Introduction to Parallel Algorithms

ECE 1762 Algorithms and Data Structures
Fall Semester, 2011

1 Preliminaries

Since the early 1990s, there has been a significant research activity in efficient parallel algorithms and novel computer architectures for problems that have been already solved sequentially (sorting, maximum flow, searching, etc). In this handout, we are interested in parallel algorithms and we avoid particular hardware details. The primary architectural model for our algorithms is a simplified machine called *Parallel RAM* (or PRAM). In essence, the PRAM model consists of a number p of processors that can read and/or write on a shared "global" memory in parallel (i.e., at the same time). The processors can also perform various arithmetic and logical operations in parallel.

The PRAM model was proposed to simplify architectural details of parallelism in the 1980s. Multiprocessor-based computers have been around for decades and various types of computer architectures [2] have been implemented in hardware throughout the years with different types of advantages/performance gains depending on the application. Nevertheless, there has not been any single model of parallelism that has been widely adopted by vendors and the scientific community. Today a trend of parallelism has been established through the use of *multi-core* processors and *dynamic multithreading*. Unlike static threading, where the operating system gives an illusion to the programmer of the existence of many "virtual processors", dynamic multithreading allows programmers for multi-core computers to specify a certain level degree of parallelism in applications withour worrying about communication protocols, load balancing and processor sharing [4]. In this handout, we are interested in a different paradigm for parallel programming. Whereas modern techniques execute programming tasks (loops, nested statements) in a concurrent manner, here we are interested to learn how to break a task into sub-tasks that can be executed in parallel. Not surprisingly, the PRAM model we use for this demonstration, it has a lot of similarities with the multi-core architecture in modern computers.

Depending on the underlying architecture (as we shall see, it has a significant impact on the performance and flexibility of the algorithm) a PRAM machine can allow or do not allow concurrent reads/writes on the shared memory cells. As such, a *concurrent-read* algorithm is a PRAM algorithm during whose execution multiple processors can read from the same location of shared memory at the same time. An *exclusive-read* algorithm is a PRAM algorithm in which no two processors ever read the same memory location at the same time. We can make similar distinction with respect to whether or not multiple processors can write into the same memory location at the same time generating new classes of PRAM algorithms: *concurrent-write* and *exclusive-write*. With all these in mind, the type of parallel algorithms that we will encounter are classified as follows:

- *EREW:* exclusive-read and exclusive-write,
- *CREW:* concurrent-read and exclusive-write,
- *ERCW:* exclusive-read and concurrent-write, and
- *CRCW:* concurrent-read and concurrent-write.

Of these types of algorithm models, the two extremes (EREW and CRCW) also happen to be the more popular. In this handout we will consider some simple algorithms running on this shared PRAM memory model. We will also discuss interesting techniques we should keep in mind when designing parallel algorithms. Throughout our discussion, we assume that all processors execute the same lines of code on, possibly, different data (SIMD machine). Instructions are executed by all processors *synchronously*.

1.1 Terminology

Fix some PRAM algorithm. Throughout our presentation, we use the following terminology:

- The *total time* (total number of parallel steps) is denoted with $T(n)$ and it is a function of the input size n.
- The *number of processors* is denoted with $P(n)$, also dependent on the input size.
- The *cost* of the computation is $\textsc{Cost}(n) = P(n) \times T(n)$.
- The *work* performed by the algorithm, $\textsc{Work}(n)$, is the total number of operations actually performed from all processors during all parallel steps.

Note that $\textsc{Cost}(n) \geq \textsc{Work}(n)$, since $P(n)$ processors working for $T(n)$ time can only perform at most $P(n) \times T(n)$ operations.

1.2 The Complexity of PRAM algorithms

Fix some PRAM algorithm. Then the following statements are equivalent.

1. The algorithm requires $T(n)$ time with $P(n)$ processors.
2. The algorithm has cost $\textsc{Cost}(n)$ and time $T(n)$.
3. The algorithm uses time $O(\frac{\textsc{Cost}(n)}{p})$ for any number of processors $p \leq P(n)$.
4. The algorithm uses time $O(\frac{\textsc{Cost}(n)}{p} + T(n))$ for any number of processors p.

