Introduction to Parallel Algorithms (DRAFT)

Guy E. Blelloch, Laxman Dhulipala and Yihan Sun

October 16, 2019

1 Introduction

This gives a brief introduction to Parallel Algorithms. We start by discussing cost models, and then go into specific parallel algorithms.

2 Models

To analyze the cost of algorithms it is important to have a concrete model with a well defined notion of costs. Sequentially the Random Access Machine (RAM) model has served well for many years. The RAM 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 total "time" of a computation is then just the 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, we assume that for an input of size n each register and memory location can store $\Theta(\log n)$ bits.

The RAM model is by no stretch meant to model runtimes 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 would be hard to use and hard to gain intuition from. Although a RAM does not precisely model the performance of real machines, 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 should really only be used for asymptotic (i.e. big-O) analysis. Beyond serving as a cost model for algorithms that is useful for comparisons and asymptotic analysis, the RAM has some other nice features: it is simple, and, importantly, code in most high-level languages can be naturally translated into the model.

In the context of parallel algorithms we would like to use a cost model that satisfies a similar set of features. Here we use one, the MP-RAM, that we find convenient and seems to satisfy the features. It is based on the RAM, but allows the dynamic forking of new processes. It measures costs in terms of two quantities: the work, which is the total number of instructions across all processes, and the depth, which is the longest chain of sequential dependences. 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-depth, processor-time relationship:

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

where W is the work, D the depth, P the processors, and T the time.

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 parital order. The *work* of a computation is the total number of instructions, and the *depth* 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 depth of the subcomputation is the maximum of the depth 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 depth.

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, 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] := f(A[i]);
```

In pseudocode $[s : e]$ means the sequence of integers from s (inclusive) to e (exclusive), and `:=` means array 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 depth is the maximum of the depth 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
    $l$ = sum($A[:|A|/2]$) ||
    $r$ = sum($A[|A|/2:]$);
    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 code has to wait for the parallel calls 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 at the beginning (end) of the array. Taking a slices takes $O(1)$ work and depth 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 depth is the maximum of the depths 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 depth 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 loops and parallel dos can be nested in an arbitrary fashion.

Binary and 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. We therefore use MP2-RAM to indicate the version that only requires binary forking to satisfy the given bounds, and in some cases give separate bounds for MP-RAM and MP2-RAM. It is always possible to implement n-way forking using binary forking by creating a tree of binary forks of depth $\log n$. In general this can increase the depth, but in some of our algorithms it does not affect the depth. In these cases we will use `parfor2` to indicate we are using a tree to fork the n parallel calls.

Additional Instructions. In the parallel context it is useful to add some additional instructions that manipulate memory. The instructions we consider are a test-and-set (TS), fetch-and-add (FA), and priority-write (PW) and we discuss our model with these operations as the TS, FA, and PW variants of the MP-RAM. A `test_and_set(&x)` instruction takes a reference to a memory location x, checks if x is 0 and if so atomically sets it to 1 and returns *true*; otherwise it returns *false*.

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 Some Building Blocks

Several problems, like computing prefix-sums, merging sorted sequences and filtering frequently arise as subproblems when designing other parallel algorithms.

3.1 Scan

A *scan* or *prefix-sum* operation takes a sequence A, an associative operator $\oplus$, and an identity element $\perp$ and computes the sequence

$$[\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i]]$$

as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. Scan is useful because it lets us compute a value for each element in an array that depends on all previous values. We often refer to the *plusScan* operation, which is a scan where $\oplus = +$ and $\perp = 0$.

Pseudocode for a recursive implementation of scan is given in Figure 2. The code works with an arbitrary associative function f. Conceptually, the implementation

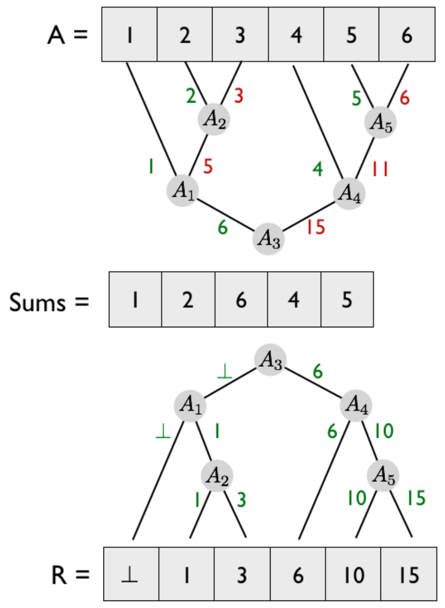

Figure 1: Recursive implementation of scan.

performs two traversals of a balanced binary tree built on the array. Both traversals traverse the tree in exactly the same way—internal nodes in the tree correspond to midpoints of subsequences of A of size greater than one. Figure 2 visually illustrates both traversals. Interior nodes are labeled by the element in A that corresponds to this index.

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, recurses on each half, writes the resulting sum from the left into $L[m-1]$, and returns the overall sum. 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 passes s, the sum of elements to the left of a node, down the tree. An internal node passes s to its left child, and passes $s + L[m]$ to its right child. Leafs in the tree write the value passed to them by the parent, which contains the partial sum of all elements to the left.

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 depth as

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

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

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

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

scan($A, f, I$) =
  $L$ = array[$|A| - 1$];
  $R$ = array[$|A|$];
  total = scanUp($A, L, f$);
  scanDown($R, L, f, I$);
  return ⟨$R$, total⟩;
```

