)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Red-black (RB) Trees [6].. RB trees associate a color with every node and maintain two invariants: (the red rule) no red node has a red child, and (the black rule) the number of black nodes on every path from the root down to a leaf is equal. All *nil-nodes* are always black. Unlike some other presentations, we do not require that the root of a tree is black. Although this does not affect the correctness of our algorithms, our proof of the work bounds requires allowing a red root. We define the *black height* of a node T, denoted $\hat{h}(T)$ to be the number of black nodes on a downward path from

Notation	Description
$\|T\|$	The size of tree T
$h(T)$	The height of tree T
$\hat{h}(T)$	The black height of an RB tree T
$w(T)$	The weight of tree T (i.e, $\|T\|+1$)
$p(T)$	The parent of node T
$k(T)$	The key of node T
$lc(T)$	The left child of node T
$rc(T)$	The right child of node T
$\texttt{expose}(T)$	$\langle lc(T), k(T), rc(T)\rangle$

Table 2: **Summary of notation**.

the node to a leaf (inclusive of the node). Any RB tree of size n has height at most $2\log_2(n+1)$.

Weight-balanced (WB) Trees. WB trees are defined with parameter α (also called BB[α] trees) [19] maintain for every $T = node(T_l, e, T_r)$ the invariant $\alpha \leq \frac{w(T_l)}{w(T)} \leq 1-\alpha$. We say two weight-balanced trees T_1 and T_2 have *like* weights if $node(T_1, e, T_2)$ is weight balanced. Any α weight-balanced tree of size n has height at most $\log_{\frac{1}{1-\alpha}} n$. For $\frac{2}{11} < \alpha \leq 1 - \frac{1}{\sqrt{2}}$ insertion and deletion can be implemented on weight balanced trees using just single and double rotations [19, **?**]. We require the same condition for our implementation of *join*, and in particular use $\alpha = 0.29$ in experiments. We also denote $\beta = \frac{1-\alpha}{\alpha}$, which means that either subtree could not have a size of more than β times of the other subtree.

Treaps. [22] Treaps associate a uniformly random priority with every node and maintain the invariant that the priority at each node is no greater than the priority of its two children. Any treap of size n has height $O(\log n)$ with high probability (w.h.p).

The notation we use for binary trees is summarized in Figure 2.

7.2 The *join* Algorithms for Each Balancing Scheme

Here we describe algorithms for *join* for the four balancing schemes we defined in Chapter 7.1, as well as define the rank for each of them. We will then prove they are joinable. For *join*, the pivot can be either just the data entry (such that the algorithm will create a new tree node for it), or a pre-allocated tree node in memory carrying the corresponding data entry (such that the node may be reused, allowing for in-place updates).

As mentioned in the introduction and the beginning of this chapter, *join* fully captures what is required to rebalance a tree and can be used as the only function that knows about and maintains the balance invariants. For AVL, RB and WB trees we show that *join* takes work that is proportional to the difference in rank of the two trees. For treaps the work depends on the priority of k. All the *join* algorithms are sequential so the span is equal to the work. We show in this thesis that the *join* algorithms for all balancing schemes we consider lead to optimal work for many functions on maps and sets.

7.2.1 AVL Trees

```
1  joinRightAVL($T_l, k, T_r$) {
2    $(l, k', c)$ = expose($T_l$);
3    if $h(c) \leq h(T_r) + 1$ then {
4      $T' = node(c, k, T_r)$;
5      if $h(T') \leq h(l) + 1$ then return $node(l, k', T')$;
6      else return rotateLeft($node(l, k'$, rotateRight($T'$)));
7    } else {
8      $T'$ = joinRightAVL($c, k, T_r$);
9      $T''$ = $node(l, k', T')$;
10     if $h(T') \leq h(l) + 1$ then return $T''$; else return rotateLeft($T''$); }}
11 join($T_l, k, T_r$) {
12   if $h(T_l) > h(T_r) + 1$ then return joinRightAVL($T_l, k, T_r$);
13   else if $h(T_r) > h(T_l) + 1$ then return joinLeftAVL($T_l, k, T_r$);
14   else return $node(T_l, k, T_r)$; }
```

Figure 21: **The *join* algorithm on AVL trees** – `joinLeftAVL` is symmetric to `joinRightAVL`.

For AVL trees, we define the rank as the height, i.e., $r(T) = h(T)$. Pseudocode for AVL *join* is given in Figure 21 and illustrated in Figure 22. Every node stores its own height so that $h(\cdot)$ takes constant time. If the two trees T_l and T_r differ by height at most one, *join* can simply create a new $node(T_l, e, T_r)$. However if they differ by more than one then rebalancing is required. Suppose that $h(T_l) > h(T_r) + 1$ (the other case is symmetric). The idea is to follow the right spine of T_l until a node c for which $h(c) \leq h(T_r) + 1$ is found (line 3). At this point a new $node(c, e, T_r)$ is created to replace c (line 4). Since either $h(c) = h(T_r)$ or $h(c) = h(T_r) + 1$, the new node satisfies the AVL invariant, and its height is one greater than c. The increase in height can increase the height of its ancestors, possibly invalidating the AVL invariant of those nodes. This can be fixed either with a double rotation if invalid at the parent (line 6) or a single left rotation if invalid higher in the tree (line 10), in both cases restoring the height for any further ancestor nodes. The algorithm will therefore require at most two rotations, as we summarized in the following lemma.

Lemma 7.1. *The join algorithm in Figure 21 on AVL trees requires at most two rotations.*

Lemma 7.2. *For two AVL trees T_l and T_r, the AVL join algorithm works correctly, runs with $O(|h(T_l) - h(T_r)|)$ work, and returns a tree satisfying the AVL invariant with height at most $1 + \max(h(T_l), h(T_r))$.*

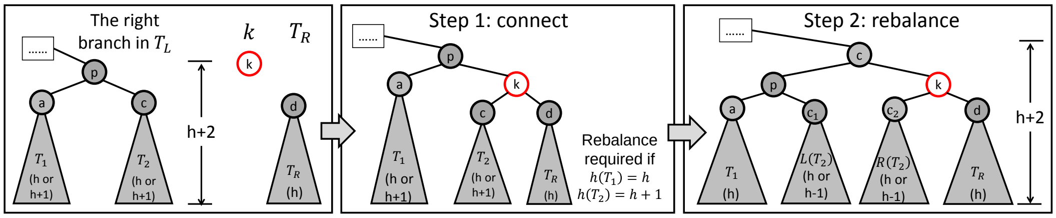


Figure 22: **An example for *join* on AVL trees** – An example for *join* on AVL trees ($h(T_l) > h(T_r) + 1$). We first follow the right spine of T_l until a subtree of height at most $h(T_r) + 1$ is found (i.e., T_2 rooted at c). Then a new $node(c, k, T_r)$ is created, replacing c (Step 1). If $h(T_1) = h$ and $h(T_2) = h + 1$, the node p will no longer satisfy the AVL invariant. A double rotation (Step 2) restores both balance and its original height.