Proof:

(1 ⇒ 2) Trivial since the cost $\textsc{Cost}(n)$ is equal to $T(n) \cdot P(n)$ by definition.

(2 ⇒ 3) We use the p processors to simulate the $P(n)$ processors, p at a time. Every step of the original algorithm now takes $\lceil P(n)/p \rceil$ time.

(3 ⇒ 4) When $p \leq P(n)$ we have time $O(\frac{\textsc{Cost}(n)}{p})$. But when $p \geq P(n)$, we can't do any better than $T(n)$ time (since we haven't specified how to use the additional processors that are available). So the time is $O(\max(\frac{\textsc{Cost}(n)}{p}, T(n))) = O(\frac{\textsc{Cost}(n)}{p} + T(n))$.

(4 ⇒ 1) Take $p = P(n)$ in (4).

1.3 Optimality of a PRAM Algorithm

Fix a problem and let $T_{seq}(n)$ and $\textsc{Work}_{seq}(n)$ be the time and work for an optimal sequential algorithm for this problem, respectively. We obviously have $T_{seq}(n) = \textsc{Work}_{seq}(n)$ and $\textsc{Work}_{seq}(n) \leq \textsc{Work}(n)$, for any parallel algorithm that solves the problem. We say a parallel algorithm is *optimal* for this problem if

$$\textsc{Cost}(n) = \Theta(T_{seq}(n))$$

Equivalently, we say that the parallel algorithm has *optimal speedup* if

$$P(n) = O(\frac{T_{seq}(n)}{T(n)})$$

Let optimal parallel algorithm A that uses $P(n)$ processors. Then, by the discussion above, we can construct a new algorithm A' that can be executed in time $T_p(n) = \textsc{Cost}(n)/p$ with $p \leq P(n)$ processors. The *speedup* — the ratio of sequential time to parallel time using p processors — of this new algorithm is $O(\frac{T_{seq}(n)}{T_p(n)}) = p$ (why?) and this is also an optimal speedup. Equivalently, A' is also optimal.

An algorithm is *strongly optimal* if it is optimal, and its time $T(n)$ is minimum for all parallel algorithms solving the same problem. For example, assume we have a problem that needs $\textsc{Work}_{seq}(n) = O(n)$ for an optimal single processor algorithm. If X and Y are two parallel algorithms for this problem and X runs in $O(\log n)$ time with $O(n/\log n)$ processors while Y runs in $O(1)$ time with $O(n)$ processors, then both X and Y are optimal, however, Y is strongly optimal.

1.4 The Complexity Class NC

The complexity class NC plays the same role in parallel computation that P —defined later in this class— plays in sequential computation. A problem is, at least in theory, considered to be *efficiently parallelizable* if it can be shown to belong in NC. NC stands for *Nick's Class* from Nick Pippenger who invented it.

Formally, NC is the collection (or class) of problems that can be solved on some PRAM machine in $(\log n)^{O(1)}$ or polylogarithmic time using $n^{O(1)}$ or polynomially many processors.

We should emphasize that the question $NC \stackrel{?}{=} P$ is analogous to the $P \stackrel{?}{=} NP$ question that we will consider during our presentation of NP–completeness.

2 Brent's Scheduling Principle (Brent's Theorem)

Assume that we are given a PRAM algorithm doing $\textsc{Work}(n)$ *work in* $T(n)$ *time for some number of processors. Assume that we have p processors available. If, for each parallel step i of the algorithm,*

1. *we can compute in constant time the number of operations performed at step i, and*
2. *we can allocate the available processors to these tasks in constant time,*

then we can run the algorithm with p processors in $O(T(n) + \frac{\textsc{Work}(n)}{p})$ *time.*

Proof. Let $\textsc{Work}_i(n)$ be the number of operations of the algorithm at each parallel step i. We can compute $\textsc{Work}_i(n)$ and allocate the p processors to do these operations in $1 + \frac{Work_i(n)}{p}$ time. So the total time required by the modified algorithm, using p processors, is

$$\sum_{i=1}^{T(n)} \left(1 + \frac{\textsc{Work}_i(n)}{p}\right) = T(n) + \frac{1}{p}\sum_{i=1}^{T(n)} \textsc{Work}_i(n) = T(n) + \frac{\textsc{Work}(n)}{p}$$

as stated.

