CS584/684 Algorithm Analysis and Design – Spring 2017
Week 3: Multithreaded Algorithms

Additional sources for this lecture include:

1. Umut Acar and Guy Blelloch, *Algorithm Design: Parallel and Sequential* (`http://www.parallel-algorithms-book.com`).
2. Guy Blelloch, "Prefix Sums and Their Applications," in J.H. Reif, ed.,*Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1991.

1 Introduction

We now briefly consider how to handle multiple parallel processors when designing an algorithm for a problem. This is a big topic, worthy of a course all to itself—and traditionally, study of parallel algorithms has been a somewhat separate subject. Given the fundamentally parallel nature of the physical world—for example, hardware circuits naturally show parallel behavior, and can only be constrained to behave sequentially via lots of effort—parallelism has been widely studied. Moreover, there have been commercial parallel multiprocessors for many decades. But in today's commodity multicore environment, with multiple cores in most laptops and many smart phones, there's a good case to be made that any study of algorithms should take parallelism into account as an added dimension from the very beginning.

Before we can start writing and analyzing parallel algorithms, we need to know the computation model under which they will run. Unlike the situation for sequential algorithms, there are quite a few alternative parallel computation models, that have greater or lesser fidelity to the realities of parallel hardware. CLRS adopts a fairly simple model based on dynamically nested threads, communicating via shared memory, and synchronized using a fork-join discipline. This model corresponds to popular and widely-available language-level facilities for parallelism.

The basic operations are:

- **spawn** (often called *fork*): create a new thread of control that can run in parallel with current thread; and
- **sync** (often called *join*): block until all children of current thread have completed

A further convenience operation is the **parallel loop**, but this is really just shorthand for a tree of spawn operations (see CLRS).

One forked, computations run asynchronously until their join point, so each thread behaves independently of other threads in the system. Note that the thread structure of the computation can evolve dynamically, based on computed data values. The key feature here is that subcomputations compose either *sequentially* or *in parallel*.

This model simplifies away many things, including:

- The mechanics of scheduling threads onto processors. A major issue in parallel code is load balancing among the available processors. Here we assume the existence of a scheduler that is responsible for this task. To avoid making unrealistic demands on the performance of such a scheduler, we generally assume that it uses a simple *greedy* algorithm: whenever there is a processor free and a task ready to execute, it assigns the task to the processor and starts running it immediately.

- The overhead of fork/join operations and of the scheduler. These are presumably just constants, but they can be large. A well-known consequence is that real systems cannot benefit from parallelism unless the *granularity* of threads is sufficiently large. We generally ignore this issue, however, even though it has some impact on algorithm design (much as it does in the sequential case where we might include a cut-off point to stop doing divide-and-conquer after sub-problems get sufficiently small).
- Non-uniform memory access time. The model assumes that every processor can access every memory address in constant time; in reality, this assumption becomes unreasonable as the number of processors grows large. Fancier models for memory access allow costs to vary according to locality of the data.

In the realm of asymptotic analyis, these simplifications don't matter too much, except perhaps for the last one. But they can have big effects on the constants, so in practical parallel programming, they are quite important.

One less attractive aspect of this model is that it is easy to make mistakes when programming shared-memory systems; *data races* among threads that access the same memory locations in an unsynchronized way can lead to ill-defined algorithms or programs. We can get around this if necessary by restricting ourselves to *pure* (side-effect free) computations. Even algorithms that appear to depend fundamentally on mutation can generally be re-cast systematically into the pure world at a cost of no more than $O(\log n)$ additional time. [How?]

There are alternative models. One well-known one is the PRAM (Parallel Random Access Machine), a synchronous model in which all processors run under a common clock. Each processor has access to its unique processor id number, so it can behave differently from the other processors by testing that number. But PRAMs are typically programmed in SIMD (single instruction multiple data) style: all processors run the same program in lock-step, but access different addresses computed based on the processor number. Because it treats processors rather than threads as fundamental, PRAM programs must explicitly consider scheduling and load balancing issues, so they are comparatively hard to write—particularly when sub-problem sizes vary a lot, as is the case for divide-and-conquer algorithms.

2 Work, Span, and All That

The two fundamental concepts we define for the CLRS model are:

- **Work.** This is the total number of steps needed for a computation summed over all processors.
- **Span.** This is the number of steps on the longest sequential path in the computation (the *critical path*).