Proof. Since the algorithm only visits nodes on the path from the root to c, and only requires at most two rotations (Lemma 7.1), it does work proportional to the path length. The path length is no more than the difference in height of the two trees since the height of each consecutive node along the right spine of T_l differs by at least one. Along with the case when $h(T_r) > h(T_l) + 1$, which is symmetric, this gives the stated work bounds. The resulting tree satisfies the AVL invariants since rotations are used to restore the invariant. The height of any node can increase by at most one, so the height of the whole tree can increase by at most one. □

7.2.2 Red-black Trees

Tarjan describes how to implement the *join* function for red-black trees [27]. Here we describe a variant that does not assume the roots are black (this is to bound the increase in rank by *union*). The pseudocode is given in Figure 23. We store at every node its black height $\hat{h}(\cdot)$. Also, we define the *increase-2* node as a black node, whose both children are also black. This means that the node increases the rank of its children by 2. In the algorithm, the first case is when $\hat{h}(T_r) = \hat{h}(T_l)$. Then if the input node is a increase-2 node, we use it as a black node and directly concatenate the two input trees. This increases the rank of the input by at most 2. Otherwise, if both $root(T_r)$ and $root(T_l)$ are black, we create red $node(T_l, e, T_r)$. When either $root(T_r)$ or $root(T_l)$ is red, we create black $node(T_l, e, T_r)$.

The second case is when $\hat{h}(T_r) < \hat{h}(T_l) = \hat{h}$ (the third case is symmetric). Similarly to AVL trees, *join* follows the right spine of T_l until it finds a black node c for which $\hat{h}(c) = \hat{h}(T_r)$. It then creates a new red $node(c, k, T_r)$ to replace c. Since both c and T_r have the same height, the only invariant that can be violated is the red rule on the root of T_r, the new node, and its parent, which can all be red. In the worst case we may have three red nodes in a row. This is fixed by a single left rotation: if a black node v has $rc(v)$ and $rc(rc(v))$ both red, we turn $rc(rc(v))$ black and perform a single left rotation on v, turning the new node black, and then performing a single left rotation

```
joinRightRB($T_l, k, T_r$) {
  if ($r(T_l) = \lfloor r(T_r)/2 \rfloor \times 2$) then return $node(T_l, \langle k, \mathtt{red} \rangle, T_r)$; else {
    ($L', \langle k', c' \rangle, R'$)=expose($T_l$);
    $T'$ = $node(L', \langle k', c' \rangle$,joinRightRB($R', k, T_r$));
    if ($c'$=black) and ($color(rc(T')) = color(rc(rc(T')))$=red) then {
      set $rc(rc(T'))$ as black;
      return rotateLeft($T'$);
    } else return $T'$; }}

joinRB($T_l, k, T_r$) {
  if $T_l$ has a larger black height then {
    $T'$ =joinRightRB($T_l, k, T_r$);
    if ($color(T')$=red) and ($color(rc(T'))$=red) then return $node(lc(T'), \langle k(T'), \mathtt{black} \rangle, rc(T'))$;
    else return $T'$;
  } else if $T_r$ has a larger black height then {
    $T'$ =joinLeftRB($T_l, k, T_r$);
    if ($color(T')$=red) and ($color(lc(T'))$=red) then return $node(lc(T'), \langle k(T'), \mathtt{black} \rangle, rc(T'))$;
    else return $T'$;
  } else {
    if (k is a increase-2 node) then
      return $node(T_l, \langle k, \mathtt{black} \rangle, T_r)$;
    else if ($color(T_l)$=black) and ($color(T_r)$=black)
      return $node(T_l, \langle k, \mathtt{red} \rangle, T_r)$;
    else return $node(T_l, \langle k, \mathtt{black} \rangle, T_r)$; }
}
```

Figure 23: **The *join* algorithm on red-black trees** – The *join* algorithm on red-black trees. joinLeftRB is symmetric to joinRightRB.

on v. The update is illustrated in Figure 24. The rotation, however can again violate the red rule between the root of the rotated tree and its parent, requiring another rotation. Obviously the triple-red issue is resolved after the first rotation. Therefore, expect the bottommost level, a triple-red issue does not happen. The double-red issue might proceed up to the root of T_l. If the original root of T_l is red, the algorithm may end up with a red root with a red child, in which case the root will be turned black, turning T_l rank from $2\hat{h}-1$ to $2\hat{h}$. If the original root of T_l is black, the algorithm may end up with a red root with two black children, turning the rank of T_l from $2\hat{h}-2$ to $2\hat{h}-1$. In both cases the rank of the result tree is at most $1 + r(T_l)$.

We note that the rank of the output can increase the larger rank of the input trees by 2 only when the pivot is an increase-2 node and the two input trees are balanced both with black roots. In general we do not need to deal with the increase-2 nodes specifically for a correct *join* algorithm. We define the increasing-2 nodes for the purpose of bounding the cost of some *join*-based algorithms.

Lemma 7.3. *For two RB trees T_l and T_r, the RB join algorithm works correctly, runs with $O(|\hat{h}(T_l) - \hat{h}(T_r)|)$ work, and returns a tree satisfying the red-black invariants and* *with black height at most* $1 + \max(\hat{h}(T_l), \hat{h}(T_r))$.

Proof. The base case where $h(T_l) = h(T_r)$ is straight-forward. For symmetry, here we only prove the case when $h(T_l) > h(T_r)$. We prove the proposition by induction.

We first show the correctness. As shown in Figure 24, after appending T_r to T_l, if p is black, the rebalance has been done, the height of each node stays unchanged. Thus the RB tree is still valid. Otherwise, p is red, p's parent g must be black. By applying a left rotation on p and g, we get a balanced RB tree rooted at p, except the root p is red. If p is the root of the whole tree, we change p's color to black, and the height of the whole tree increases by 1. The RB tree is still valid. Otherwise, if the current parent of p (originally g's parent) is black, the rebalance is done here. Otherwise a similar rebalance is required on p and its current parent. Thus finally we will either find the current node valid (current red node has a black parent), or reach the root, and change the color of root to be black. Thus when we stop, we will always get a valid RB tree.

Since the algorithm only visits nodes on the path from the root to c, and only requires at most a single rotation per node on the path, the overall work for the algorithm is proportional to the depth of c in T_r. This in turn is no more than twice the difference in black height of the two trees since the black height decrements at least every two nodes along the path. This gives the stated work bounds.

For the rank, note that throughout the algorithm, before reaching the root, the black rule is never invalidated (or is fixed immediately), and the only invalidation occurs on the red rule. If the two input trees are originally balanced, the rank increases by at most 2. The only case that the rank increases by 2 is when k is from an increase-2 node, and both *root*(T_r) and *root*(T_l) are black.

If the two input tree are not balanced, the black height of the root does not change before the algorithm reaching the root (Step 3 in Figure 24). There are then three cases:

1. The rotation does not propagate to the root, and thus the rank of the tree remains as $\max(\hat{h}(T_l), \hat{h}(T_r))$.

2. (Step 3 Case 1) The original root color is red, and thus a double-red issue occurs at the root and its right child. In this case the root is colored black. The black height of the tree increases by 1, but since the original root is red, the rank increases by only 1.

3. (Step 3 Case 1) The original root color is black, but the double-red issue occurs at the root's child and grandchild. In this case another rotation is applied as shown in Figure 24. The black height remains, but the root changed from black to red, increasing the rank by 1.