Informally, Brent's Theorem says that whenever conditions (1) and (2) of the theorem are met, we can design an algorithm in the following way. Use as many processors as you want to find an algorithm that runs in time $T(n)$ and performs total work $\textsc{Work}(n)$. Recall that for any parallel algorithm $Work(n) \leq Cost(n)$. Now by taking $p = \textsc{Work}(n)/T(n)$, Brent's Theorem says that the algorithm can be modified to run using p processors in time

$$T_p(n) = \frac{\textsc{Work}(n)}{p} + T(n) = T(n) + T(n) = O(T(n)) \ .$$

In other words, we get an algorithm with time $T(n)$ and processors $P(n) = \textsc{Work}(n)/T(n)$. Note that $\textsc{Work}(n) = P(n)T(n) = \textsc{Cost}(n)$, that is, the new algorithm does not underutilize the processors.

2.1 Brent's Theorem: Optimal Prefix Sums in Arrays

This example illustrates Brent's Theorem with an optimal algorithm for prefix sums in an array, not in linked lists, as we discussed before. Since it takes $O(n)$ time to do it with a single processor, here we present how this theorem helps us develop an $O(logn)$ time algorithm with $O(n/logn)$ processors.

Let A be an array of n integers stored in shared memory locations $T[n], \ldots, T[2n-1]$ and let n processors $P_1, \ldots, P_n$. For simplicity assume that n is a power of 2. We want to develop an optimal parallel algorithm to compute the sum of the numbers in A. A single processor algorithm does the job in $O(n)$ time (work). Therefore, for the optimal parallel algorithm we should have $\textsc{Cost}(n) = T(n)P(n) = O(n)$ according to the discussion in subsection 1.3. We will develop such an algorithm for the CREW PRAM machine.

We build a complete binary tree, with all n integers being the leaves, using an array T of size $2n-1$. Every location in the array represents a node of the tree: $T[1]$ is the root, with children at $T[2]$ and $T[3]$. For any other node $T[i]$, its children are at $T[2i]$ and $T[2i+1]$. Observe that the leaves of the tree T correspond to locations $T[n], \ldots, T[2n-1]$.

Now each processor P_i executes the following program synchronously:

```
— P_i copies the ith array elements into the ith leaf of the tree.
T[(n − 1) + i] ← A[i]

— Now fill in values at the internal nodes, one level at a time.
for h = 1 to log_2 n do
    — At the hth step, only the first n/2^h processors are active.
    if i ≤ n/2^h then
        — Set the value at an internal node to the sum of its children.
        T[i] ← T[2i − 1] + T[2i]

— The sum is now in T[1], the root of the tree.
```

With $P(n) = n$ processors, this algorithm does exactly $\log n$ parallel steps, therefore, $T(n) = O(\log n)$. The total cost of the algorithm is

$$\textsc{Cost}(n) = P(n) \cdot T(n) = O(n \log n)$$

but the total number of operations (total work) is

$$\begin{aligned} \textsc{Work}(n) &= \sum_{h=1}^{\log n} \frac{n}{2^h} \\ &= 2n - 1 \\ &= O(n) \end{aligned}$$

since there are only $n/2^h$ processors active at every step $h = 1, \ldots, \log n$. Since the cost is more than the work done by a single processor machine, the algorithm is not optimal.