If we write T_P for the time (number of steps) needed for the computation running on P processors, we can write T_1 for the work and T_∞ for the span.

The *(average) parallelism* of an algorithm $\mathbb{P} = T_1/T_\infty$; it tells us roughly how many processors that can be efficiently employed in running the algorithm. (CLRS uses the related quantity "slackness," defined as $\mathbb{P}/P$.)

Our basic design goal is *first* to minimize work and *then* maximize parallelism. As long as P is effectively a constant wrt/ n, minimizing asymptotic work remains most important, because we get at most a constant speedup by parallelization. (Although as always, sometimes the constants are more important than the asymptotic behavior on problem sizes of interest.) A *work-efficient* parallel algorithm is one whose work is asymptotically the same as the best known sequential algorithm; we should always aim for work-efficient algorithms.

The *speedup* of an algorithm is T_1/T_P; it says how much faster the parallel algorithm gets when we use P processors instead of 1. *Perfect (or linear)* speedup is P.

From now on, we will assume that we use a greedy scheduler. It is easy to prove that doing so guarantees a running time

$$T_P \leq T_1/P + T_\infty$$

where first component comes from complete steps (processors saturated) and second component comes from incomplete steps (which must reduce the remaining critical path by one).

In fact, greedy scheduling is (asymptotically) quite effective. The best we can hope to do [why?] with an optimal schedule is

$$T_P^* = \max(T_1/P, T_\infty)$$

so we have

$$T_P \leq 2 \cdot \max(T_1/P, T_\infty) \leq 2T_P^*$$

So at worst, the greedy scheduler makes things only twice as slow as an ideal scheduler. Moreover, since $T_\infty = T_1/\mathbb{P}$, we have

$$T_P \leq \frac{T_1}{P} + \frac{T_1}{\mathbb{P}} = \frac{T_1}{P}\left(1 + \frac{P}{\mathbb{P}}\right)$$

so if $\mathbb{P} \gg P$ (parallelism is much greater than number of processors) then $T_P \approx T_1/P$, so the speedup $T_1/T_P \approx P$, i.e. nearly perfect.

3 Simple Example: Finding a Minimum

Suppose we wish to find the minimum of an array of integers. We know that this requies $\Theta(n)$ sequential time, using a simple scan of the array (see CLRS p. 214). There is no obvious way to parallelize that scan, however.

When trying to parallelize, it is often nice to start from a a divide-and-conquer algorithm. So here is one:

```
MINIMUM-DC(A, p, r)
1  if p == r
2       return A[p]
3  q = ⌊(p + r)/2⌋
4  l = MINIMUM-DC(A, p, q)
5  r = MINIMUM-DC(A, q + 1, r)
6  return min(l, r)
```

As usual, we kick-start the procedure by invoking MINIMUM-DC$(A, 1, A.\mathit{length})$.

Since all the non-recursive steps take just unit time, we have the following recurrence for the running time (where $n = r - p + 1$):

$$T(n) = 2T(n/2) + \Theta(1)$$

which is clearly $\Theta(n)$ (e.g. by the recursion tree method). So as a sequential algorithm, this is no better than the scan method, and it is certainly more complicated.

But it has the advantage of supporting easy parallelization, by simply inserting appropriate fork/join operations:

MINIMUM-P(A, p, r)

```
1  if p == r
2       return A[p]
3  q = ⌊(p + r)/2⌋
4  l = spawn MINIMUM-P(A, p, q)
5  r = MINIMUM-P(A, q + 1, r)
6  sync
7  return min(l, r)
```

Let us analyze the running time of this algorithm. The work T_1 is just the same as the running time of the *serialized* version, i.e., MINIMUM-DC, so is $\Theta(n)$. The span T_∞ is given by longest sequence of steps that must be executed sequentially due to dependence (the critical path) in the computation graph. Here, this is the path starting with the initial invocation of MINIMUM-P, following down the stack of recursive invocations to a base case, and then returning back through the stack. Because the two recursive calls to MINIMUM-P are performed in parallel, we only need to consider the *longer* of them when calculating span. Since the array is divided exactly in half, it doesn't matter which we pick. Since, again, all the non-recursive steps take unit time, we end up with the recurrence

$$T_\infty(n) = T_\infty(n/2) + \Theta(1)$$

which is easily seen to solve to $T_\infty(n) = \Theta(\lg n)$.