□

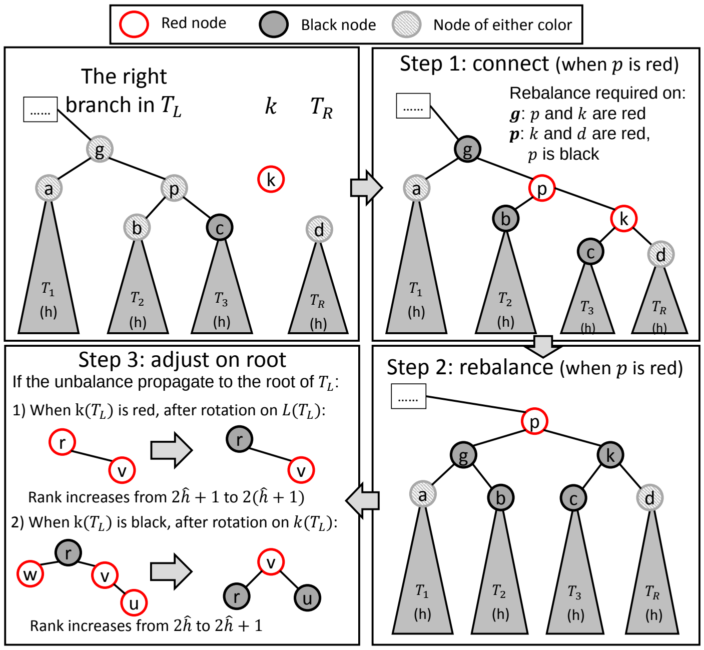

Red node
Black node
Node of either color
The right branch in T_L
k
T_R
......
g
a
p
b
c
k
d
T_1 (h)
T_2 (h)
T_3 (h)
T_R (h)
Step 1: connect (when p is red)
Rebalance required on:
g: p and k are red
p: k and d are red, p is black
Step 2: rebalance (when p is red)
Step 3: adjust on root
If the unbalance propagate to the root of T_L:
1) When k(T_L) is red, after rotation on $L(T_L)$:
r
v
Rank increases from $2\hat{h}+1$ to $2(\hat{h}+1)$
2) When k(T_L) is black, after rotation on $k(T_L)$:
w
r
v
u
Rank increases from $2\hat{h}$ to $2\hat{h}+1$

Figure 24: **An example of *join* on red-black trees** – An example of *join* on red-black trees ($\hat{h} = \hat{h}(T_l) > \hat{h}(T_r)$). We follow the right spine of T_l until we find a black node with the same black height as T_r (i.e., c). Then a new red $node(c, k, T_r)$ is created, replacing c (Step 1). The only invariant that can be violated is when either c's previous parent p or T_r's root d is red. If so, a left rotation is performed at some black node. Step 2 shows the rebalance when p is red. The black height of the rotated subtree (now rooted at p) is the same as before ($h + 1$), but the parent of p might be red, requiring another rotation. If the red-rule violation propagates to the root, the root is either colored red, or rotated left (Step 3).

7.2.3 Weight Balanced Trees

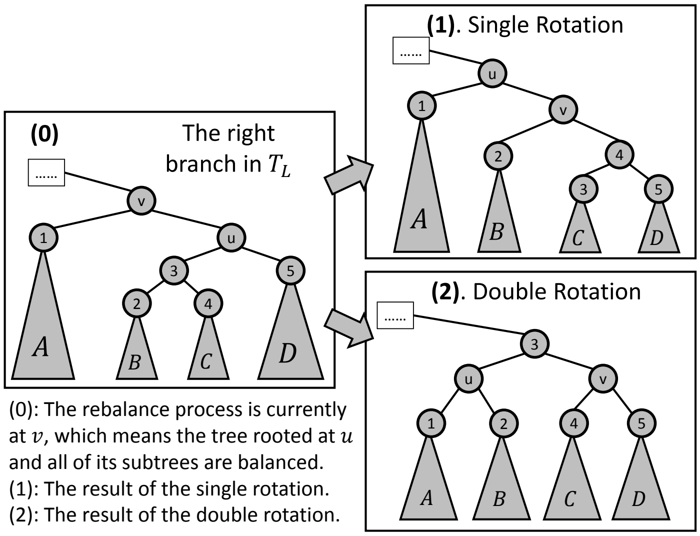
For WB trees $r(T) = \log_2(w(T)) - 1$. We store the weight of each subtree at every node. The algorithm for joining two weight-balanced trees is similar to that of AVL trees and RB trees. The pseudocode is shown in Figure 25. The *like* function in the code returns true if the two input tree sizes are balanced based on the factor of α, and false otherwise. If T_l and T_r have like weights the algorithm returns a new $node(T_l, e, T_r)$. Suppose $|T_r| \leq |T_l|$, the algorithm follows the right branch of T_l until it reaches a node c with like weight to T_r. It then creates a new $node(c, r, T_r)$ replacing c. The new node will have weight greater than c and therefore could imbalance the weight of c's ancestors. This can be fixed with a single or double rotation (as shown in Figure 26) at each node assuming α is within the bounds given in Section 7.1.

Lemma 7.4. *For two α weight-balanced trees T_l and T_r and $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$, the weight-balanced join algorithm works correctly, runs with $O(|\log(w(T_l)/w(T_r))|)$ work, and returns a tree satisfying the α weight-balance invariant.*

```
joinRightWB($T_l, k, T_r$) {
  $(l, k', c)$=expose($T_l$);
  if (balance($|T_l|, |T_r|$) then return node($T_l, k, T_r$)); else {
    $T'$ = joinRightWB($c, k, T_r$);
    $(l_1, k_1, r_1)$ = expose($T'$);
    if like($|l|, |T'|$) then return node($l, k', T'$);
    else if (like($|l|, |l_1|$)) and (like($|l| + |l_1|, r_1$)) then return rotateLeft(node($l, k', T'$));
    else return rotateLeft(node($l, k'$,rotateRight($T'$))); }}

joinWB($T_l, k, T_r$) {
  if heavy($T_l, T_r$) then return joinRightWB($T_l, k, T_r$);
  else if heavy($T_r, T_l$) then return joinLeftWB($T_l, k, T_r$);
  else return node($T_l, k, T_r$); }
```

Figure 25: **The *join* algorithm on weight-balanced trees** – `joinLeftWB` is symmetric to `joinRightWB`.



(0): The rebalance process is currently at v, which means the tree rooted at u and all of its subtrees are balanced.
(1): The result of the single rotation.
(2): The result of the double rotation.

Figure 26: **An illustration of single and double rotations possibly needed to rebalance weight-balanced trees** – In the figure the subtree rooted at u has become heavier due to joining in T_l and its parent v now violates the balance invariant.

The proof of this lemma can be found in **??**. Notice that this upper bound is the same as the restriction on α to yield a valid weighted-balanced tree when inserting a single node. Then we can induce that when the rebalance process reaches the root, the new weight-balanced tree is valid. The proof is intuitively similar as the proof stated in [19, **?**], which proved that when $\frac{2}{11} \leq \alpha \leq 1 - \frac{1}{\sqrt{2}}$, the rotation will rebalance the tree after one single insertion. In fact, in the *join* algorithm, the "inserted" subtree must be along the left or right spine, which actually makes the analysis easier.

```
joinTreap($T_l, k, T_r$) {
  if prior($k, k_1$) and prior($k, k_2$) then return node($T_l, k, T_r$) else {
    ($l_1, k_1, r_1$)=expose($T_l$);
    ($l_2, k_2, r_2$)=expose($T_r$);
    if prior($k_1, k_2$) then return node($l_1, k_1$,joinTreap($r_1, k, T_r$));
    else return node(joinTreap($T_l, k, l_2$),$k_2, r_2$); }}
```

Figure 27: **The *join* algorithm on treaps** – prior(k_1, k_2) decides if the node k_1 has a higher priority than k_2.