By Brent's Scheduling Principle, the algorithm can be modified to run with only $O(n/\log n)$ processors in $O(n/(n/\log n)+T(n)) = O(\log n)$ time. This new algorithm is optimal since $\textsc{Cost}(n) = O(n/\log n)O(\log n) = O(n) =$ (work of sequential algorithm). It can be shown [1] that this algorithm is also strongly optimal for the CREW PRAM.

As a final step, we need to show that both conditions of Brent's Theorem are satisfied. This is true, because (1) We know that there are exactly $n/2^h$ operations at each step h. (2) We know exactly which array

positions in T are affected in each step (which level of the binary tree). These are just contiguous elements in an array. So given p processors, we process these array locations in groups of p.

In this case, we do not really need to go to Brent's Theorem to solve the problem of finding an optimal algorithm. Instead we could do it like this. Suppose we have only $n/\log n$ processors. Now partition the original array A into $n/\log n$ blocks of size $\log n$. Assign one processor to each block. Each processor can sum up the elements in its block sequentially in time $O(\log n)$. This leaves us with $n/\log n$ partial sums, and $n/\log n$ processors. Now we can use the non–optimal binary tree based algorithm and find the sum of all numbers in

$$O(\log(n/\log n)) = O(\log n - \log\log n) = O(\log n)$$

time. What we really did is to replace the first $\log\log n$ levels of the tree with a sequential computation [1]. This algorithm runs in $O(\log n)$ total time using $n/\log n$ processors.

3 Pointer Jumping

3.1 List Ranking

Among the more interesting, yet simple, PRAM algorithms are those that involve pointers. Our first algorithm operates on linked-lists. Suppose that we are given a linked list L with n objects and wish to compute for each object in L its distance from the end of the list. More formally, if *next* is a pointer field, we want to compute a value $d[i]$ for each object i such that $d[i] = d[next[i]] + 1$ if $next[i] \neq NIL$, and $d[i] = 0$ if $next[i] = NIL$. We call this the *list-ranking* problem.

In the following discussion, we assume that there are n processors allocated to the problem, that is, one for each node in the linked list. A parallel EREW solution requiring only $O(logn)$ time that uses the technique of *pointer-doubling* is given by the following pseudo-code:

```
    — initialize
for each processor i in parallel do
    if next[i] = NIL then
        d[i] = 0
    else
        d[i] = 1
while there exists a node i where next[i] ≠ NIL do
    for each processor i in parallel do
        if next[i] ≠ NIL then
            d[i] = d[i] + d[next[i]]
            next[i] = next[next[i]]
```

Figure 1 shows the operation of the algorithm to compute the distances. Each part of the figure shows the state of the list before an iteration of the while-loop. Part (a) of the figure shows the list after initialization and before the while-loop. The result after the first iteration of the while-loop is shown in Figure 1(b). It can be also shown in Figure 1(b) (Figure 1(c)), that in the second (third) iteration of the loop, there are only four (two) objects with non-NIL pointers. The final distances are shown in Figure 1(d).

Clearly, this EREW algorithm takes $O(logn)$ time to terminate. Since n processors are used, the total amount of work required comes to be $O(nlogn)$. Since a single processor can calculate the result in $O(n)$ time by simply traversing the list twice, the presented algorithm is sub-optimal. Although this algorithm is

[1] We will formalize this idea in subsection 4.5.

