Lecture 2 — Divide-and-Conquer and Recurrences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Kanat Tangwongsan — January 19, 2012

Material in this lecture: Today's lecture is about *divide and conquer*.

- Multiply n-bit numbers
- Solving recurrences using the tree method.

1 Divide and Conquer

Divide and conquer is a highly versatile technique that generally lends itself very well to parallel algorithms design. You probably have seen the divide-and-conquer technique many times before this class, but this is such an important technique that it is worth seeing it over and over again. It is particularly suited for "thinking parallel" because it offers a natural way of creating parallel tasks.

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving (e.g. recursive sorting in merge sort). This is not always the case. Often, you'll need more information from the subproblems to properly combine results of the subproblems. In this case, you'll need to strengthen the problem, much in the same way that you strengthen an inductive hypothesis when doing an inductive proof. We will go through some examples in this class where problem strengthening is necessary. But you have seen some such examples already from Recitation 1 and your Homework 1.

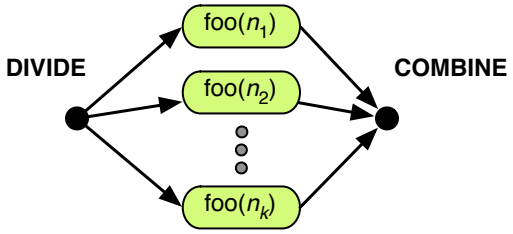
The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out cost bounds. The general structure looks as follows:

— **Base Case:** When the problem is sufficiently small, we return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:** First, the algorithm divides the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem. Then, it recurses on each of the parts to obtain answers for the parts. In proofs, this is where we assume inductively that the answers for these parts are correct, and based on this assumption, it combines the answers to produce an answer for the original instance I.

This process can be schematically depicted as

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.



On the assumption that the subproblems can be solved independently, the work and span of such an algorithm can be described as the following simple recurrences: If the problem of size n is broken into k subproblems of size $n_1, \ldots, n_k$, then the work is

$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n)$$

and the span is

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combine}}(n)$$

Note that the work recurrence is simply adding up the work across all components. More interesting is the span recurrence: First, note that a divide and conquer algorithm has to split a problem instance into subproblems *before* these subproblems are recursively solved. Furthermore, it cannot combine the results from these subproblems to generate the ultimate answer *until* the recursive calls on the subproblems are complete. This forms a chain of sequential dependencies, explaining why we add their span together. The parallel execution takes place among the recursive calls since we assume that the subproblems can be solved independently—this is why we take the max over the subproblems' span.

Applying this formula results in familiar recurrences such as $W(n) = 2W(n/2) + O(n)$. In the rest of this lecture, we'll get to see other recurrences. For how, let's derive a closed-form for this expression.

The first recurrence we're looking at is $W(n) = 2W(n/2) + O(n)$, which you probably have seen many times already. To derive a closed form for it, we'll review the tree method, which you have seen in 15-122 and 15-251.

But first, let's think through what it means when we write $O(f(n))$ in an expression (e.g., when we write $4W(n/2) + O(n)$ in the recurrence above). In these expression, we write $O(f(n))$ in place of *some* function $g(n) \in O(f(n))$. From the definition of $O(\cdot)$, this means that there exist positive constants N_0 and c such that for all $n \geq N_0$, we have $g(n) \leq c \cdot f(n)$. It follows that *there exist constants k_1 and k_2 such that for all $n \geq 1$,*

$$g(n) \leq k_1 \cdot f(n) + k_2,$$

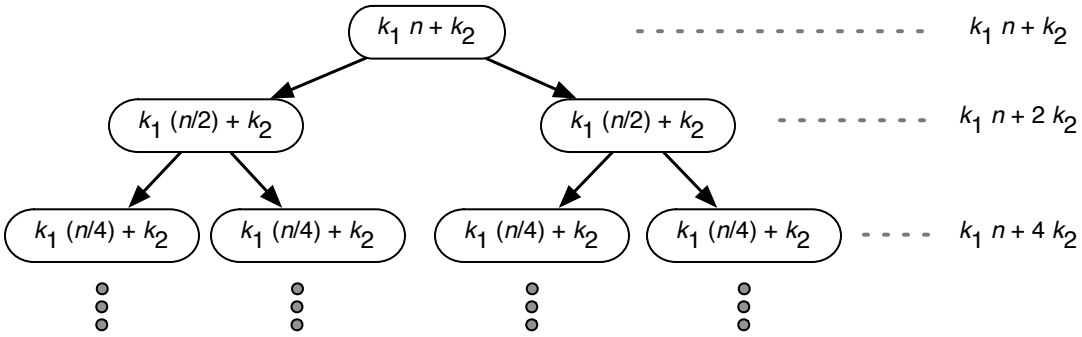
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} |g(i)|$.