7.2.4 Treaps

The treap *join* algorithm (as in Figure 27) first picks the key with the highest priority among k, $k(T_l)$ and $k(T_r)$ as the root. If k is the root then the we can return $node(T_l, k, T_r)$. Otherwise, WLOG, assume $k(T_l)$ has a higher priority. In this case $k(T_l)$ will be the root of the result, $lc(T_l)$ will be the left tree, and $rc(T_l)$, k and T_r will form the right tree. Thus *join* recursively calls itself on $rc(T_l)$, k and T_r and uses result as $k(T_l)$'s right child. When $k(T_r)$ has a higher priority the case is symmetric. The cost of *join* is therefore the depth of the key k in the resulting tree (each recursive call pushes it down one level). In treaps the shape of the result tree, and hence the depth of k, depend only on the keys and priorities and not the history. Specifically, if a key has the t^{th} highest priority among the keys, then its expected depth in a treap is $O(\log t)$ (also w.h.p.). If it is the highest priority, for example, then it remains at the root.

Lemma 7.5. *For two treaps T_l and T_r, if the priority of k is the t-th highest among all keys in $T_l \cup \{k\} \cup T_r$, the treap join algorithm works correctly, runs with $O(\log t + 1)$ work in expectation and w.h.p., and returns a tree satisfying the treap invariant.*

Function	***Work***	***Span***
insert, *delete*, *update*, *find*, *first*, *last*, *range*, *split*, *join2*, *previous*, *next*, *rank*, *select*, *up_to*, *down_to*	$O(\log n)$	$O(\log n)$
union, *intersection*, *difference*	$O\left(m \log\left(\frac{n}{m}+1\right)\right)$	$O(\log n \log m)$
map, *reduce*, *map_reduce*, *to_array*	$O(n)$	$O(\log n)$
build, *filter*	$O(n)$	$O(\log^2 n)$

Table 3: **The core *join*-based algorithms and their asymptotic costs** – The cost is given under the assumption that all parameter functions take constant time to return. For functions with two input trees (*union*, *intersection* and *difference*), n is the size of the larger input, and m of the smaller.

7.3 Algorithms Using *join*

Split

```
split($T$,$k$) {
  if $T = \emptyset$ then
    return ($\emptyset$,false,$\emptyset$);
  $(L,m,R)$ = expose($T$);
  if $k = m$ then return ($L$,true,$R$);
  if $k < m$ then {
    $(T_L,b,T_R)$ = split($L,k$);
    return ($T_L$,$b$,join($T_R,m,R$)); }
  $(T_L,b,T_R)$ = split($R,k$);
  return (join($L,m,T_L$),$b$,$T_R$); } }
```

join2

```
split_last($T$) { // split_first is symmetric
  $(L,k,R)$ = expose($T$);
  if $R = \emptyset$ then return($L,k$);
  $(T',k')$ = split_last($R$);
  return (join($L,k,T'$),$k'$); }
join2($T_l$,$T_r$) {
  if $T_l = \emptyset$ then return $T_r$;
  $(T',k)$ = split_last($T_l$);
  return join($T',k,T_r$);
}
```

Figure 28: ***split* and *join2* algorithms** – They are both independent of balancing schemes.

The *join* function, as a subroutine, has been used and studied by many researchers and programmers to implement more general set operations. In this section, we describe algorithms for various functions that use just *join*. The algorithms are generic across balancing schemes. The pseudocodes for the algorithms in this section is shown in Figure 30. Beyond *join* the only access to the trees we make use of is through *expose*, which only read the root. main set operations, which are *union*, *intersection* and *difference*, are optimal (or known as *efficient*) in work. The pseudocode for all the algorithms introduced in this section is presented in Figure 31.

7.3.1 Two Helper Functions: *split* and *join2*

We start with presenting two helper functions *split* and *join2*. For a BST T and key k, *split*(T, k) returns a triple (T_l, b, T_r), where T_l (T_r) is a tree containing all keys in T that are less (larger) than k, and b is a flag indicating whether $k \in T$. *join2*(T_l, T_r) returns a binary tree for which the in-order values is the concatenation of the in-order values of the binary trees T_l and T_r (the same as *join* but without the middle key). For BSTs, all keys in T_l have to be less than keys in T_r.

Although both sequential, these two functions, along with the *join* function, are essential for help other algorithms to achieve good parallelism. Intuitively, when processing a tree in parallel, we recurse on two sub-components of the tree in parallel by *split*ing the tree by some key. In many cases, the splitting key is just the root, which means directly using the two subtrees of natural binary tree structure. After the recursions return, we combine the result of the left and right part, with or without the middle key, using *join* or *join2*. Because of the balance of the tree, this framework usually gives high parallelism with shallow span (e.g., poly-logarithmic).

Split.. As mentioned above, *split*(T, k) splits a tree T by a key k into T_l and T_r, along with a bit b indicating if $k \in T$. Intuitively, the *split* algorithm first searches for k in T, splitting the tree along the path into three parts: keys to the left of the path, k itself (if it exists), and keys to the right. Then by applying *join*, the algorithm merges all the subtrees on the left side (using keys on the path as intermediate nodes) from bottom to top to form T_l, and merges the right parts to form T_r. Writing the code in a recursive manner, this algorithm first determine if k falls in the left (right) subtree, or is exactly the root. If it is the root, then the algorithm straightforwardly returns the left and the right subtrees as the two return trees and *true* as the bit b. Otherwise, WLOG, suppose k falls in the left subtree. The algorithm further *split* the left subtree into T_L and T_R with the return bit b'. Then the return bit $b = b'$, the T_l in the final result will be T_L, and T_r means to *join* T_R with the original right subtree by the original root. Figure 29 gives an example.

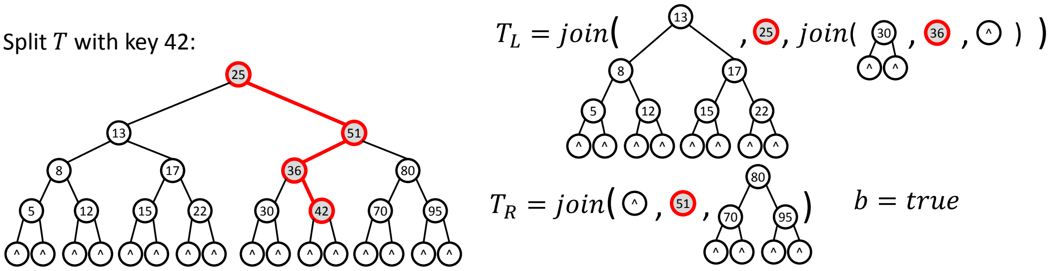


Figure 29: **An example of *split* in a BST with key** 42 – We first search for 42 in the tree and split the tree by the searching path, then use *join* to combine trees on the left and on the right respectively, bottom-top.

The cost of the algorithm is proportional to the rank of the tree, as we summarize and prove in the following theorem.

Theorem 7.1. *The work of split(T, k) is $O(h(T))$ for AVL, RB, WB trees and treaps.*

Proof Sketch. We only consider the work of joining all subtrees on the left side. The other side is symmetric. Suppose we have l subtrees on the left side, denoted from bottom to top as $T_1, T_2, \ldots T_l$. We consecutively join T_1 and T_2 returning T_2', then join T_2' with T_3 returning T_3' and so forth, until all trees are merged. The overall work of *split* is the sum of the cost of $l - 1$ *join* functions.