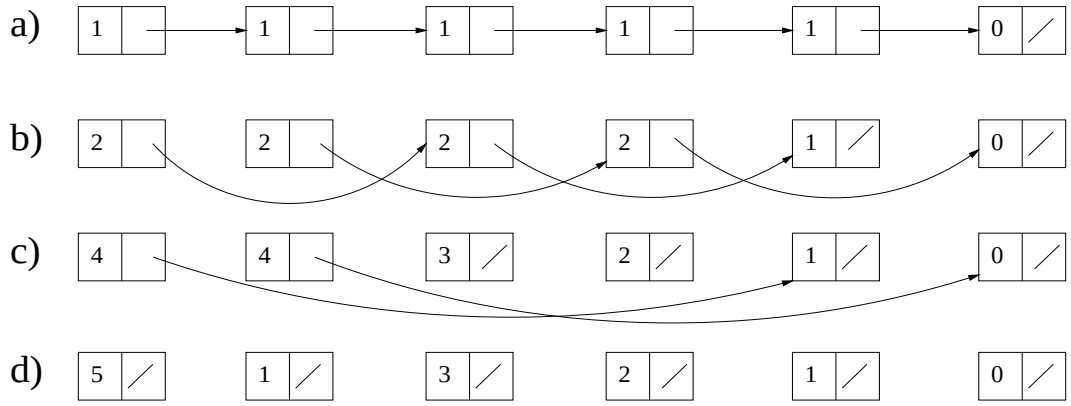

Figure 1: List Ranking Example

presented due to its simplicity, we should note that there exist optimal parallel algorithms for this problem that run in $O(logn)$ time with $O(n/logn)$ processors but we do not examine them here.

3.2 Parallel Prefix Computation

What if the numbers contained in the nodes are not single-unit distances? The technique of pointer jumping extends well beyond the application of list ranking. A *prefix computation* is defined in terms of a binary associative operator $\otimes$. The computation takes as input a sequence $< x_n, x_{n-1}, \ldots, x_2, x_1 >$ and produces an output sequence $< y_n, y_{n-1}, \ldots, y_2, y_1 >$ such that $y_1 = x_1$ and

$$y_k \;=\; y_{k-1} \otimes x_k \;=\; x_1 \otimes x_2 \otimes \ldots \otimes x_k$$

In other words, each y_k is obtained by using the operation $\otimes$ in the first k elements of the sequence, hence the term "prefix".

The algorithm to compute a prefix computation is identical to the one we presented for the list ranking problem. We simply replace the line $d[i] = d[i] + d[next[i]]$ with $d[i] = d[i] \otimes d[next[i]]$. As an example of applying this algorithm, Figure 2 shows the result of *prefix sums*, that is, the operator $\otimes$ is a simple addition. Prefix sums are important because in arithmetic hardware circuits, a "prefix" computation can be used to perform addition faster when using a carry-lookahead adder. Nevertheless, the details of this implementation [3] extend beyond the context of the class.

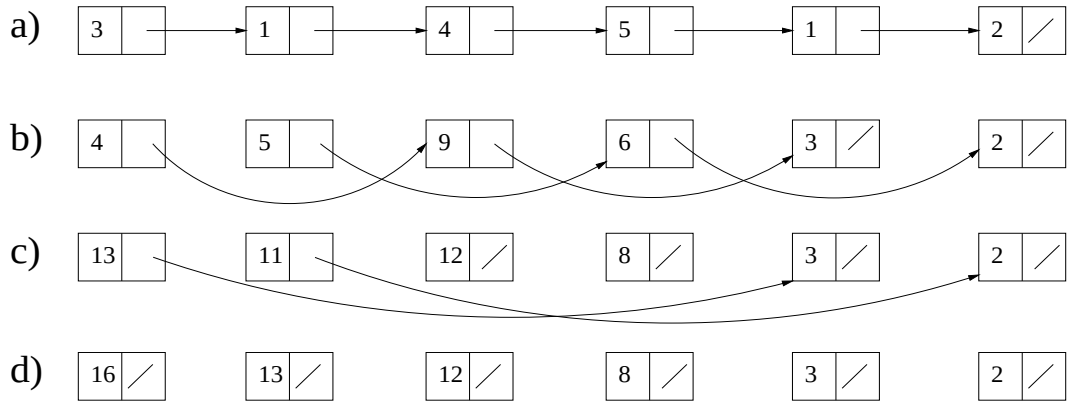

Figure 2: Prefix Sums Example

4 Finding the Maximum

In the remaining of this handout, we will consider the problem of designing an efficient parallel algorithm that returns the maximum of n numbers stored in an array A. A single processor can return an answer in $O(n)$ time. As we explained earlier, this also gives a tight bound for the total cost of the optimal parallel algorithm.