By this argument, we can establish that

$$W(n) \leq 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. We'll now use the tree method.

The idea of the tree method is to consider the recursion tree of the algorithm in hope to derive an expression for what's happening at each level of the tree. In this particular example, we can see that each node in the tree has 2 children, whose input is half the size of that of the parent node. Moreover, if the input has size n, the recurrence shows that the work, excluding that of the recursive calls, is at most $k_1 \cdot n + k_2$. Therefore, our recursion tree annotated with cost looks like this:



To apply the tree method, there are some key questions we should ask ourselves to aid drawing out the recursion tree and to understand the cost associated with the nodes:

- How many levels are there in the tree?
- What is the problem size at level i?
- How much work is being done at each node in level i?
- How many nodes are there at level i?
- How much work is performed across the nodes in level i?

Our answers to these questions lead to the following analysis: We know that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

$$2^i \cdot \left(k_1 \frac{n}{2^i} + k_2\right) \quad = \quad k_1 \cdot n + 2^i \cdot k_2.$$

Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} \left(k_1 \cdot n + 2^i \cdot k_2\right) \\ &= k_1 n(1 + \log n) + k_2(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots + 1) \\ &\leq k_1 n(1 + \log n) + 2k_2 n \\ &\in O(n \log n), \end{aligned}$$

where in the second to last step, we apply the fact that for $a > 1$,

$$1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \leq a^{n+1}.$$

2 Example I: Multiply n-bit Numbers

Let's take a look at one of the most basic computing primitives. Suppose we want to multiply two n-bit numbers, say A and B. More formally, we have:

Problem 2.1 (The Multiplication Problem)**.** Given two n-bit numbers

$$A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle \text{ and}$$
$$B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle,$$

compute the product AB in binary.

For example, if A is 210 (11010010 in binary) and B is 251 (11111011 in binary), then the answer we're looking for is 52,710, or 1100110111100110 in binary.

How do we solve this problem? According to what it means to multiply, what we're after is the result of adding $A = 210$ to itself $B = 251$ times, or symmetrically, adding $B = 251$ to itself $A = 210$ times. This is not particular efficient. Say we can add two n-bit numbers in $O(n)$ work and $O(\log n)$ span. This naïve approach can need as many as $\Theta(2^n)$ additions, so:

Super Naïve Multiply: $\Theta(n2^n)$ work and $\Theta(2^n \log n)$ span.

That's exponential work and span. Very slow.

There is clearly a better way to do this. As we learn back in grade school, we can multiply numbers like this (though, they probably didn't teach you to multiply binary numbers):

```
          11111011
          11010010    x
-----------------------
         11111011
      11111011
    11111011          +
   11111011
-----------------------
 1100110111100110
=======================
```

This is what is known *long multiplication*. In this process, each digit of the multiplier is multiplied with the multiplicand—and the results are added up. Because there are n bits in the multiplier, this method generates n n-bit numbers for us to add, which requires $\Theta(n)$ additions. Now if we add them one by one sequentially[1], we have:

Grade School Multiply: $\Theta(n^2)$ work and $\Theta(n \log n)$ span.

[1] You might remember from 15-150 that there is an operation called `reduce`, which allows one to add up these numbers in parallel. Indeed, the same idea applies here, but since we're working with "big" numbers and the cost of adding up two numbers is superconstant, we'll need a more refined cost analysis, which we'll discuss in a few lectures.

We might ask what the parallelism is here. Remember that the parallelism measure is simply W/S, so

$$\text{Parallelism} \approx \frac{n^2}{n \log n} = n/\log n.$$

Is this the best way to multiply two numbers? For a long time, it was believed that multiplying two numbers requires $\Omega(n^2)$ work. In 1952, Andrey Kolmogorov even conjectured that this is essentially the best possible. As it turns out, we now know that we can multiply two numbers much faster than this. Today, we will reconstruct an algorithm due to Karatsuba that takes only $O(n^{1.585})$ work. Anatolii Karatsuba, then a 23-year-old student in Russia, invented the algorithm in 1960 although the paper wasn't published until 1962, and interestingly enough, he did not write this paper—Kolmogorov, possibly together with Yuri Ofman, wrote the paper under Karatsuba and Ofman's names.