Figure 2: The Scan function.

```
filter(A, p) =
  n = |A|;
  F = array[n];
  parfor i in [0:n]
    F[i] := p(A[i]);
  ⟨X, count⟩ = plusScan(F);
  R = array[count];
  parfor i in [0:n]
    if (F[i]) then R[X[i]] := A[i];
  return R;
```

Figure 3: The filter function.

```
flatten(A) =
  sizes = array(|A|);
  parfor i in [0:|A|]
    sizes[i] = |A[i]|;
  ⟨X, total⟩ = plusScan(sizes);
  R = array(total);
  parfor i in [0:|A|]
    off = X[i];
    parfor j in [0:|A[i]|]
      R[off + j] = A[i][j];
  return R;
```

Figure 4: The flatten function.

3.2 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 `count`, 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 depth 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,`

```
// finds which of k blocks contains v, returning block and offset
findBlock(A, v, k) =
  stride = (end-start)/size;
  result = k;
  parfor i in [0:k-1]
    if (A[i*stride] < v and A[(i+1)*stride] > v)
    then result = i;
  return (A[i*stride, (i+1)*stride], i*stride);

search(A, v, k) =
  (B, offset) = findBlock(A, v, min(|A|, k));
  if (|A| <= k) then return offset;
  else return offset + search(B, v, k);
```

Figure 5: The search function.

`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`.

3.3 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 depth. In parallel it is possible to reduce the depth, at the cost of increasing the work. The idea is to use a k-way search instead of binary search. 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 depth. By picking $k = n^\alpha$ for $0 < \alpha \leq 1$, the algorithm runs in $O(n^\alpha)$ work and $O(1/\alpha)$ depth. This algorithm requires a k-way fork and is strictly worse than binary search for the 2MP-RAM.

Another related problem is given two sorted sequences A and B, and an integer k, to find the k smallest elements. More specifically $\texttt{kth}(A, B, k)$ returns locations (l_a, l_b) in A and B such that $l_a + l_b = k$, and all elements in $A[: l_a] \cup B[: l_b]$ are less than all elements in $A[l_a :] \cup B[l_b :]$. 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 depth.

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 depth. 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

```
kthHelp(A, aoff, B, boff, k) =
  if (|A| + |B| == 0) then return (aoff,boff);
  else if (|A| == 0) then return (aoff, boff + k);
  else if (|B| == 0) then return (aoff + k, boff);
  else
    amid = |A|/2;   bmid = |B|/2;
    case (A[amid] < B[bmid], k > amid + bmid) of
      (T,T) ⇒ return kthHelp(A[amid+1:], aoff+amid+1, B, boff, k-amid-1);
      (T,F) ⇒ return kthHelp(A, aoff, B[:bmid], boff, k);
      (F,T) ⇒ return kthHelp(A, aoff, B[bmid+1:], boff+bmid+1, k-bmid-1);
      (F,F) ⇒ return kthHelp(A[:amid], aoff, B, boff, k);

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

Figure 6: The kth function.

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 $k = n^\alpha$ this will result in an algorithm taking $O(n^{2\alpha})$ work and $O(1/\alpha^2)$ depth. As with the constant depth sorted array search problem, this does not work on the 2MP-RAM.

3.4 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. Here we describe a few different algorithms for the problem.

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 2MP-RAM. 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 2MP-RAM.

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

Figure 7: 2-way D&C merge.

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

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

The depth 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 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 2MP-RAM since the `parFor` can be done with binary 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 3.1. *Merging can be solved in $O(n)$ work and $O(\log n)$ depth in the 2MP-RAM and $O(n)$ work and $O(\log \log n)$ depth 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 $O(\log n)$ depth on the 2MP-RAM. When used with a small constant, e.g. $f(n) = .1 \times n/\log n$, this version works well in practice.

3.5 K-th Smallest

The k-th smallest problem is to find the k-smallest elemennt in an sequences. Figure 9 gives an algorithm for the problem. The peformance 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