We will first present and analyze some simple algorithms to solve the problem. The section will conclude with the presentation of a strongly optimal deterministic parallel algorithm and illustrate a powerful technique used in the design of parallel algorithms, accelerating cascades. We will also have the opportunity to investigate some of the differences of various PRAM models.

4.1 A Simple Optimal Algorithm

Suppose that we want to develop an algorithm for finding the maximum of n elements that runs on a EREW PRAM. A simple optimal algorithm proceeds as the one described in subsection 2.1 using a *tournament tree*. Again, we base the structure of the algorithm on a complete binary tree: at each step we pair the remaining elements up, compare them, and discard the smaller of the two. This reduces the number of elements by a factor of $\frac{1}{2}$ at each step, so it takes about $\log n$ steps. Can we do better?

Unfortunately, on the EREW PRAM, we can't. A sketch of proof is as follows. Consider any EREW algorithm that finds the max of n elements. At each step, some portion of the elements know that they are not the max. The others are still candidates. Since the reads done at each step are distinct, at most a constant fraction of the elements can learn that they are not the max at any step. So there must be $\Omega(\log n)$ steps with a reasoning very similar to the one we used in our discussion for the depth of the recurrence tree of quicksort. In fact, Cook, Dwork and Reischuk proved that finding the max element still requires $\Theta(\log n)$ time on a CREW PRAM as well. With concurrent writing, the situation, as we shall see, is better.

4.2 Finding the Max in $O(1)$ time

Recall that the common CRCW PRAM allows several processors to write to the same memory location, as long as they all write the same value.

Theorem 1 *We can find the max of n elements in $O(1)$ time on a common CRCW PRAM with n^2 processors.*

To show this, assume that we have $n^2/2$ processors, and let m be an auxiliary boolean array of size n.

Step 1. For $i = 1, \ldots, m$, set $m[i] :=$ true.

Step 2. For all i and j in parallel, where $1 \leq i < j \leq n$, if $A[i] < A[j]$, set $m[i]:=$ false.

Now $m[i]$ is true exactly when $A[i] \leq A[j]$ for all j, *i.e.* when $A[i]$ holds a copy of the max value.

Step 3. For $i = 1, \ldots, n$, if $m[i] =$ true then set $max = A[i]$.

Even if several locations hold a copy of the same value, all values written to *max* will be the same.

This algorithm runs on a CRCW PRAM with $n^2/2$ processors in $O(1)$ time. However, this algorithm is not optimal (since the problem can be solved sequentially in $O(n)$ time). Valiant proved that any n-processor CRCW PRAM algorithm requires $\Omega(\log \log n)$ time to find the max of n elements. In fact, as we will see, $O(\log \log n)$ time is sufficient as well.

Notice that this discussion also shows that the CRCW model is strictly more powerful than the EREW and CREW models.

4.3 Partitioning

Theorem 2 *We can find the maximum of n elements in $O(1/\epsilon)$ time on a common CRCW PRAM with $O(n^{1+\epsilon})$ processors, for any $0 < \epsilon \leq 1$.*

This just means that we can find the max in constant time with $O(n^k)$ processors, as long as $1 < k \leq 2$. The amount of time will depend on the choice of k.

The idea is to partition the input into p parts of sufficiently small size so that we have enough processors to apply Theorem 1 to all parts simultaneously. This reduces the problem to one of size p. We repeat the basic step until there is just one element left.

First we'll just assume that $\epsilon = \frac{1}{2}$, so we have $n^{\frac{3}{2}} = n\sqrt{(}n)$ processors.

Step 1. Divide the array into $\sqrt{n}$ blocks, each of size $\sqrt{n}$.

Step 2. Now we simultaneously apply Theorem 1 to each part. Since we need only $(\sqrt{n})^2 = n$ processors for each of the parts, we can do this in $O(1)$ time.

Step 3. This leaves us with $\sqrt{n}$ elements, and the max element is among them. We again apply Theorem 1 to these elements to find the maximum.