2.1 First Attempt Using Divide-and-Conquer

Let's try divide and conquer. The first question to ask ourselves is, how to divide up the problem? The first thing that comes to mind might be the simplest split, dividing them each into their most-significant half and their least-significant half:

$$A = p2^{n/2} + q \qquad A = \boxed{p}\boxed{q}$$

$$B = r2^{n/2} + s \qquad B = \boxed{r}\boxed{s}$$

So then, the product $A \cdot B$ is simply

$$A \cdot B = pr \cdot 2^n + (ps + rq) \cdot 2^{n/2} + qs$$

That is, to compute $A \cdot B$, we need to compute pr, ps, rq, qs—that's a total of 4 multiplies. These multiplies are independent, so they can be done together. The size of these numbers are also only $n/2$. In addition to this, we will need 2 shift operations and 3 adds. For concreteness, we can write the following pseudocode:

```
1   fun mult(A,B) =
2     if |A| ≤ 1 then ⟨A_0 · B_0⟩
3     else let
4           val (p,q) = (A_{0..n/2}, A_{n/2+1..n-1})
5           val (r,s) = (B_{0..n/2}, B_{n/2+1..n-1})
6           val (pr,ps,rq,qs) = par(mult(p,r)||mult(p,s)||mult(r,q)||mult(q,s))
7           val sum = ps + rq
8         in
9           shift(pr,n) + shift(sum,n/2) + qs
10        end
```

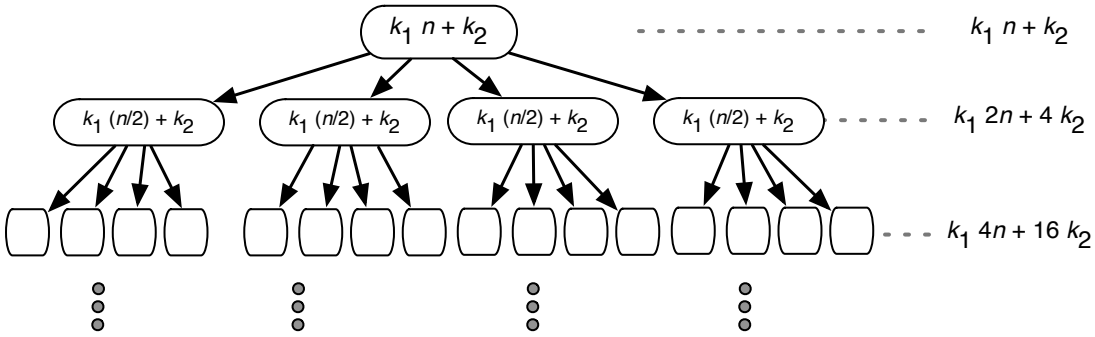
The recursion calls in this code all happen on Line 6. This gives us the following recurrences:

$$W(n) = 4W(n/2) + O(n)$$
$$S(n) = S(n/2) + O(\log n)$$

Solving these recurrences: Again, we'll use the tree method. Like before, this recurrence translates to

$$W(n) \quad \leq \quad 4W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. We can draw the recursion tree annotated with cost which looks like this:



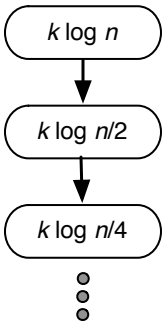
Let's use the convention that the root is level $i = 0$. We know that the problem size at level i is $n/2^i$, so each node costs at most $k_1(n/2^i) + k_2$. We also know that level i contains 4^i nodes. Thus, the total cost in level i is at most

$$4^i \cdot \left(k_1 \frac{n}{2^i} + k_2\right) \quad = \quad k_1 \cdot 2^i \cdot n + 4^i \cdot k_2.$$

Once again, since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, using a similar reasoning as before, we have