```
kthSmallest(A, k) =
  $p$ = selectPivot($A$);
  $L$ = filter($A$, $\lambda x.(x < p)$);
  $G$ = 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($A, l$) =
  $l = f(|A|)$;
  $m = (|A| - 1)/l + 1$;
  B = array[$m$];
  parfor $i$ in $[0 : m]$
    $s = i \times l$;
    $B[i]$ = kthSmallest($A[s : s + l], l/2$);
  return kthSmallest($B, m/2$);
```

Figure 10: Randomized and deterministic pivot selection.

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 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 depth $O(\log n)$ for an array of size n, the overall depth is $O(\log|A|^2)$ w.h.p.. The work is $O(|A|)$ in expectation. The algorithm runs on the 2MP-RAM.

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 satisifies 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 depth 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 depth $D_{sort}(n)$ this gives the recurrences:

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

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 depth to $D(n) = O(D_{sort}(n) \log\log n)$.

4 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 like mergesort, quicksort, samplesort and radix-sort. We also discuss related problems like semisorting and parallel integer sorting.

4.1 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[:mid]` and `A[mid:]` 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 3 to merge the results of the two recursive calls. The work of `mergesort` is given by the following recurrence:

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

which solves to $O(n \log n)$, and the depth as

$$D(n) = D(n/2) + O(\log^2 n)$$

which solves to $O(\log^3 n)$. The $O(n)$ term in the work recurrence and the $O(\log^2 n)$ term in the depth recurrence are due to the merging the results of the two recursive calls.

The parallel merge from Section 3 can be improved to run in $O(n)$ work and $O(\log n)$ depth which improves the depth of this implementation to $O(\log^2 n)$. We give pseudocode for the merge with improved depth in Figure 8. The idea is to recurse on $\sqrt{n}$ subproblems, instead of just two subproblems. The i'th subproblem computes the ranges `[as, ae]` and `[bs, be]` s.t. `A[as:bs]` and `B[bs:be]` contain the $i\sqrt{n}$ to the $(i+1)\sqrt{n}$'th elements in the sorted sequence. The work for this implementation is given by the recurrence

$$W(m) = \sqrt{m}(W(\sqrt{m})) + O(\sqrt{m}\log m)$$

```
mergesort(A) =
  if (|A| == 1) then return A;
  else
    mid = |A|/2;
    l = mergesort(A[:mid]) ||
    r = mergesort(A[mid:]);
    return merge(l, r);
```

Figure 11: Parallel mergesort.

```
quicksort(A) =
  if (|A| == 1) then return A;
  else
    p = select_pivot(A);
    e = filter(A, $\lambda x.(x = p)$);
    l = quicksort(filter(A, $\lambda x.(x < p)$)) ||
    r = quicksort(filter(A, $\lambda x.(x > p)$));
    return flatten([l, e, r]);
```

Figure 12: Parallel quicksort.

which solves to $O(m)$ and the depth by

$$D(m) = D(\sqrt{m}) + O(\log m)$$

which solves to $O(\log m)$. The $O(\log m)$ term in the depth is for the binary search. Note that if we use binary-forking, we can still fork $O(\sqrt{m})$ tasks within in $O(\log m)$ depth without increasing the overall depth of the merge.

4.2 Quicksort and Samplesort

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. The depth of this algorithm can be preciscely analyzed using, for example, Knuth's technique for randomized recurrences. Instead, if we optimistically assume that each choice of pivot splits A approximately in half, we get the depth recurrence:

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

which solves to $O(\log^2 n)$. The $O(\log n)$ term in the depth recurrence is due to the calls to filter and flatten.

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).

4.3 Radix sort

Radix sort is a comparison based sort that performs very well in practice, and is commonly used as a parallel sort when the maximum integer being sorted is bounded. Unlike comparison-based sorts, which perform pairwise comparisons on the keys to determine the output, radix-sort interprets keys as b-bit integers and performs a sequence of stable sorts on the keys. As each of the intermediate sorts is stable, the output is stably sorted [1]

Pseudocode for a single bit at-a-time parallel bottom-up radix sort (sorts from the least-significant to the most-significant bit) is given in Figure 13. The code performs b sequential iterations, each of which perform a stable sort using a `split` operation. `split` takes a sequence A, and a predicate p and returns a sequence containing all elements not satisfying p, followed by all elements satisfying p. `split` can be implemented stably using two `plusScan`s. As we perform b iterations, where each iteration performs $O(n)$ work $O(\log n)$ depth, the total work of this algorithm is $O(bn)$ and the depth is $O(b \log n)$. For integers in the range $[0, n]$, this integer sort which sorts 1-bit at a time runs in $O(n \log n)$ work and $O(\log^2 n)$ depth, which is not an improvement over comparison sorting.

Sequentially, one can sort integers in the range $[0, n^k]$ in $O(kn)$ time by chaining together multiple stable counting sorts (in what follows we assume distinct keys for simplicity, but the algorithms generalize to duplicate keys as expected). The algorithm sorts $\log n$ bits at a time. Each $\log n$ bit sort is a standard stable counting sort, which runs in $O(n)$ time. Unfortunately, we currently do not know how to efficiently parallelize this algorithm. Note that the problem of integer sorting keys in the range $[0, n^k]$ is reducible to stably sorting integers in the range $[0, n]$. The best existing work-efficient integer sorting algorithm can unstably sort integers in the range $[0, n \log^k n)]$ in $O(kn)$ work in expectation and $O(k \log n)$ depth with high probability [17].

Using the same idea as the efficient sequential radix-sort we can build a work-efficient parallel radix sort with polynomial parallelism. We give psueodocode for this algorithm in Figure 14. The main substep is an algorithm for stably sorting $\epsilon \log n$ bits in $O(n)$ work and $n^{1-\epsilon}$ depth. Applying this step $1/\epsilon$ times, we can sort keys in the range $[1, n]$ in $O(n)$ work and $n^{1-\epsilon}$ depth. At a high level, the algorithm just breaks the array into a set of blocks of size $n^{1-\epsilon}$, computes a histogram within each block for the n^{ϵ} buckets, and then transposes this matrix to stably sort by the buckets.

We now describe the algorithm in Figure 14 in detail. The algorithm logically breaks the input array into n^{ϵ} blocks each of size $n^{1-\epsilon}$. We allocate an array `H`, initialized to all 0, which stores the histograms for each block. We first map over all blocks in parallel and sequentially compute a histogram for the n^{ϵ} buckets within each block. The sequential histogram just loops over the elements in the block and increments a counter for the correct bucket for the element (determined by $\epsilon \log n$ bits of the element). Next, we perform a transpose of the array based on the histograms within each block. We can perform the transpose using a strided-scan with $+$; a strided scan

[1]Stable sorting is important for chaining multiple sorts together over the same sequence.