With $n^{1+\epsilon}$ processors, the idea is similar. Actually, it suffices to show that this works for $\epsilon = 1/c$, for all integers c (why?), and we will use induction to prove it.

The algorithm above is the base of the induction as it works for $c = 2$. For the induction hypothesis, assume we have a constant time algorithm for $c - 1$. We will show how to construct a constant time algorithm for c. Now if we have only $n^{1+1/c}$ processors, we partition the problem into blocks of size $n^{1/c}$. We can apply the constant time algorithm to each block, reducing the problem to one of size $N = n^{1-1/c}$. (N is now the number of elements remaining, and the max is one of these. $n = N^{\frac{c}{c-1}}$.) The number of processors available is

$$n^{1+1/c} = N^{\frac{c+1}{c-1}} = N^{1+2/(c-1)} \ .$$

By assumption, we already have an algorithm that will find the max of N elements using $N^{1+1/(c-1)}$ processors, so we're done. Unrolling the induction, we can see that the number of steps in the algorithm using $N^{1+1/c}$ processors is $O(c)$.

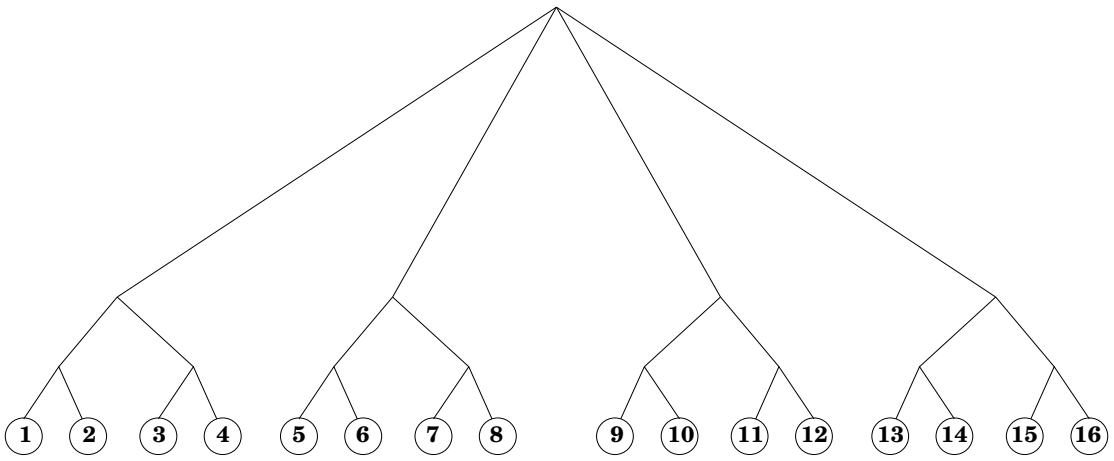

Figure 3: A doubly logarithmic–depth tree on 16 nodes

4.4 Valiant's $O(\log \log n)$ time algorithm

By extending the ideas presented earlier we can do even better.

Theorem 3 *We can find the max of n elements in $O(\log \log n)$ time with $O(n)$ processors on a common CRCW PRAM.*

The algorithm proceeds as above. At each step we partition the elements into small enough groups to apply the basic step of Theorem 1 simultaneously to all of the groups.

For the first step (call this step 0), we have n elements, and partition the elements into groups of 2. Finding the max of each group leaves $n/2$ elements. Next, for step 1, we partition the elements into $n/8$ groups of 4. To apply Theorem 1, each group needs $4^2/2 = 8$ processors, so all can be done simultaneously with the n processors available. This leaves $n/8$ elements. Next, for step 2, we can partition into $n/128$ groups of 16 elements. Each group needs $16^2/2 = 128$ processors — just enough. Observe that the basic structure of the algorithm corresponds to a doubly-logarithmic depth tree. Such a tree, for 16 leaf nodes, is shown in Fig. 3.

Continuing in this way, we see that step k reduces the number of elements by a factor of 2^{2^k}. So it takes $O(\log \log n)$ steps before we get down to just one element, the maximum of the array. Unfortunately, if we count the amount of work (number of operations) performed by this algorithm, it turns out to be $O(n \log \log n)$. Thus, the algorithm is not optimal.