We now use an AVL tree as an example to show the proof. Recall that each *join* costs time $O(|h(T_{i+1}) - h(T_i')|)$, and increase the height of T_{i+1} by at most 1. Also $h(T_i')$ is achieved by joining T_i and T_{i-1}'. Considering T_i is a subtree in T_{i+1}'s sibling, and thus $h(T_i')$ is no more than $h(T_{i+1}) + 2$. The overall complexity is $\sum_{i=1}^{l} |h(T_{i+1}) - h(T_i')| \leq \sum_{i=1}^{l} h(T_{i+1}) - h(T_i') + 2 = O(h(T))$.

For RB and WB trees, the proof is similar to the above proof for AVL trees, but only changes *join* cost based on the difference in black-height or log of weight, instead of height.

For treaps, each *join* uses the key with the highest priority since the key is always on a upper level. Hence by Lemma 7.5, the complexity of each *join* is $O(1)$ and the work of split is at most $O(h(T))$. □

Union

```
union($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $T_2$;
  if $T_2 = \emptyset$ then return $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = union($L_1$,$L_2$) ||
        $T_r$ = union($R_1$,$R_2$);
  return join($T_l$,$k_2$,$T_r$);
}
```

Intersection

```
intersect($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $\emptyset$;
  if $T_2 = \emptyset$ then return $\emptyset$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = intersect($L_1$,$L_2$) ||
    $T_r$ = intersect($R_1$,$R_2$);
  if $b$ then return join($T_l$,$k_2$,$T_r$);
  else return join2($T_l$,$T_r$); }
```

Difference

```
difference($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then $\emptyset$;
  if $T_2 = \emptyset$ then $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = difference($L_1$,$L_2$) ||
    $T_r$ = difference($R_1$,$R_2$);
  return join2($T_l$,$T_r$);
}
```

Figure 30: ***join*-based algorithms for set-set operations** – They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

Join2.. As stated above, the *join2* function is defined similar to *join* without the middle entry. The *join2* algorithm first choose one of the the input trees, and extract its last (if it is T_l) or first (if it is T_r) element k. The two cases take the same asymptotical cost. The extracting process is similar to the *split* algorithm. The algorithm then uses k as the pivot to *join* the two trees. In the code shown in Figure 28, the *split_last* algorithm first finds the last element k (by following the right spine) in T_l and on the way back to root, *join*s the subtrees along the path. We denote the result of dropping k in T_L as T'. Then *join*(T', k, T_r) is the result of *join2*. Unlike *join*, the work of *join2* is proportional to the rank of both trees since both *split* and *join* take at most logarithmic work.

Theorem 7.2. *The work of* T =*join2*(T_l, T_r) *is* $O(r(T_l) + r(T_r))$ *for all joinable trees.*

The cost bound holds because *split_last* and *join* both take work asymptotically no more than the larger tree rank.

7.4 Set-set Functions Using *join*

In this section, we will present the *join*-based algorithm on set-set functions, including *union*, *intersection* and *difference*. Many other set-set operations, such as symmetric difference, can be implemented by a combination of *union*, *intersection* and *difference* with no extra asymptotical work. We will start with presenting some background of these algorithms, and then explain in details about the *join*-based algorithms. Finally, we show the proof of their cost bound.

Background. The parallel set-set functions are particularly useful when using parallel machines since they can support parallel bulk updates. As mentioned, although supporting efficient algorithms for basic operations on trees, such as insertion and deletion, are rather straightforward, implementing efficient bulk operations is more challenging, especially considering parallelism and different balancing schemes. For example, combining two ordered sets of size n and $m \leq n$ in the format of two arrays would take work $O(m+n)$ using the standard merging algorithm in the merge sort algorithm. This makes even inserting an single element into a set of size n to have linear cost. This is because even most of the chunks of data in the input remain consecutive, the algorithm still need to scan and copy them to the output array. Another simple implementation is to store both sets as balanced trees, and insert the elements in the smaller tree into the larger one, costing $O(m \log n)$ work. It overcomes the issue of redundant scanning and copying, because many subtrees in the larger tree remain untouched. However, this results in $O(n \log n)$ time, for combining two ordered sets of the same size, while it is easy to make it $O(n)$ by arrays. The problem lies in that the algorithm fails to make use of the ordering in the smaller tree.

The lower bound for comparison-based algorithms for *union*, *intersection* and *difference* for inputs of size n and $m \leq n$, and returning an ordered structure[2], is $\log_2 \binom{m+n}{n} = \Theta\left(m \log\left(\frac{n}{m}+1\right)\right)$ ($\binom{m+n}{n}$ is the number of possible ways n keys can be interleaved with $m$keys). The bound is interesting since it shows that implementing insertion with union, or deletion with difference, is asymptotically efficient ($O(\log n)$ time), as is taking the union of two equal sized sets ($O(n)$ time).

Brown and Tarjan first matched these bounds, asymptotically, using a sequential algorithm based on red-black trees [11]. Adams later described very elegant algorithms for union, intersection, and difference, as well as other functions based on *join* [1, 2]. Adams' algorithms were proposed in an international competition for the Standard ML community, which is about implementations on "set of integers". Prizes were awarded in two categories: fastest algorithm, and most elegant yet still efficient program. Adams won the elegance award, while his algorithm is almost as fast as the fastest program for very large sets, and was faster for smaller sets. Because of the elegance of the algorithm, at least three libraries use Adams' algorithms for their implementation of ordered sets and tables (Haskell [16] and MIT/GNU Scheme, and SML). Indeed the *join*-based algorithm that will be introduced later in this section is based on Adams' algorithms. Blelloch and Reid-Miller later show that similar algorithms for treaps [10], are optimal for work (i.e. $\Theta\left(m \log\left(\frac{n}{m}+1\right)\right)$), and are also parallel. Later, Blelloch et al. [8] extend Adams' algorithms to multiple balancing schemes and prove the cost bound.

Algorithms. *union*(T_1, T_2) takes two BSTs and returns a BST that contains the union of all keys. The algorithm uses a classic divide-and-conquer strategy, which is parallel. At each level of recursion, T_1 is split by $k(T_2)$, breaking T_1 into three parts: one with all keys smaller than $k(T_2)$ (denoted as L_1), one in the middle either of only one key equal to $k(T_2)$ (when $k(T_2) \in T_1$) or empty (when $k(T_2) \notin T_1$), the third one with all keys larger than $k(T_2)$ (denoted as R_1). ger) than $k(T_1)$. Then two recursive calls to *union* are made in parallel. One unions $lc(T_2)$ with L_1, returning T_l, and the other one

[2] By "ordered structure" we mean any data structure that can output elements in sorted order without any further comparisons—e.g., a sorted array, or a binary search tree.

unions $rc(T_2)$ with R_1, returning T_r. Finally the algorithm returns $join(T_l, k(T_2), T_r)$, which is valid since $k(T_2)$ is greater than all keys in T_l are less than all keys in T_r.

The functions *intersection* (T_1, T_2) and *difference* (T_1, T_2) take the intersection and difference of the keys in their sets, respectively. The algorithms are similar to *union* in that they use one tree to split the other. However, the method for joining and the base cases are different. For *intersection*, *join2* is used instead of *join* if the root of the first *is not* found in the second. Accordingly, the base case for the *intersection* algorithm is to return an empty set when either set is empty. For *difference*, *join2* is used anyway because $k(T_2)$ should be excluded in the result tree. The base cases are also different in the obvious way.

The cost of the algorithms described above can be summarized in the following theorem.