```
radix_sort(A, b) =
  for i in [0:b]
    A = split(A, lambda x.(x >> i) mod 2);
```

Figure 13: Parallel radix sort (one bit at-a-time).

just runs a scan within each bucket across all blocks. The outputs of the scan within each bucket are written to the array for the `num_blocks`'th block, which we refer to as `all_bkts` in the code. We `plusScan` this array to compute the start of each bucket in the output array. The last step is to map over all blocks again in parallel; within each block we sequentially increment the histogram value for the element's bucket, add the previous value to the global offset for the bucket to get a unique offset and finally write the element to the output. Both of the `parfor`s perform $O(n)$ work and run $O(n^{1-\epsilon})$ depth, as the inner loop sequentially processes $O(n^{1-\epsilon})$ elements. The strided scan can easily be performed in $O(n)$ work and $O(\log n)$ depth. Therefore, one `radix_step` can be implemented in $O(n)$ work and $O(n^{1-\epsilon})$ depth. As the `radix_sort` code just calls `radix_step` a constant number of times, `radix_sort` also runs in $O(n)$ work and $O(n^{1-\epsilon})$ depth. We can sort keys in the range $[1, n^k]$ in $O(kn)$ work and $O(kn^{1-\epsilon})$ depth by just running `radix_sort` on $\log n$ bits at a time.

4.4 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. Gu, Shun, Sun and Blelloch [12] 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 $h : U \to [1, \dots, n^k]$ for some constant k, and an equality function on keys, $f : U \times U \to$ `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 $O(\log n)$ depth w.h.p. on the TS-MP-RAM.

5 Graph Algorithms

In this section, we present parallel graph algorithms for breadth-first search, low-diameter decomposition, connectivity, maximal independent set and minimum spanning tree which illustrate useful techniques in parallel algorithms such as random-

```
radix_step(A, num_buckets, shift_val) =
  get_bkt = lambda x.(x >> shiftval) mod num_buckets);
  block_size = floor(|A| / num_buckets);
  num_blocks = ceil(|A| / block_size);
  H = array(num_buckets * (num_blocks+1), 0);
  parfor i in [0:num_blocks]
    i_hist = H + i*num_buckets;
    for j in [i*block_size, min((i+1)*block_size, |A|)]
      i_hist[get_bkt(A[j])]++;

  strided_scan(H, num_buckets, num_blocks+1);
  all_bkts = array(H + num_blocks*num_buckets, num_buckets);
  plus_scan(all_bkts, all_bkts);

  R = array(|A|);
  parfor i in [0:num_blocks]
    i_hist = H + i*num_buckets;
    for j in [i*block_size, min((i+1)*block_size, |A|)]
      j_bkt = get_bkt(A[j]);
      bkt_off = i_hist[j_bkt]++;
      global_off = all_bkts[j_bkt];
      R[global_off + bkt_off] = A[j];
  return R;

radix_sort(A, num_buckets, b) =
  n_bits = log(num_buckets);
  n_iters = ceil(b / n_bits);
  shift_val = 0;
  for iters in [0:n_iters]
    A = radix_step(A, num_buckets, shift_val);
    shift_val += n_bits;
  return A;
```

Figure 14: A parallel radix sort with polynomial depth.

