Finding the max of n elements in parallel

Suppose n distinct numbers are stored in an array $A[1, \ldots, n]$. We describe a parallel algorithm for finding the max, due to Leslie Valiant. We assume a very idealized parallel computing model. In our model, there are n processors. In one parallel comparison step, every processor may read two numbers from memory, compare them, and write a value in memory. The challenge in designing a parallel algorithm is to utilize these processors in parallel so as to find the max in as few parallel comparison steps as possible.

The algorithm works roughly as follows. The array A is partitioned into $\sqrt{n}$ subarrays, each with $\sqrt{n}$ elements and the set of processors is partitioned into $\sqrt{n}$ subsets, each with $\sqrt{n}$ processors. The algorithm calls itself recursively with one subset of processors assigned to each subarray. There are $\sqrt{n}$ recursive calls, and all can be done in parallel. Let $m(i)$ be the maximum value in the ith subarray. The values

$m(i)$ are returned by the recursive calls, and one of these values is the true maximum of the whole array.

In the second phase of the algorithm, the processors assigned to the ith subarray determine if $m(i)$ is indeed the maximum of the whole array. Note that $m(i)$ is the maximum of the whole array if and only if $m(i) \geq m(j)$ for $1 \leq j \leq \sqrt{n}$. This is a total of $\sqrt{n}$ comparisons, and these are done in parallel by all processors assigned to the ith subarray.

To avoid floors and ceilings, it is convenient to describe the algorithm when $n > 2$ is a perfect square (so that $\sqrt{n}$ is an integer). Also, it is convenient to use $n = 2$ as the base case.

```
FIND-MAX(A, P), where A is an array of n numbers and P is a set of n
processors

   If n = 2 then compare the two elements in A and return the maximum
   Else {n > 2}

      Divide A into root(n) subarrays, each containing root(n) numbers
      Divide P into root(n) subsets, each containing root(n) processors

      IN PARALLEL for each subarray, recursively call FIND-MAX on that
      subarray with the corresponding subset of processors

      Let m(i) be the maximum number in the i_th subarray, 1<=i<=root(n)

      IN PARALLEL for the i_th subarray, determine if m(i) is the max of
      m(1), m(2), ..., m(root(n)) as follows: IN PARALLEL, the j_th processor
      in the i_th subset compares m(i) and m(j) and if m(i) < m(j) the fact
      that m(i) is not the maximum of the whole array is communicated to all
      other processors in the i_th subset of processors. If in the i_th
      subset of processors there is no communication, then m(i) is the
      maximum of the whole array and is returned by the algorithm.
```


Let $M(n)$ be the number of parallel comparison steps needed to find the maximum element in array A. We have the following recurrence for $M(n)$:

$\star$

$M(n) =$

The $M(\sqrt{n})$ term counts the number of parallel comparison steps needed in a single recursive call. Since all the recursive calls are done *in parallel* we don't need to multiply this term by $\sqrt{n}$. The additional $+1$ counts the single comparison done by each processor in the second phase of the algorithm. Again, since all processors are doing these comparisons in parallel, only one parallel comparison step is needed.

Again, we can solve this recurrence using the iteration method:

⋆

$M(n) =$

Note that the number of parallel comparison steps on an input of size 16 is 3 and on an input of size 256 is only 4.

Recurrences Table

The following table lists several recurrence relations and their corresponding closed form expressions.

	Recurrence	Closed Form Expression	
1.	$f(n) = f(n-1) + 3$	$f(n) = \Theta(n)$	("linear")
2.	$f(n) = f(n-1) + 2n$	$f(n) = \Theta(n^2)$	("quadratic")
3.	$f(n) = 2f(n/2) + n$	$f(n) = \Theta(n \log n)$	
4.	$f(n) = f(n/3) + 1$	$f(n) = \Theta(\log n)$	("logarithmic")
5.	$f(n) = f(n/2) + n$	$f(n) = \Theta(n)$	("linear")
6.	$f(n) = 2f(n-1) + 1$	$f(n) = 2^{\Theta(n)}$	("exponential")

Using Proof by Induction to Solve Recurrence Relations

Often when analyzing algorithms, the recurrence relations one needs to work are similar to those above, but not so "clean." For example, there may be floors or ceilings in the recurrences, or extra constants appearing inside the function on the right hand side, as in the next example. Applying the iteration method to such recurrences can be very messy. We describe an alternative way to handle such recurrences in this lecture.

The idea is very simple. Faced with a new recurrence,

1. Compare it to recurrences you already know (as in the recurrences table) and find one that looks like the new recurrence you are trying to solve.

2. Guess that the closed form expression to your new recurrence is exactly the same as for the corresponding simpler recurrence, ignoring constant factors. (This is why Θ notation comes in handy.)

3. Finally, use proof by induction to verify that your guess is correct.

$\star$

Example:

$$\begin{cases} f(n) < 1000 & n \leq 7 \\ f(n) < f(\lfloor n/3 \rfloor + 5) + 1 & n \geq 8 \end{cases}$$

$f(30) =$

This recurrence looks like row 4 of the table above. Therefore, we might guess that $f(n) = \Theta(\log n)$. In order to prove this, we need to show that $f(n) = O(\log n)$ and that $\log n = O(f(n))$. We will just do the first of these two tasks. The other can be done in a similar fashion (try it).

Going back to the definition of O-notation, we see that to show $f(n) = O(\log n)$ it is sufficient to show that $f(n) \leq c \log_2 n$ for some constant c and all $n > 1$.