Theorem 7.3. *For AVL, RB, WB trees and treaps, the work and span of the algorithm (as shown in Figure 30) of union, intersection or difference on two balanced BSTs of sizes m and n ($n \geq m$) is $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$ (in expectation for treaps) and $O(\log n \log m)$ respectively (w.h.p. for treaps).*

The work bound for these algorithms is optimal in the comparison-based model. In particular considering all possible interleavings, the minimum number of comparisons required to distinguish them is $\log \binom{m+n}{n} = \Theta\left(m \log\left(\frac{n}{m} + 1\right)\right)$ [15]. A generic proof of Theorem 7.3 working for all the four balancing schemes can be found in [8]. The span of these algorithms can be reduced to $O(\log m)$ for weight-balanced trees even on the binary-forking model [9] by doing a more complicated divide-and-conquer strategy.

7.5 Other Tree algorithms Using *join*

Insert and Delete. Instead of the classic implementations of *insert* and *delete*, which are specific to the balancing scheme, we define versions based purely on *join*, and hence independent of the balancing scheme.

We present the pseudocode in Figure 31 to insert an entry e into a tree T. The base case is when t is empty, and the algorithm creates a new node for e. Otherwise, this algorithm compares k with the key at the root and recursively inserts e into the left or right subtree. After that, the two subtrees are *join*ed again using the root node. Because of the correctness of the *join* algorithm, even if there is imbalance, *join* will resolve the issue.

The *delete* algorithm is similar to *insert*, except when the key to be deleted is found at the root, where *delete* uses *join2* to connect the two subtrees instead. Both the *insert* and the *delete* algorithms run in $O(\log n)$ work (and span since sequential).

One might expect that abstracting insertion or deletion using *join* instead of specializing for a particular balance criteria has significant overhead. In fact experiments show this is not the case—and even though some extra metadata (e.g., the reference

Build

```
build_sorted($S,i,j$) {
  if $i = j$ then return $\emptyset$;
  if $i+1 = j$ then
    return singleton($S[i]$);
  $m = (i+j)/2$;
  $L$ = build'($S,i,m$) ||
    $R$ = build'($S,m+1,j$);
  return join($L,S[m],R$); }
build($S,m$) {
  (S2, m2) = sort_rm_dup(S, m);
  build_sorted(S2,0,m2);}
```

Filter

```
filter($T$,$f$) {
  if $T = \emptyset$ then return $\emptyset$;
  ($L$,$e$,$R$) = expose($T$);
  $L'$ = filter($L$,$f$) ||
      $R'$ = filter($R$,$f$);
  if $f(e)$ then
    return join($L'$,$e$,$R'$);
  else join2($L'$,$R'$); }
```

Map and Reduce

```
map_reduce($T,g',f',I'$) {
  if $T = \emptyset$ then return $I'$;
  $\langle L,k,v,R \rangle$ = expose($T$);
  $L'$ = MapReduce($L,g',f',I'$) ||
    $R'$ = MapReduce($R,g',f',I'$);
  return $f'(L', f'(g'(k,v),R'))$; }
```

Range

```
range($T,l,r$) {
  ($T_1$,$T_2$) = split($T,l$);
  ($T_3$,$T_4$) = split($T_2,r$);
  return $T_3$; }
```

Foreach Index

```
foreach_index(T, $\phi$, s) {
  if (t = $\emptyset$) return;
  ($L,e,R$) = expose($T$);
  left = size(L);
  L = foreach_index(L, $\phi$, s); ||
    R = foreach_index(R, $\phi$, s+1+left);
  $\phi$(e, left);}
```

Insertion

```
insert($T,e$) {
  if $T = \emptyset$ then return singleton($e$);
  $\langle L,e',R \rangle$ = expose($T$);
  if $k(e) = k(e')$ then return T;
  if $k(e) < k(e')$ then
    return join(insert($L,e$),$e'$,$R$);
  return join($L,e'$, insert($R,e$)); } }
```

Deletion

```
delete($T,k$) {
  if $T = \emptyset$ then return $\emptyset$;
  $\langle L,e',R \rangle$ = expose($T$);
  if $k < k(e')$ then return join(delete($L,k$),$e',R$);
  if $k(e') < k$ then return join($L,e'$, delete($R,k$));
  return join2($L,R$); }
```

Multi-insertion

```
multi_insert_s(T, A, m) {
  if (T = $\emptyset$) return build(A, m);
  if (m = 0) return t;
  $\langle L,e,R \rangle$ = expose($T$);
  b = binary_search(A, m, k(e));
  d = (b < m) and (k(A[b]) > k(e));
  L = multi_insert_s(r→lc, A, b) ||
    R = multi_insert_s(r→rc, A+b-d, m-b-d);
  return concat(L, e, R); }
multi_insert(t, A, m) {
  (A2, m2) = sort_rm_dup(A, m);
  return multi_insert_sorted(t, A2, m2);}
```

Figure 31: **Pseudocode of some *join*-based functions** – They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

counter), the *join*-based insertion algorithm is only 17% slower sequentially than the highly-optimized C++ STL library [26].

Theorem 7.4. *The join-based insertion algorithm cost time at most $O(\log |T|)$ for an AVL, RB, WB tree or a treap.*

Proof Sketch. The insertion algorithm first follow a path in the tree to find the right location for k, and then performs $O(\log n)$ *join* algorithms. Each *join* connects T_1 and $T_2 \cup \{k\}$, where T_1 and T_2 were originally balanced with each other. For any of the discussed balancing schemes, the cost of the *join* is a constant. A more rigorous proof can be shown by induction. □

Theorem 7.5. *The join-based deletion algorithm cost time at most $O(\log |T|)$ for an AVL, RB, WB tree or a treap.*

Proof Sketch. The proof is similar to the proof of Theorem 7.4. The only exception is that at most one *join2* algorithm can be performed. This only adds an extra $O(\log n)$ cost. □

Build. A balanced binary tree can be created from a sorted array of key-value pairs using a balanced divide-and-conquer over the input array and combining with *join*. To construct a balanced binary tree from an arbitrary array we first sort the array by the keys, then remove the duplicates. All entries with the same key are consecutive after sorting, so the algorithm first applies a parallel sorting and then follows by a parallel packing. The algorithm then extracts the median in the de-duplicated array, and recursively construct the left/right subtree from the left/right part of the array, respectively. Finally, the algorithm uses *join* to connect the median and the two subtrees. The work is then $O(W_{\text{sort}(n)} + W_{\text{remove}(n)} + n)$ and the span is $O(S_{\text{sort}(n)} + S_{\text{remove}(n)} + \log n)$. For work-efficient sort and remove-duplicates algorithms with $O(\log n)$ span this gives the bounds in Table 3.

Bulk Updates. We use *multi_insert* and *multi_delete* to commit a batch of write operations. The function *multi_insert*(T, A, m) takes as input a tree root t, and the head pointer of an array A with its length m.

We present the pseudocode of *multi_insert* in Figure 31. This algorithm first sorts A by keys, and then removes duplicates in a similar way as in *build*. We then use a divide-and-conquer algorithm *multi_insert_s* to insert the sorted array into the tree. The base case is when either the array A or T is empty. Otherwise, the algorithm uses a binary search to locate t's key in the array, getting the corresponding index b in A. d is a bit denoting if k appears in A. Then the algorithm recursively multi-inserts A's left part (up to $A[b]$) into the left subtree, and A's right part into the right subtree. The two recursive calls can run in parallel. The algorithm finally concatenates the two results by the root of T. A similar divide-and-conquer algorithm can be used for *multi_delete*, using *join2* instead of *join* when necessary.