```
edge_map(G, U, update) =
  nghs = array(|U|, <>);
  parfor i in [0, |U|]
    v = U[i];
    out_nghs = G[v].out_nghs;
    update_vtx = lambda x.update(v, x);
    nghs[i] = filter(out_nghs, update_vtx);
  return flatten(nghs);
```

Figure 15: `edge_map`.

ization, 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.

5.1 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 `edge_map` in the style of Ligra [19].

`edge_map` 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 `edge_map` 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 `edge_map` based on flatten in Figure 15. 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 [6, 19] discuss theoretically efficient and practically efficient implementations of `edge_map`.

5.2 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 [11].

```
BFS(G(V, E), v) =
  n = |V|;
  frontier = array(v);
  visited = array(n, 0); visited[v] = 1;
  parents = array(n, -1);
  update = lambda (u, v).
    if (!visited[v] && test_and_set(&visited[v]))
      parents[v] = u;
      return true;
    return false;
  while (|frontier| > 0):
    frontier = edge_map(G, frontier, update);
  return parents;
```

Figure 16: Parallel breadth-first search.

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 16. The algorithm first creates an initial frontier which just consists of v, initializes a visited array to all 0, and a parents arrray to all -1 and marks v as visited. We perform a BFS by looping while the frontier is not empty and applying edge_map on each iteration to compute the next frontier. The update function supplied to edge_map 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.

5.3 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 cuts, 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 and 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 [4], 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 [4]. The algorithm repeatedly picks an arbitrary uncovered vertex v and grows a ball around it using breadth-first search until the number of edges incident to the current frontier is at most a β fraction of the number of internal edges. As each edge is examined once, this results in an $O(n+m)$ time sequential algorithm. One can prove that the diameter of a ball grown in this manner is $O(\log n/\beta)$.

Miller, Peng and Xu [15] give a work-efficient randomized algorithm for low-diameter decomposition based on selecting *randomly shifted start times* from the exponential distribution. Their algorithm works as follows: for each $v \in V$, the algorithm draws a start time, δ_v, from an exponential distribution with parameter β. The clustering is done by assigning each vertex u to the center v which minimizes $d(u,v) - \delta_v$. We will sketch a high-level proof of their algorithm, and refer the reader to [15, 21] for related work and full proofs.

Recall that the exponential distribution with a rate parameter λ. Its probability density function is given by

$$f(x,\lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The mean of this distrubtion is $1/\lambda$. The LDD algorithm makes use of the *memoryless property* of the exponential distribution, which states that if $X \sim Exp(\beta)$ then

$$\mathbf{Pr}[X > m + n | X \geq m] = \mathbf{Pr}[X > n]$$

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 $\lfloor \delta_{\max} - \delta_v \rfloor$ steps have elapsed and it is not yet covered by another vertex's BFS. Ties between different searches can be deterministically broken by comparing the δ_v's. Alternately, we can break the ties non-deterministically which increases the number of cut edges by a constant factor in expectation, leading to an $(2\beta, O(\log n/\beta))$ decomposition in the same work and depth.

Figure 19 shows pseudocode for the Miller-Peng-Xu based on breaking ties deterministicaly. The algorithm computes a $(\beta, O(\log n/\beta))$ decomposition in $O(m)$ work and $O(\log^2 n)$ depth w.h.p. on the TS-MP-RAM. We first draw independent samples from $Exp(\beta)$ and compute `S`, the start time for each vertex. The array `C` holds a tuple containing the shifted distance and the cluster id of each vertex, which are both initially ∞. In each round, we add all vertices that have a start time less than the current round and are not already covered by another cluster to the current frontier, `F`. Next, we compute the next frontier by performing two `edge_maps`. The first `edge_map` performs a priority-write the fractional bits of the start time of the cluster center for $u \in$ `F` to an unvisited neighbor $v \in N(u)$. The second `edge_map` checks whether u successfully acquired its neighbor, v, and sets the cluster-id of v to the cluster-id of v if it did, returning `true` to indicate that v should be in the output vertex subset.

We first argue that the maximum radius of each ball is $O(\log n/\beta)$ w.h.p. We can see this easily by noticing that the starting time of vertex v is $\delta_{\max} - \delta_v$, and as each start time is ≥ 0, all vertices will have "woken up" and started their own cluster after $\delta_{\max}$ rounds. Next, we argue that the probability that all vertices haven't woken up after $\frac{c \log n}{\beta}$ rounds can be made arbitrarily small. To see this, consider the probability that a single vertex picks a shift larger than $\frac{c \log n}{\beta}$:

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

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:

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

and therefore

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

The next step is to argue that at most βm edges are cut in expectation. The MPX paper gives a rigorous proof of this fact using the order statistics of the exponential distribution. We give a shortened proof-sketch here that conveys the essential ideas of the proof. The proof will show that the probability that an arbitrary edge $e = (u, v)$ is cut is $< \beta$. Applying linearity of expectation across the edges then gives that at most βm edges are cut in expectation.

First, we set up some definitions. Let c be the 'midpoint' of the edge (u, v), where the (u, c) and (v, c) edges each have weight 0.5. Now, we define the shifted distance d_v to the midpoint c from $v \in V$ as $d_v = \delta_{\max} - \delta_v + dist_G(v, c)$. That is, the shifted distance is just the start time of the vertex plus the distance to c. Clearly, the vertex that minimizes the shifted distance to c is vertex which acquires c. The center which acquires c can also be written as $\max_{v \in V} \rho_v$ where $\rho_v = \delta_v - dist_G(v, c)$. Let $\hat{\rho}_i$ be the value of the i'th largest ρ_i.

Next, notice that the edge (u, v) is cut exactly when the difference between largest $\hat{\rho}_n$ and $\hat{\rho}_{n-1}$ (the largest ρ_v and second largest ρ_v) is less than 1. We can bound this probability by showing that the difference $\hat{\rho}_n - \hat{\rho}_{n-1}$ is also an exponential distribution with parameter β (this can be shown by using the memoryless property of the exponential distribution, see Lemma 4.4 from [15]). The probability is therefore

$$\mathbf{Pr}[\hat{\rho}_n - \hat{\rho}_{n-1} < 1] = 1 - e^{-\beta} < \beta$$

where the last step uses the taylor series expansion for e^x.

