Introduction to Parallel Algorithms
15-853 : Algorithms in the Real World
(DRAFT)

Guy E. Blelloch and Laxman Dhulipala

March 23, 2018

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 is 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

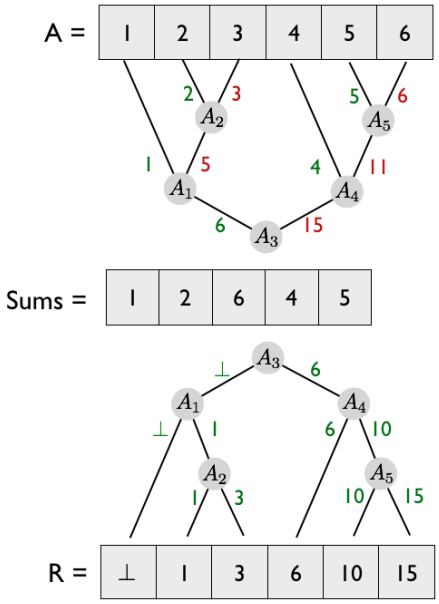

Figure 1: Recursive implementation of scan.

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, 5, 1, 7, 8]`.

```
// 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.

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 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

```
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.

$\log_k |A| + \log_k |B|$ 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 × l, |R|);
      e = min((i + 1) × 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, λx.(x < p));
  G = filter(A, λx.(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 × 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, λx.(x = p));
    l = quicksort(filter(A, λx.(x < p))) ||
    r = quicksort(filter(A, λx.(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 [4].

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 [2] 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 $h : U \to [1, \ldots, n^k]$ 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 randomization, pointer-jumping, and contraction. Unless otherwise specified, all graphs are

```
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`.

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 [**?**].

`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 $\texttt{update}(u, v) = \texttt{true}$. 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. $\texttt{update}(u, v) = \texttt{true}$ 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 [**?**, **?**] 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 [**?**].

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

```
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.

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 [1], 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 [1]. 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 [3] 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 [3, 5] 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_map`s. 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 [3]). 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.

5.4 Connectivity

6 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

```
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.

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.

6.1 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.

6.2 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 6.1. *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.

6.3 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 6.1. *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.

```
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(v1, v2) ⇒ pushBot(Q[i], v2); pushBot(Q[i], v1);
14           next(v) ⇒ pushBot(Q[i], v);
```

Figure 20: Work stealing scheduler. The processors need to synchronize between line 7 and the next line, and between the two phases.

6.4 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 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 20. 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 6.2. *Between any two rounds of the work stealing algorithm on a DAG G, there is at least one prime vertex that belongs to the ECP.*

Proof. (Outline) There must be exactly one ready vertex v on the critical path, and that vertex must reside in some queue. We claim that all vertices above v it in that queue are right children of the critical path, and hence on the expanded critical path. Therefore the top element of that queue is on the ECP and prime. The right children property follows from the fact that when pushing on the bottom of the queue on a fork, we first push the right child and then the left. We will then pop the left and the right will remain. Pushing a singleton onto the bottom also maintains the property, as does popping a vertex from the bottom or stealing from the top. Hence the property is maintained under all operations on the queue. □

We can now prove our bounds on work-stealing.

Theorem 6.3. *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:

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

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

$$\mu = E[X] = \sum_{i=0}^{\infty} \rho_i = \sum_{i=0}^{\infty} \frac{q_i}{p} \left(1 + \frac{1}{e}\right) = \frac{S}{p} \left(1 + \frac{1}{e}\right).$$

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 $\mu = \frac{S}{p}\left(1 + \frac{1}{e}\right)$, this solves to:

$$S > \frac{2p(d(2 + \ln 2) + \ln(1/\epsilon))}{1 - 1/e}.$$

This indicates that any S larger than the right hand quantity has probability at most ϵ. The probability that S is at most $O(pd + p\ln(1/\epsilon)))$ is thus at least $(1 - \epsilon)$. This gives us our bound on steal attempts. □

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

Corollary 6.1. *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] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

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

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

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

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