The parallelism $\mathbb{P}$ is thus $\Theta(n/\lg n)$, which grows quickly with n. This means we can utilize quite large numbers of processors efficiently on problems of only modest size. For example, to find the minimum of a million elements, we could expect to approach perfect linear speedup using up to hundreds of processors.

Warning: we can never really achieve perfect speedup in reality, because of the overheads of thread coordination, scheduling, etc.

4 Parallelizing Merge Sort

Now let's address parallel sorting. Again, sequential divide-and-conquer methods are obvious candidates for parallelization. Let's try MERGE-SORT. As a first attempt, we can just try parallelizing the recursive calls in the usual definition, where MERGE is the standard (sequential) $\Theta(n)$-time algorithm (CLRS p. 31).

MERGE-SORT'(A, p, r)

```
1  if p < r
2      q = ⌊p + r/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

Since the sequentialization of this code is just ordinary MERGE-SORT, the work is the usual running time for that algorithm:

$$T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \lg n)$$

Since the two recursive calls run in parallel, the span is given by

$$T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n)$$

The resulting parallelism $\mathbb{P}$ is thus $\Theta(n \lg n/n) = \Theta(\lg n)$, which is pretty poor. It says we can only get perfect speedup when our problem size is exponential in the number of processors!

The problem here is with the span, which is dominated by the sequential MERGE step. Happily, with some cleverness, we can make this step parallel as well, by defining a new P-MERGE algorithm. As before, we will use divide and conquer, now just on the merge task itself. As usual, the challenge is to find a way to split the task into separable sub-parts, each a constant factor smaller, while doing as little work as possible on the splitting and combination stages.

For merging, the obvious approach is to split each of the (sorted) input arrays into two roughly equal parts around some value, merge the lower and upper parts from each half separately, and then combine the results. We have two challenges: getting a non-trivial split into sub-problems, and making the split and combination processes fast. Also, since we might not get the split exactly even (indeed, it won't even be close!), we can no longer assume that the inputs to the sub-problems have equal size.

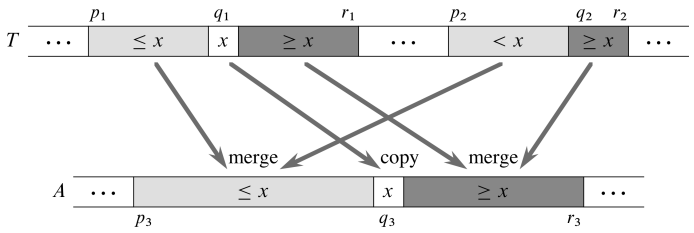
What we elect to do is to make the combination process trivial ($\Theta(1)$ time) by spending a bit more time doing the split (but still only $\Theta(\lg n)$). At each recursive invocation of P-MERGE we'll determine the ultimate output position of just *one* element, copy that element into position, and use it to split the input arrays into suitable sub-problems.

Here's the process:

- Find the median x at position q_1 of the first input array (in $O(1)$ time). Obviously this splits the first array nicely into equal halves.
- Find the position q_2 where x would live in the other (sorted) input array, using (sequential) binary search (in $O(\lg n)$ time). This might not split the array evenly at all, but that's ok, as we'll see.
- Calculate the position q_3 of x in the output array; this just requires a little arithmetic ($O(1)$ time).
- Copy x into position q_3 of the output array ($O(1)$ time).
- In parallel:

– Recursively merge the parts of the input arrays to the left of q_1, q_2 into the output array to the left of q_3.
– Recursively merge the parts of the input arrays to the right of q_1, q_2 into the output array to the right of q_3.

CLRS Fig. 27-6 illustrates:



All of this is set in a context where we work on slices of sub-arrays. The full details are in CLRS, p. 800. There are a few subtleties to handle base cases properly.

Note that each recursive invocation does very little "real work" (the $O(1)$ copy operation); most of the $O(\lg n)$ time needed is spent setting up for the recursive calls.

Let's analyze P-MERGE. The key question is how bad (i.e. uneven) the split into sub-problems might be. Let n_1 and n_2 be the sizes of the two input arrays, and assume wlog/ that $n_1 \geq n_2$ (if not, just swap the roles of the two arrays) and hence $n_2 \leq n/2$. We know that the first input array is split exactly in half, so it contributes $n_1/2$ elements to one sub-problem. In the worst case, one sub-problem combines those $n_1/2$ elements with all n_2 elements of the second input array. But then the size of that sub-problem is still at most $3n/4$ — a constant factor smaller than our original problem.

Armed with this, and the fact that the work per invocation exclusive of the recursive calls is dominated by the $\Theta(\lg n)$ binary search, we can calculate [how?] the worst-case span as:

$$T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$

Calculuting the work is harder. We first note a lower bound of $\Omega(n)$ since all n elements of the input arrays must be copied to the output array. We don't know exactly how big the sub-problems are, but they must be of size αn and $(1-\alpha)n$ for some α in the range $1/4 \leq \alpha \leq 3/4$. Thus, we have

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + O(\lg n)$$

where α is really a member of a family of constants that might be different at each level of the recursion. This is a nasty recurrence to solve by intuitive methods, but "guess and check" shows that it gives $T_1(n) = O(n)$, matching the lower bound.

So, P-MERGE is work-efficient (relative to the sequential algorithm) and has $\mathbb{P}$ of $\Theta(n/\lg^2 n)$.

Finally we plug P-MERGE into a version of MERGE-SORT suitably modified to generate its result in a separate output array. The resulting P-MERGE-SORT algorithm calls itself in parallel on the two halves of its input array, and then (after synchronizing) invokes P-MERGE.

The work and span at each stage are dominated by the call to P-MERGE, so we get

$$T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \log n)$$

$$T_\infty(n) = T_\infty(n/2) + \Theta(lg^2 n) = \Theta(\lg^3 n)$$

[where the last equality needs some justification]. Finally, the $\mathbb{P}$ for P-MERGE-SORT is $\Theta(n \log n)/\Theta(\lg^3 n) = \Theta(n/lg^2 n)$, which is much better than our previous $\lg n$ value.

5 Reduce and Scan

Suppose we want a parallel algorithm to sum the elements of an array. Does this resemble a problem we have already seen? After just a little thought, we can see that it is very similar to the MINIMUM problem. Just as for that problem, there is an obvious $O(n)$ sequential algorithm that processes the array sequentially from left-to-right. And, there is a straightforward parallelizable divide-and-conquer algorithm: use MINIMUM-P and simply change the way in which sub-problem results are combined:

```
SUM-P(A, p, r)
1  if p == r
2       return A[p]
3  q = ⌊(p + r)/2⌋
4  l = spawn SUM-P(A, p, q)
5  r = SUM-P(A, q + 1, r)
6  sync
7  return l + r
```

Again, we get $\Theta(n)$ work (so we are work-efficient) and $\Theta(\lg n)$ span, for excellent $\mathbb{P} = \Theta(n/\lg n)$

Generalizing some more, we can see that the sequential algorithms here can both be written as instances of a more general algorithmic pattern, parameterized by a combining operation $\oplus$:

```
REDUCE(A, p, r, ⊕)
1  s = A[p]
2  for i = p + 1 to r
3       s = s ⊕ A[i]
4  return s
```

Similarly, the divide-and-conquer algorithms can be written:

```
REDUCE-P(A, p, r, ⊕)
1  if p == r
2       return A[p]
3  q = ⌊(p + r)/2⌋
4  l = spawn REDUCE-P(A, p, q, ⊕)
5  r = REDUCE-P(A, q + 1, r, ⊕)
6  sync
7  return l ⊕ r
```

Suppose we can express an algorithm as an instance of REDUCE. Can we always switch to REDUCE-P to get a nice parallel version of the algorithm "for free"? No: there is a restriction on $\oplus$, namely that it must be

associative (i.e. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for all a, b, c). Otherwise, the tree-like order in which operations occur in REDUCE-P might lead to a different answer than sequential REDUCE. In other words, $\oplus$ and its underlying set of values form a *semigroup*.

(We can also be handy to extend the definitions of REDUCE and REDUCE-P to handle *empty* arrays. To do this, we pass an additional parameter I to describe what to return for the empty array case. To get a nice theory, we'll want I to be the *identity* for $\oplus$, so that we have a *monoid*.)

There are lots of interesting applications of REDUCE, but it is limited to producing values of the array element type. One common elaboration is to first *map* (apply a function to) each of the elements to a different type which has an appropriate monoid. We can readily define the map operation using a **parallel for** statement:

```
MAP(A, p, r, f, B)
1  parallel for i = p to r
2      B[i] = f(A[i])
```

Note that **parallel for** is not a primitive in our model of computation: it must be implemented in terms of **spawn** and **sync**. The most efficient way to do this is to build a balanced binary tree of **spawn** operations, which induces an extra $\Theta(\lg n)$ term in the span calculation for the parallel loop. Thus, if we assume that f has work T_1^f and span T_∞^f independent of the particular value it is applied to, then overall MAP has work $T_1(n) = nT_1^f$ and span $T_\infty(n) = T_1^f + \Theta(\lg n)$.

For many other applications, it is desirable to transform one array to another of the same length, accumulating information as we go. The paradigmatic example is *prefix sums*: Given an input array

$$x_1, x_2, \ldots, x_n$$

produce the result array

$$x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots, \sum_{i=1}^{n} x_i$$

Generalizing a bit in anticipation, we can give a sequential algorithm for this problem for arbitrary operators $\oplus$ and identity element I:

```
SCAN(A, p, r, ⊕, I, B)
1  s = I
2  for i = p to r
3      s = s ⊕ A[i]
4      B[i] = s
```

Rather remarkably, SCAN can also be given a generic, highly parallel implementation when $\oplus$ is associative. Here's the basic approach given in a somewhat abstract form. Think of the input array elements as being labels on the leaves of a complete binary tree taken from left to right; to make this work, we assume that array size n is a power of 2. The result array elements will also appear as labels on the leaves, again taken left to right. We will make two passes over the tree, the first bottom-up (the "up sweep") and the second top-down (the "down sweep"). On the up sweep, we will store an "up-label" value at each node of the tree, which will later be used by the down sweep. The up-labels of the leaves are the input array elements; the up-labels of internal nodes will be computed in a manner described below. On the down sweep, we will

compute and store a "down-label" value at each node. The down-labels on the leaves will be the elements of the result array. For technical convenience, we will actually compute the *prescan* of the array $x_1, x_2, \ldots, x_n$, which is defined to be

$$I, x_1, x_1 \oplus x_2, \ldots, (x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1})$$

(It should be clear that we can get the SCAN from this by a shift and one extra $\oplus$ operation.)

The whole point of this exercise is that the operations we perform on the tree are highly parallelizable, because there are no dependencies between adjacent subtrees—only between parents and children. Hence a straightforward translation to fork-join threads gives an algorithm with $\Theta(n)$ work and $\Theta(\lg n)$ span.

Once the basic idea is understood, it should be straightforward, if somewhat tedious, to make the algorithm more concrete (by figuring out how to store the node labels in arrays, handle arbitrary array sizes, compute the actual SCAN instead of the prescan, etc.), add threading constructs, and prove that the parallelized algorithm computes the same result array as sequential SCAN. The homework asks you to do this.

In fact, SCAN is so useful that it is often provided as a (parallel) hardware primitive, e.g. in GPU's.

Meanwhile, here are the details of the tree-based algorithm. Assume that the tree levels are labeled from 1 (for the root) to l (for the leaves).

TREE-SCAN

1 initialize up-labels of tree leaves to the input array elements
2 UP-SWEEP
3 DOWN-SWEEP
4 result array elements are in down-labels of tree leaves

UP-SWEEP