```
LDD(G(V, E), beta) =
  n = |V|; num_finished = 0;
  E = array(n, lambda i.Exp(beta));
  C = array(n, -1);
  parfor i in [0:n]
    C[i] = v in V minimizing (d(v, i) - E[v]);
  return C;
```

Figure 17: Low-diameter decomposition.

```
LDD(G(V, E), beta) =
  n = |V|; num_finished = 0;
  E = array(n, lambda i.return Exp(beta));
  S = array(n, lambda i.return max(E) - E[i]);
  C = array(n, (infty, infty));
  num_processed = 0; round = 1;
  while (num_processed < n)
    F = F ∪ {v in V | S[v] < round, C[v] == infty};
    num_processed += |F|;
    update = lambda (u,v).
      cluster_u = C[u].snd;
      if (C[v].snd == infty)
        writeMin(&C[v].fst, frac(S[cluster_u]));
      return false;
    edge_map(G, F, update);
    check = lambda (u,v).
      cluster_u = C[u].snd;
      if (C[v].fst == frac(S[cluster_u]))
        C[v].snd = cluster_u;
        return true;
      return false;
    F = edge_map(G, F, check);
    round++;
  return C;
```

Figure 18: Deterministic low-diameter decomposition.

```
Connectivity(G(V, E), beta) =
  L = LDD(G, beta);
  G'(V',E') = Contract(G, L);
  if (|E'| == 0)
    return L
  L' = Connectivity(G', beta)
  L'' = array(n, lambda v.return L'[L[v]];);
  return L'';
```

Figure 19: Parallel connectivity.

5.4 Connectivity

6 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 [7]. *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*.

6.1 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

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 1: **Summary of notation**.

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 [5].. 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 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) [16] 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 [16, **?**]. 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. [18] 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 1.

6.2 The *join* Algorithms for Each Balancing Scheme

Here we describe algorithms for *join* for the four balancing schemes we defined in Chapter 6.1, 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.

6.2.1 AVL Trees

```
joinRightAVL($T_l, k, T_r$) {
  $(l, k', c)$ = expose($T_l$);
  if $h(c) \leq h(T_r) + 1$ then {
    $T'$ = node($c, k, T_r$);
    if $h(T') \leq h(l) + 1$ then return node($l, k', T'$);
    else return rotateLeft(node($l, k'$, rotateRight($T'$)));
  } else {
    $T'$ = joinRightAVL($c, k, T_r$);
    $T''$ = node($l, k', T'$);
    if $h(T') \leq h(l) + 1$ then return $T''$; else return rotateLeft($T''$); }}
join($T_l, k, T_r$) {
  if $h(T_l) > h(T_r) + 1$ then return joinRightAVL($T_l, k, T_r$);
  else if $h(T_r) > h(T_l) + 1$ then return joinLeftAVL($T_l, k, T_r$);
  else return node($T_l, k, T_r$); }
```

Figure 20: **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 20 and illustrated in Figure 21. 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 6.1. *The join algorithm in Figure 20 on AVL trees requires at most two rotations.*

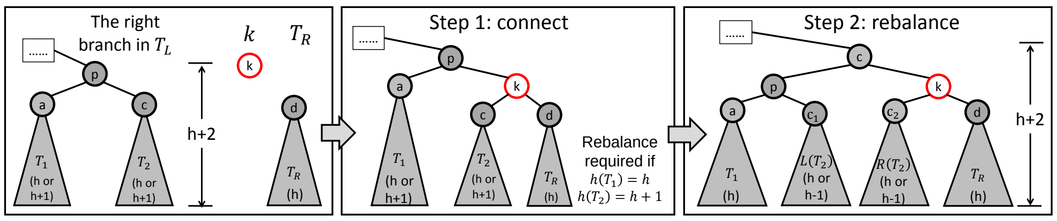

Figure 21: **An example for *join* on AVL trees** – An example for *join* on AVL trees ($h(T_l) > h(T_r)+1$). We first follow the right spine of T_l until a subtree of height at most $h(T_r)+1$ is found (i.e., T_2 rooted at c). Then a new $node(c, k, T_r)$ is created, replacing c (Step 1). If $h(T_1) = h$ 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.

Lemma 6.2. *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 $1 + \max(h(T_l), h(T_r))$.*

Proof. Since the algorithm only visits nodes on the path from the root to c, and only requires at most two rotations (Lemma 6.1), 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. $\square$

6.2.2 Red-black Trees

Tarjan describes how to implement the *join* function for red-black trees [**?**]. Here we describe a variant that does not assume the roots are black (this is to bound the