Decoupling sorting from inserting has several benefits. First, parallel sorting is well-studied and there exist highly-optimized sorting algorithms that can be used. This simplifies the problem. Second, after sorting, all entries in A that to be merged with a certain subtree in T become consecutive. This enables the divide-and-conquer approach which provides good parallelism, and also gives better locality.

The total work and span of inserting or deletion an array of length m into a tree of size $n \geq m$ is $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ and $O(\log m \log n)$, respectively [8]. The analysis is similar to the *union* algorithm.

Range. *range* extracts a subset of tuples in a certain key range from a tree, and output them in a new tree. The cost of the *range* function is $O(\log n)$. The pure *range* algorithm copies nodes on two paths, one to each end of the range, and using them as pivots to *join* the subtrees back. When the extracted range is large, this pure *range* algorithm is much more efficient (logarithmic time) than visiting the whole range and copying it.

Filter. The *filter*(t, ϕ) function returns a tree with all tuples in T satisfying a predicate ϕ. This algorithm filters the two subtrees recursively, in parallel, and then determines if the root satisfies ϕ. If so, the algorithm uses the root as the pivot to *join* the two recursive results. Otherwise it calls *join2*. The work of *filter* is $O(n)$ and the depth is $O(\log^2 n)$ where n is the tree size.

Map and Reduce. The function *map_reduce*$(T, f_m, \langle f_r, I\rangle)$ on a tree t (with data type E for the tuples) takes three arguments and returns a value of type V'. $f_m : E \mapsto V'$ is the a map function that converts each stored tuple to a value of type V'. $\langle f_r, I\rangle$ is a monoid where $f_r : V' \times V' \mapsto V'$ is an associative reduce function on V', and $I \in V'$ is the identity of f_r. The algorithm will recursively call the function on its two subtrees in parallel, and reduce the results by f_r afterwards.

8 Other Models and Simulations

In this section we consider some other models (currently just the PRAM) and discuss simulation results between models. We are particularly interested in how to simulate the MP-RAM on a machine with a fixed number of processors. In particular we consider the *scheduling* problem, which is the problem of efficiently scheduling processes onto processors.

8.1 PRAM

The Parallel Random Access RAM (PRAM) model was one of the first models considered for analyzing the cost of parallel algorithms. Many algorithms were analyzed in the model in the 80s and early 90s. A PRAM consists of p processors sharing a memory of unbounded size. Each has its own register set, and own program counter, but they all run synchronously (one instruction per cycle). In typical algorithms all processors are executing the same instruction sequence, except for some that might be inactive. Each processor can fetch its identifier, an integer in $[1, \ldots, p]$. The PRAM differes from the MP-PRAM in two important ways. Firstly during a computation it always has a fixed number of processors instead of allowing the dynamic creation of processes. Secondly the PRAM is completely synchronous, all processors working in lock-step.

Costs are measured in terms of the number of instructions, the time, and the number of processors. The time for an algorithm is often a function of the number of processors. For example to take a sum of n values in a tree can be done in $O(n/p + \log p)$ time. The idea is to split the input into blocks of size n/p, have processor i sum the elements in the i^{th} block, and then sum the results in a tree.

Since all processors are running synchronously, the types of race conditions are somewhat different than in the MP-RAM. If there is a reads and a writes on the same cycle at the same location, the reads happen before the writes. There are variants of the PRAM depending on what happens in the case of multiple writes to the same location on the same cycle. The exclusive-write (EW) version disallows concurrent writes to the same location. The Arbitrary Concurrent Write (ACW) version assumes an arbitrary write wins. The Priority Concurrent Write (PCW) version assumes the processor with highest processor number wins. There are asynchronous variants of the PRAM, although we will not discuss them.

8.2 Simulations

To be written.

8.3 The Scheduling Problem

We are interested in scheduling the dynamic creation of tasks implied by the MP-RAM onto a fixed number of processors, and in mapping work and depth bounds onto time bounds for those processors. This scheduling problem can be abstracted as traversing a DAG. In particular the p processor *scheduling problem* is given a DAG with a single root, to visit all vertices in steps such that each step visits at most p vertices, and no vertex is visited on a step unless all predecessors in the DAG have been visited on a previous step. This models the kind of computation we are concerned with since each instruction can be considered a vertex in the DAG, no instruction can be executed until its predecessors have been run, and we assume each instruction takes constant time.

Our goal is to bound the number of steps as a function of the the number of vertices w in a DAG and its depth d. Furthermore we would like to ensure each step is fast. Here we will be assuming the synchronous PRAM model, as the target, but most of the ideas carry over to more asynchronous models.

It turns out that in general finding the schedule with the minimum number of steps is NP-hard [28] but coming up with reasonable approximations is not too hard. Our first observation is a simple lower bound. Since there are w vertices and each step can only visit p of them, any schedule will require at least w/p steps. Furthermore since we have to finish the predecessors of a vertex before the vertex itself, the schedule will also require at least d steps. Together this gives us:

Observation 8.1. *Any p processor schedule of a DAG of depth d and size w requires at least* $\max(w/p, d)$ *steps.*

We now look at how close we can get to this.

8.4 Greedy Scheduling

A greedy scheduler is one in which a processor never sits idle when there is work to do. More precisely a *p-greedy* schedule is one such that if there are r ready vertices on a step, the step must visit $\min(r, p)$ of them.

Theorem 8.1. *Any p-greedy schedule on a DAG of size w and depth d will take at most $w/p + d$ steps.*

Proof. Let's say a step is *busy* if it visits p vertices and *incomplete* otherwise. There are at most $\lfloor w/p \rfloor$ busy steps, since that many will visit all but $r < p$ vertices. We now bound the number of incomplete steps. Consider an incomplete step, and let j be the first level in which there are unvisited vertices before taking the step. All vertices on level j are ready since the previous level is all visited. Also $j < p$ since this step is incomplete. Therefore the step will visit all remaining vertices on level j (and possibly others). Since there are only d levels, there can be at most d incomplete steps. Summing the upper bounds on busy and incomplete steps proves the theorem. □

We should note that such a greedy schedule has a number of steps that is always within a factor of two of the lower bound. It is therefore a two-approximation of the optimal. If either term dominates the other, then the approximation is even better. Although greedy scheduling guarantees good bounds it does not it does not tell us how to get the ready vertices to the processors. In particular it is not clear we can assign ready tasks to processors constant time.

8.5 Work Stealing Schedulers

We now consider a scheduling algorithm, work stealing, that incorporates all costs. The algorithm is not strictly greedy, but it does guarantee bounds close to the greedy bounds and allows us to run each step in constant time. The scheduler we discuss is limited to binary forking and joining. We assume that visiting a vertex returns one of three possibilities: `fork`(v_1, v_2) the vertex is a fork, `next`(v) if it has a single ready child, or `empty` if it has no ready child. Note that if the child of a vertex is a join point a visit could return either `next`(v) if the other parent of v has already finished or `empty` if not. Since the two parents of a join point could finish simultaneously, we can use a test-and-set (or a concurrent write followed by a read) to order them.