4.5 Accelerating cascades

We conclude our presentation with the presentation of a general technique that can be often used to improve the performance of a parallel algorithm. In our case, accelerating cascades produces a strongly optimal algorithm for the problem of finding the maximum.

The idea behind *accelerating cascades* is as follows: you have several algorithms to solve a problem, each with a different time/processor requirement. Typically, you have slower algorithms that are optimal, and faster algorithms that are non-optimal because they require too many processors. Each of these algorithms uses a sequence of steps to reduce the problem to a similar problem of smaller size. Start with the optimal but slower algorithm, and reduce the size of the problem. Then use a faster algorithm to reduce the size even more, and continue like this. In many cases, this will allow you to derive an algorithm which is both fast and closer to optimal.

For example, to find the max of elements we already described a couple of algorithms with different time/processor requirements:

1. Solve the problem optimally (sequentially) in $O(n)$ time with 1 processor.
2. Solve the problem optimally in $O(\log n)$ time with $O(n/\log n)$ processors using the balanced binary tree scheme (pairwise comparisons). Each step of this algorithm reduces the size of the problem by $\frac{1}{2}$.
3. Solve the problem in $O(\log \log n)$ time with $O(n)$ processors (Valiant's algorithm).

We will now combine these algorithms to get an optimal $O(\log \log n)$ time one that has only $O(n)$ cost (*i.e.* uses $O(n/\log \log n)$ processors).

Here are two ways that you can do it:

1. Use (2) and (3). First we apply $\log \log \log n$ steps of the binary tree algorithm (2). There is $O(n)$ work here and, since each step reduces the problem size by $\frac{1}{2}$, it reduces the problem to one of size

$N = n/2^{\log\log\log n} = n/\log\log n$. Now apply Valiant's algorithm (3) to these elements: this requires $O(\log\log N) = O(\log\log n)$ time and work

$$O(N\log\log N) = O\left(\frac{n}{\log\log n}\cdot\log\log\left(\frac{n}{\log n}\right)\right) = O(n)\ .$$

So the total time is $O(\log\log n)$ and work is $O(n)$. By Brent's Theorem, this can be done with $O(n/\log\log n)$ processors.

2. Similarly, we can use (1) and (3) and $n/\log\log n$ processors. Partition the array into $n/\log\log n$ blocks of size $\log\log n$ each. Assign one processor to each block and (using (1)) each processor computes the max of its block of elements. This reduces the problem to one of size $n/\log\log n$ in $\log\log n$ steps. Now apply Valiant's algorithm to these elements and find the max in $O(\log\log n)$ steps using only $n/\log\log n$ processors.

It can be shown [1] that any n-processor CRCW PRAM algorithm that uses only comparisons on elements of the array requires $\Omega(\log\log n)$ time to find the max of n elements. Therefore, both algorithms described above are strongly optimal comparison based parallel algorithms for this problem.

5 Further Reading

There is a rich literature on parallel machine models, parallel architectures and parallel algorithms. The text by [2] is a good start as it contains a comprehensive description of algorithms and different architecture topologies for the network model (tree, hypercube, mesh, and butterfly). The literature in [1] and [3] discuss in depth recent results and algorithms for the shared memory PRAM model.

In the world of new microprocessing chips with multiple cores and multi-threading[4], parallel algorithms gain real time acceptance.

References

[1] *Jájá J.*, "An Introduction to Parallel Algorithms," *Addison–Wesley,* 1992.

[2] *Leighton T.*, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes," *Morgan–Kaufmann,* 1991.

[3] *T. Cormen, C. Leiserson and R. Rivest*, "Introduction to Algorithms," *McGraw-Hill* (1st Ed.), 1990.

[4] *T. Cormen, C. Leiserson, R. Rivest and Stein*, "Introduction to Algorithms," *MIT Press* (3rd Ed.), 2009.