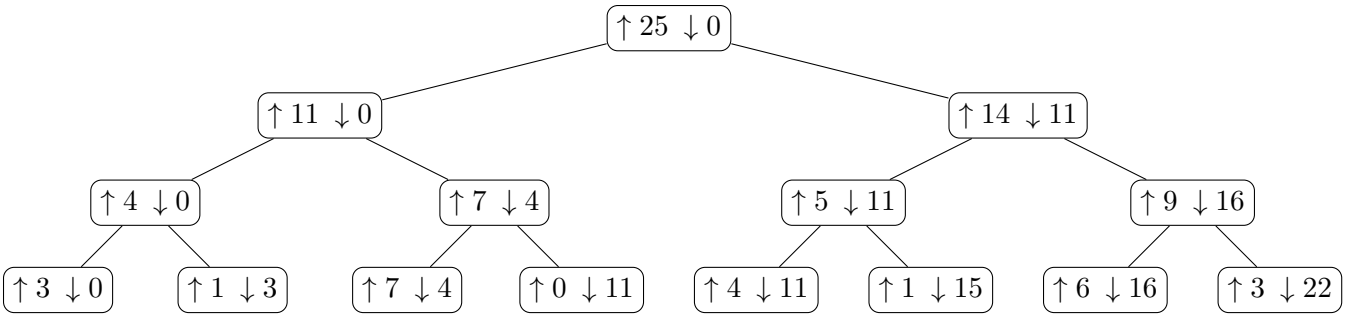
1 **for** each tree level from $l - 1$ up to 1:
2 **for** each node at this level
3 compute $v = v_l \oplus v_r$ where v_l and v_r are the up-labels of this node's left and right children
4 store v as this node's up-label

It should be clear that after UP-SWEEP completes, the up-label at each node contains the $\oplus$-sum of all the leaf up-label values in the subtree rooted at that node.

DOWN-SWEEP

1 store I as the root's down-label
2 **for** each tree level from 1 down to $l - 1$:
3 **for** each node at this level, having down-label w
4 store w as the down-label of this node's left child
5 compute $w' = w \oplus v_l$, where v_l is the up-label of this node's left child
6 store w' as the down-label of this node's right child

Here's an example for the input array $3, 1, 7, 0, 4, 1, 6, 3$ with $\oplus$ taken as ordinary addition and $I = 0$. The result of the prescan is $0, 3, 4, 11, 11, 15, 16, 22$.



The key fact that makes this algorithm work is that after DOWN-SWEEP completes, the down-label at each node contains the $\oplus$-sum of the up-label values for all the leaves that precede it in a pre-order traversal. (Recall that a pre-order traversal is a depth-first, left-to-right, traversal that visits first the node, then its children.) Thus in particular, the ith element of the output array, which is just the down-label of the leaf corresponding to position i, contains $I \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{i-1}$.

It is not hard to prove this by induction on the depth of the node. This figure from Blelloch, p. 43 helps explain the proof.

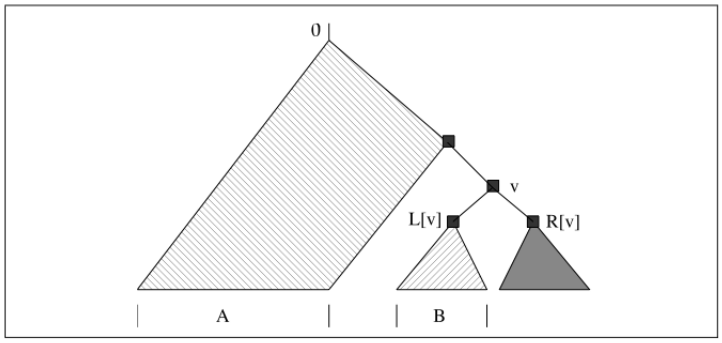


FIGURE 1.5

- Depth 0: Root. The root has no elements preceding it, so its down-label is correctly I.
- Depth $n > 0$: Any node at depth n must have a parent at depth $n - 1$ which we assume, as inductive hypothesis, has a correct down-label. (For example, let this be v in the figure above.) So it suffices to show that both children of a correctly labeled parent are also correctly labeled.

  The left child has exactly the same set of leaves preceding it does as the parent (A in the figure), because the preorder traversal always visits the left child of a node immediately after the node. So the inductive assumption that the parent's down-label is correct implies that the left child's down-label is also correct.

  The right child has two sets of leaves preceding it: those that precede the parent (A in the figure) and those in the subtree rooted at the left child (B in the figure). Thus combining the parent's down-label (assumed correct by induction) with the left child's up-label (characterized by an earlier remark) produces the correct down-label for the right child.

The SCAN mechanism is quite powerful indeed: it can be used to solve a wide variety of problems, some rather non-obvious. For example, we can use scans to move information down the array. Suppose we want to take an input array of integers and produce an output array of the same length, where each element contains

the previous positive value. For example, for input $0, 7, 0, 0, 3, 0$, the output should be $0, 0, 7, 7, 7, 3$. We can do this by scanning with a combining function $\triangleright$ (pronounced "copy")

$$a \triangleright b = \text{if } b > 0 \text{ then } b \text{ else } a$$

To use this combining function in a *parallel* scan, we must prove that $\triangleright$ is associative, i.e. $(a \triangleright b) \triangleright c = a \triangleright (b \triangleright c)$. This is tedious but straightforward to prove by considering all eight cases corresponding to whether each of a, b, c is positive or not.