```
joinRightRB($T_l, k, T_r$) {
  if ($r(T_l) = \lfloor r(T_r)/2 \rfloor \times 2$) then return node($T_l, \langle k, \texttt{red}\rangle, T_r$); 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($lc(T'), \langle k(T'), \texttt{black}\rangle, rc(T')$);
    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($lc(T'), \langle k(T'), \texttt{black}\rangle, rc(T')$);
    else return $T'$;
  } else {
    if (k is a increase-2 node) then
      return node($T_l, \langle k, \texttt{black}\rangle, T_r$);
    else if ($color(T_l)$=black) and ($color(T_r)$=black)
      return node($T_l, \langle k, \texttt{red}\rangle, T_r$);
    else return node($T_l, \langle k, \texttt{black}\rangle, T_r$); }
}
```

Figure 22: **The *join* algorithm on red-black trees** – The *join* algorithm on red-black trees. `joinLeftRB` is symmetric to `joinRightRB`.

increase in rank by *union*). The pseudocode is given in Figure 22. 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 on v. The update is illustrated in Figure 23. 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 6.3. *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 23, 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 23). 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 23. The black height remains, but the root changed from black to red, increasing the rank by 1.

□

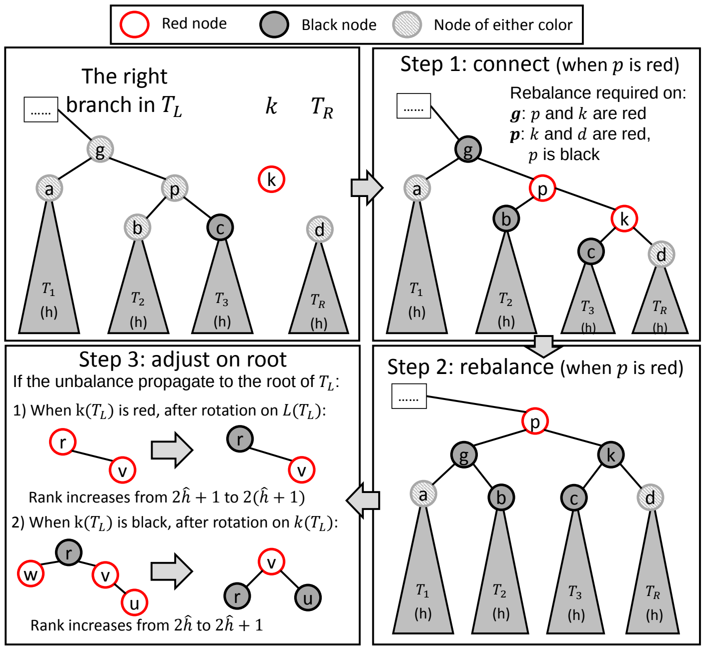
Red node
Black node
Node of either color
The right branch in T_L
k
T_R
Step 1: connect (when p is red)
Rebalance required on:
g: p and k are red
p: k and d are red, p is black
Step 2: rebalance (when p is red)
Step 3: adjust on root
If the unbalance propagate to the root of T_L:
1) When k(T_L) is red,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

Figure 23: **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).

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

6.2.3 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 24. 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 25) at each node assuming α is within the bounds given in Section 6.1.

Lemma 6.4. *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.*

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 [16, ?], 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.

6.2.4 Treaps

The treap *join* algorithm (as in Figure 26) 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

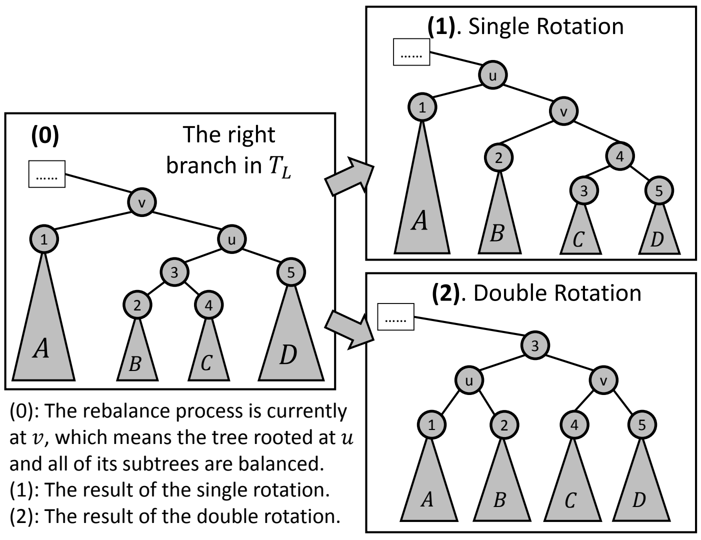
(1). Single Rotation
(0)
The right branch in T_L
(2). Double Rotation
(0): The rebalance process is currently at v, which means the tree rooted at u and all of its subtrees are balanced.
(1): The result of the single rotation.
(2): The result of the double rotation.

Figure 25: **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.

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

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 6.5. *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 2: **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.

6.3 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 27: ***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 29. 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 30.

6.3.1 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 28 gives an example.

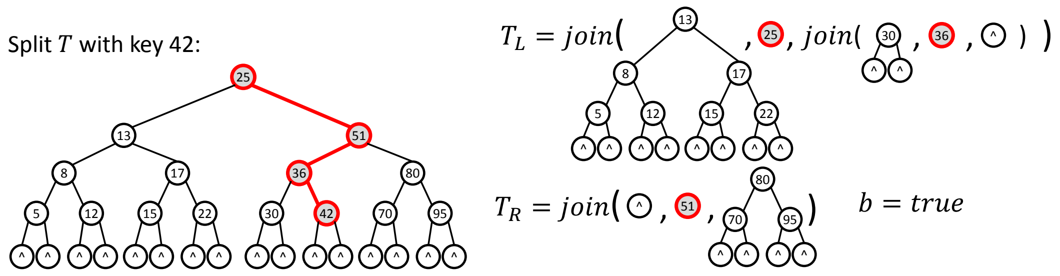

Figure 28: **An example of *split* in a BST with key** 42 – We first search for 42 in the tree and split the tree by the searching path, then use *join* to combine trees on the left and on the right respectively, bottom-top.

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

Theorem 6.1. *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 6.5, the complexity of each *join* is $O(1)$ and the work of split is at most $O(h(T))$. □

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 27, 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 6.2. *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.

6.4 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,

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 29: ***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.