Claim 4 *For all $n > 1$, $f(n) \leq c \log_2 n$.*

A proof by induction has a base case, an induction hypothesis and an induction step. The base case handles small values of n. The induction hypothesis states that the claim is true for all values of $i < n$, i.e. that for all $i, 1 < i < n$, $f(i) \leq c \log_2 i$. The induction step proves that the claim is also true for n, *given that* the claim is true for all $i, 1 < i < n$ from the induction hypothesis.

$\star$

Induction Step: We need to show that $f(n) \leq c \log_2 n$. We will start with what we know about $f(n)$, namely the definition of $f(n)$ given by the recurrence relation, apply the induction hypothesis and some other useful algebraic manipulations until we arrive at the conclusion we want, namely that $f(n) \leq c \log_2 n$.

$f(n) = f(\lfloor n/3 \rfloor + 5) + 1$

In going from the first line to the second line, we want to be able to apply the induction hypothesis. We can do this as long as $\lfloor n/3 \rfloor + 5 < n$, i.e. as long as $n \geq 8$. In going from the second to the third line, we are making our eventual task easier by replacing the rather messy quantity $\lfloor n/3 \rfloor + 5$ by the simpler quantity $n/2$. The expression in second line is $\leq$ the expression in the third line as long as $n \geq 30$. This tells us that we need the base case to handle values of $n < 30$. In going from line 4 to line 5, we learn that the constant c needs to be at least 1 in order to be able to complete our proof. We can now proceed with the base case of the proof.

$\star$

Base Case: For all $n, 1 < n < 30$, $f(n) \leq c \log_2 n$. We saw above that $f(30) < 1004$. Therefore, for all $n, 1 < n < 30$,

$f(n) \leq$

At this point we have all the information we need to write the complete proof, choosing $c = 1004$. Here it is:

Claim 5 *For all $n > 1$, $f(n) \leq 1004 \log_2 n$.*

Base Case: We show that for all $n, 1 < n < 30$, $f(n) \leq 1004 \log_2 n$. Note that $f(30) < 1004$. In fact, for all $n, 1 < n < 30$, $f(n) \leq 1004$. Therefore, for all $n, 1 < n < 30$, $f(n) \leq 1004 \log_2 n$ and we are done with the base case.

Induction Hypothesis: $f(i) \leq 1004 \log_2 i$ for all $1 < i < n$.

Induction Step: We need to show that if $n \geq 30$ then $f(n) \leq 1004 \log_2 n$. Note that

$$
\begin{aligned}
f(n) &= f(\lfloor n/3 \rfloor + 5) + 1 \\
&\leq 1004 \log_2(\lfloor n/3 \rfloor + 5) + 1 \text{ (by the induction hypothesis} \\
&\leq 1004 \log_2(n/2) + 1 \text{ (since for } n \geq 30, \lfloor n/3 \rfloor + 5 \leq n/ \\
&= 1004 \log_2 n - 1004 \log_2 2 + 1 \\
&\leq 1004 \log_2 n \text{ (assuming that } c \geq 1)
\end{aligned}
$$

Using the same method, we could in fact prove that if $f(n)$ is described by a recurrence of the form $f(n) = f(n/a) + \Theta(1)$ where $a > 1$ then $f(n) = \Theta(\log n)$. Many of the other recurrences in the table above can similarly be generalized, yielding the same closed form expression when expressed using Θ-notation.

Here is the generalized table. As an exercise, see if you can use proof by induction to explain the results summarized in this table.

	Recurrence	Closed Form Expression	
1.	$f(n) = f(n - \Theta(1)) + \Theta(1)$	$f(n) = \Theta(n)$	("linear")
2.	$f(n) = f(n - \Theta(1)) + \Theta(n)$	$f(n) = \Theta(n^2)$	("quadratic")
3.	$f(n) = af(n/a) + \Theta(n), a > 1$	$f(n) = \Theta(n \log n)$	
4.	$f(n) = f(n/a \pm \Theta(1)) + \Theta(1), a > 1$	$f(n) = \Theta(\log n)$	("logarithmic")
5.	$f(n) = f(n/a \pm \Theta(1)) + \Theta(n), a > 1$	$f(n) = \Theta(n)$	("linear")
6.	$f(n) = af(n - \Theta(1)) + \Theta(1), a > 1$	$f(n) = 2^{\Theta(n)}$	("exponential")

Row 5 of the original table can also be generalized in a different way. Namely, suppose $f(n)$ is described by a recurrence of the form:

$$f(n) = f(n/a) + f(n/b) + \Theta(n),$$

where a and b are positive constants. Then, if $1/a + 1/b < 1$, $f(n) = \Theta(n)$. Again a straightforward proof by induction argument can be used to establish this. On page 191 of the text, this is done for specific values of a and b that arise in a linear-time selection algorithm. See if you can follow the proof there.

Later as we find recurrences for algorithms studied in this course, we can refer back to this table to quickly figure out what is the closed form expression for the recurrence (ignoring constant factors). Another important point is that having a grasp of how the "form" of a recurrence for $f(n)$ determines the closed form expression for $f(n)$ can help not only in analyzing algorithms, but also in designing algorithms. For example, the linear-time selection algorithm that we will

see in a later lecture breaks an input of size n into two subproblems of sizes n/a and n/b, solves these recursively and then finds the solution for the input of size n with an additional $\Theta(n)$ operations. The designers of this algorithm understood that the constants a and b needed to be such that $1/a + 1/b < 1$ in order to achieve their goal of a linear-time algorithm. This knowledge helped them in the process of designing the algorithm.