The work stealing algorithm (or scheduler) maintains the ready vertices in a set of work queues, one per processor. Each processor will only push and pop on the bottom of its own queue and pop from the top when stealing from any queue. The scheduler

```
1  workStealingScheduler($v$) =
2    pushBot($Q[0], v$);
3    while not all queues are empty
4      parfor $i$ in $[0 : p]$
5        if empty($Q[i]$) then                % steal phase
6          $j$ = rand($[0 : p]$);
7          steal[$j$] = $i$;
8          if (steal[$j$] $= i$) and not(empty($Q[j]$) then
9            pushBot($Q[i]$,popTop($Q[j]$))
10       if (not(empty($Q[i]$)) then          % visit phase
11         $u$ = popBot($Q[i]$);
12         case (visit($u$)) of
13           fork($v_1, v_2$) $\Rightarrow$ pushBot($Q[i], v_2$); pushBot($Q[i], v_1$);
14           next($v$) $\Rightarrow$ pushBot($Q[i], v$);
```

Figure 32: Work stealing scheduler. The processors need to synchronize between line 7 and the next line, and between the two phases.

starts with the root of the DAG in one of the queues and the rest empty. Pseudocode for the algorithm is given in Figure 32. Each step of the scheduler consists of a steal phase followed by a visit phase. During the steal phase each processor that has an empty queue picks a random target processor, and attempts to "steal" the top vertex from its queue. The attempt can fail if either the target queue is empty or if someone else tries a steal from the target on the same round and wins. The failure can happen even if the queue has multiple vertices since they are all trying to steal the top. If the steal succeeds, the processor adds the stolen vertex to its own queue. In the visit phase each processor with a non-empty queue removes the vertex from the bottom of its queue, visits it, and then pushes back 0, 1 or 2 new vertices onto the bottom.

The work stealing algorithm is not completely greedy since some ready vertices might not be visited even though some processors might fail on a steal. In our analysis of work stealing we will use the following definitions. We say that the vertex at the top of every non-empty queue is *prime*. In the work stealing scheduler each join node is enabled by one of its parents (i.e., put in its queue). If throughout the DAG we just include the edge to the one parent, and not the other, what remains is a tree. In the tree there is a unique path from the root of the DAG to the sink, which we call the *critical path*. Which path is critical can depend on the random choices in the scheduler. We define the *expanded critical path* (ECP) as the critical path plus all right children of vertices on the path.

Theorem 8.2. *Between any two rounds of the work stealing algorithm on a DAG G, there is at least one prime vertex that belongs to the ECP.*

Proof. (Outline) There must be exactly one ready vertex v on the critical path, and that vertex must reside in some queue. We claim that all vertices above v it in that queue are right children of the critical path, and hence on the expanded critical path.

Therefore the top element of that queue is on the ECP and prime. The right children property follows from the fact that when pushing on the bottom of the queue on a fork, we first push the right child and then the left. We will then pop the left and the right will remain. Pushing a singleton onto the bottom also maintains the property, as does popping a vertex from the bottom or stealing from the top. Hence the property is maintained under all operations on the queue. □

We can now prove our bounds on work-stealing.

Theorem 8.3. *A work-stealing schedule with p processors on a binary DAG of size w and depth d will take at most $w/p + O(d + \log(1/\epsilon))$ steps with probability $1 - \epsilon$.*

Proof. Similarly to the greedy scheduling proof we account idle processors towards the depth and busy ones towards the work. For each step i we consider the number of processors q_i with an empty queue (these are random variables since they depend on our random choices). Each processor with an empty queue will make a steal attempt. We then show that the number of steal attempts $S = \sum_{i=0}^{\infty} q_i$ is bounded by $O(pd + p\ln(1/\epsilon))$ with probability $1-\epsilon$. The work including the possible idle steps is therefore $w + O(pd + p\ln(1/\epsilon))$. Dividing by p gives the bound.

The intuition of bounding the number of steal attempts is that each attempt has some chance of stealing a prime node on the ECP. Therefore after doing sufficiently many steal attempts, we will have finished the critical path with high probability.

Consider a step i with q_i empty queues and consider a prime vertex v on that step. Each empty queue will steal v with probability $1/p$. Therefore the overall probability that a prime vertex (including one on the critical path) is stolen on step i is:

$$\rho_i = 1 - \left(1 - \frac{1}{p}\right)^{q_i} > \frac{q_i}{p}\left(1 - \frac{1}{e}\right) > \frac{q_i}{2p},$$

i.e., the more empty queues, the more likely we steal and visit a vertex on the ECP.

Let X_i be the indicator random variable that a prime node on the ECP is stolen on step i, and let $X = \sum_{i=0}^{\infty} X_i$. The expectation $E[X_i] = \rho_i$, and the expectation

$$\mu = E[X] = \sum_{i=0}^{\infty} \rho_i > \sum_{i=0}^{\infty} \frac{q_i}{2p} = \frac{S}{2p}.$$

If X reaches $2d$ the schedule must be done since there are at most $2d$ vertices on the ECP, therefore we are interested in making sure the probability $P[X < 2d]$ is small. We use the Chernoff bounds:

$$P[X < (1-\delta)\mu] < e^{-\frac{\delta^2\mu}{2}}.$$

Setting $(1-\delta)\mu = 2d$ gives $\delta = (1 - 2d/\mu)$. We then have $\delta^2 = (1 - 4d/\mu + (2d/\mu)^2) > (1 - 4d/\mu)$ and hence $\delta^2\mu > \mu - 4d$. This gives:

$$P[X < 2d] < e^{-\frac{\mu - 4d}{2}}.$$

This bounds the probability that an expanded critical path (ECP) is not finished, but we do not know which path is the critical path. There are at most 2^d possible critical paths since the DAG has binary forking. We can take the union bound over all paths giving the probability that any possible critical path is not finished is upper bounded by:

$$P[X < 2d] \cdot 2^d < e^{-\frac{\mu - 4d}{2}} \cdot 2^d = e^{-\frac{\mu}{2} + d(2 + \ln 2)}.$$

Setting this to ϵ, and given that $\mu > \frac{S}{2p}$, this solves to:

$$S < 4p(d(2 + \ln 2) + \ln(1/\epsilon)) \in O(pd + p\ln(1/\epsilon)).$$

The probability that S is at most $O(pd + p\ln(1/\epsilon)))$ is thus at least $(1 - \epsilon)$. This gives us our bound on steal attempts. □

Since each step of the work stealing algorithm takes constant time on the ACW PRAM, this leads to the following corrolary.

Corollary 8.1. *For a binary DAG of size w and depth d, and on a ACW PRAM with p processors, the work-stealing scheduler will take time*

$$O(w/p + d + \log(1/\epsilon))$$

with probability $1 - \epsilon$*.*

References

[1] S. Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992.

[2] S. Adams. Efficient sets—a balancing act. *Journal of functional programming*, 3(04), 1993.

[3] G. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962.

[4] R. Ahmed, G. Bodwin, F. D. Sahneh, K. Hamm, M. J. Latifi Jebelli, S. Kobourov, and R. Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37:100253, 2020.

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

[6] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *SC*, 2012.

[8] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2016.

[9] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. *CoRR*, abs/1903.04650, 2019.

[10] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, 1998.

[11] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979.

[12] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, 1989.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[14] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *SPAA*, 2015.

[15] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. on Computing*, 1(1):31–39, 1972.

[16] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[17] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu. Improved parallel algorithms for spanners and hopsets. In *SPAA*, 2015.

[18] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *SPAA*, pages 196–203, 2013.

[19] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973.

[20] D. Peleg and A. A. Schäffer. Graph spanners. *J. Graph Theory*, 13(1):99–116, 1989.

[21] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 1989.

[22] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[23] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. 2(1), 1981.

[24] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[25] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *SPAA*, pages 143–153, 2014.

[26] Y. Sun, D. Ferizovic, and G. E. Blelloch. PAM: parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[27] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, 1983.

[28] J. D. Ullman. Np-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.