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 [10]. 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

[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.

algorithm, at least three libraries use Adams' algorithms for their implementation of ordered sets and tables (Haskell [14] 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 [9], 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. [7] 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 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 6.3. *For AVL, RB, WB trees and treaps, the work and span of the algorithm (as shown in Figure 29) of union, intersection or difference on two balanced BSTs of sizes m and n ($n \geq m$) is $O\left(m \log\left(\dfrac{n}{m}+1\right)\right)$ (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 $\log\binom{m+n}{n} = \Theta\left(m \log\left(\frac{n}{m}+1\right)\right)$ [13]. A generic proof of Theorem 6.3 working for all the four balancing schemes can be found in [7]. The span of these algorithms can be reduced to $O(\log m)$ for weight-balanced trees even on the binary-forking model [8] by doing a more complicated divide-and-conquer strategy.

6.5 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

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$\rightarrow$lc, A, b) ||
    R = multi_insert_s(r$\rightarrow$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 30: **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.

independent of the balancing scheme.

We present the pseudocode in Figure 30 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 counter), the *join*-based insertion algorithm is only 17% slower sequentially than the highly-optimized C++ STL library [20].

Theorem 6.4. *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. $\square$

Theorem 6.5. *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 6.4. The only exception is that at most one *join2* algorithm can be performed. This only adds an extra $O(\log n)$ cost. $\square$

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 2.

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 30. 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 [7]. 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.

7 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.

7.1 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.

7.2 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 [**?**] 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 7.1. *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.

7.3 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 7.1. *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.

7.4 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

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

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 starts with the root of the DAG in one of the queues and the rest empty. Pseudocode for the algorithm is given in Figure 31. 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 7.2. *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. $\square$

We can now prove our bounds on work-stealing.

Theorem 7.3. *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. $\square$

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

Corollary 7.1. *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] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

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

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

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

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

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

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

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

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

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

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

[21] S. C. Xu. *Exponential Start Time Clustering and its Applications in Spectrual Graph Theory*. PhD thesis, Carnegie Mellon University, 2017.