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} \left(k_1 \cdot 2^i \cdot n + 4^i \cdot k_2\right) \\ &\leq k_1 n 2^{1+\log_2 n} + k_2 4^{1+\log_2 n} \\ &= 2k_1 n \cdot \underbrace{2^{\log_2 n}}_{=n} + 4k_2 \underbrace{4^{\log_2 n}}_{=n^2} \\ &= (2k_1 + 4k_2)n^2 \\ &\in O(n^2) \end{aligned}$$

As for span, by unfolding the recurrence, we have the following recursion tree:



We conclude that the span is

$$S(n) \;=\; \sum_{i=0}^{\log n} k\log(n/2^i) \;=\; \sum_{i=0}^{\log n} k(\log n - i) \in O(\log^2 n).$$

So this algorithm is still an $O(n^2)$-work algorithm. We haven't really made any progress, have we? The span bound does get a bit better—we are down to $O(\log^2 n)$ span.

What other ideas might we try?

2.2 Second Attempt, Still Using Divide-and-Conquer

One idea is to reduce the number of subproblems that need be solved. But *how can we compute* pr, $(ps+qr)$*, and* qs *using less than* 4 *multiplies?* Remember that adding up numbers are cheap as long as we don't do too many of them.

> **Puzzle:** Given four numbers p,q,r,s, can you compute pr, qs, $(ps+qr)$ using 3 multiplies and at most 4 adds?

The solution to this puzzle is the critical step behind obtaining a divide-and-conquer algorithm with less work than before. For starters, we'll look at what happens if we multiply together $(p+q)\cdot(r+s)$. What we get is

$$(p+q)\cdot(r+s) \;=\; pr+ps+qr+qs.$$

This suggests that $ps+qr$ can be written as

$$ps+qr \;=\; (p+q)\cdot(r+s)-pr-qs.$$

This is perfect: to compute the 3 crucial terms, we only need to multiply p with r, q with s, and $p+q$ with $r+s$, and we will be just adding them up. This motivates the following improved algorithm:

```
1  fun fastMult(A,B) =
2    if |A| ≤ 1 then ⟨A_0 · B_0⟩
3    else let
4          val (p,q) = (A_{0..n/2}, A_{n/2+1..n-1})
5          val (r,s) = (B_{0..n/2}, B_{n/2+1..n-1})
6          val (suma,sumb) = (p+q, r+s)
7          val (pr,prod,qs) = par(fastMult(p,r)||fastMult(suma,sumb)||fastMult(q,s))
8          val sum = prod − pr − qs
9        in
10         shift(pr,n) + shift(sum,n/2) + qs
11       end
```

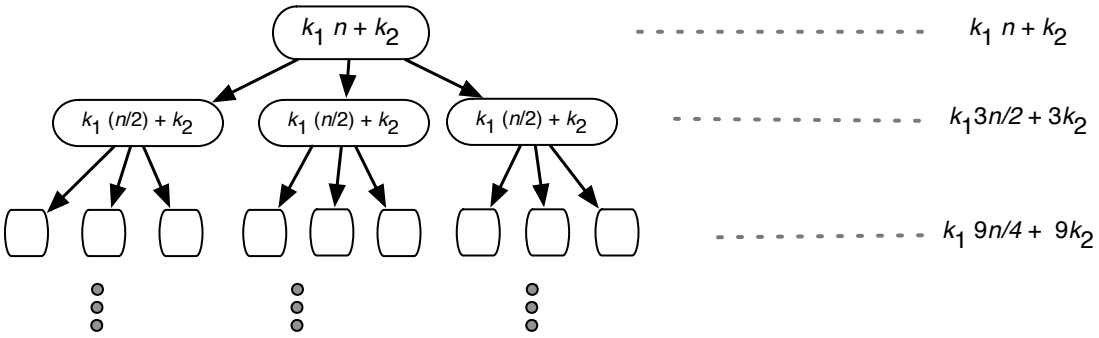
This time, we make only 3 recursive calls to `fastMult`, each on a problem of size $n/2$; therefore, our work/span recurrences become

$$W(n) = 3W(n/2)+O(n)$$
$$S(n) = S(n/2)+O(\log n)$$

Let's solve it, again, using the tree method. Already, we know that the span recurrence solves to $S(n) = O(\log^2 n)$; this is the exact same recurrence we reasoned about a while ago. For the work recurrence, applying the definition of big-O gives us

$$W(n) \leq 3W(n/2) + k_1 n + k_2 \qquad \text{for some } k_1, k_2 > 0.$$

Once more, we'll draw the recursion tree and annotate it cost. The tree looks more interesting this time:



Since every node has 3 children, the number of nodes at level i (counting from $i = 0$ at the root) is 3^i; however, the problem size shrinks by a factor of 2 at every level, so the problem size at level i is $n/2^i$ and the cost at each node at level i is at most $k_1 n/2^i + k_2$. This means that across level i, the cost is at most

$$3^i \left(k_1 \frac{n}{2^i} + k_2 \right)$$

Now, like before, there are at most $1 + \log_2 n$ levels because each level shrinks the problem size in half. Therefore, the total work is

$$\begin{aligned}
\sum_{i=0}^{\log_2 n} 3^i \left(k_1 \frac{n}{2^i} + k_2 \right) &= \sum_{i=0}^{\log_2 n} k_1 \cdot n \cdot \left(\tfrac{3}{2}\right)^i + k_2 \cdot 3^i \\
&\leq k_1 \cdot n \cdot (3/2)^{1+\log_2 n} + k_2 \cdot 3^{1+\log_2 n} \\
&= \tfrac{3}{2} k_1 \cdot n \cdot (3/2)^{\log_2 n} + 3k_2 \cdot 3^{\log_2 n} \\
&= \left(\tfrac{3}{2} k_1 + 3k_2\right) n^{\log_2 3} \in O(n^{\log_2 3})
\end{aligned}$$

(See the footnote[2] if you're puzzled about the second to last step.)

We conclude that $W(n) \in O(n^{\log_2 3})$ or about $O(n^{1.585})$. This is a significant improvement in terms of work over the previous algorithm, which was $O(n^2)$. Notice, though, that we're getting less parallelism because the simple divide-and-conquer algorithm gave us $n^2/\log^2 n$ parallelism while the "better" algorithm only gives us $n^{1.585}/\log^2 n$. Is this necessarily a bad thing?

[2] Notice that $n \cdot (3/2)^{\log_2 n} = n \cdot n^{\log_2(3/2)} = n \cdot n^{\log_2 3 - 1} = n^{\log_2 3}$.

3 Teaser — Maximum Contiguous Subsequence Sum Problem

As a teaser for next time, consider the maximum contiguous subsequence sum (MCSS) problem, defined as follows:

Definition 3.1 (The Maximum Contiguous Subsequence Sum (MCSS) Problem)**.** Given a sequence of numbers $s = \langle s_1, \ldots, s_n \rangle$, the *maximum contiguous subsequence sum* problem is to find

$$\max \left\{ \sum_{k=i}^{j} s_k \ : \ 1 \leq i \leq n, i \leq j \leq n \right\}.$$

(i.e., the sum of the contiguous subsequence of s that has the largest value).

3.1 Algorithm 1: Brute Force

Immediate from the definition is an algorithm with $O(n^3)$ work and $O(\log n)$ span. This algorithm examines all possible combinations of subsequences and for each one of them, it computes the sum and takes the maximum. Note that every subsequence of s can be represented by a starting position i and an ending position j. We will use the shorthand $s_{i..j}$ to denote the subsequence $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.

For each subsequence $i..j$, we can compute its sum by applying a plus `reduce`. This does $O(j-i)$ work and $O(\log(j-i))$ span. Furthermore, all the subsequences can be examined independently in parallel (using, e.g., `tabulate`). This leads the following bounds:

$$\begin{aligned} W(n) &= \sum_{1 \leq i \leq j \leq n} W_{\text{reduce}}(n) = \sum_{i \leq i \leq j \leq n} (j-i) = O(n^3) \\ S(n) &= \max_{1 \leq i \leq j \leq n} S_{\text{reduce}}(n) = \max_{i \leq i \leq j \leq n} \log(j-i) = O(\log n) \end{aligned}$$

Note that these bounds didn't include the cost of the max reduce taken over all these sequences. This max reduce has $O(n^2)$ work and $O(\log n)$ span[3]; therefore, the cost of max reduce is subsumed by the other costs analyzed above. Overall, this is a $O(n^3)$-work $O(\log n)$-span algorithm.

As you might notice already, this algorithm is clearly inefficient. We'll apply divide and conquer to come up with a more efficient solution.

Exercise 1. *Can you improve the work of the naïve algorithm to $O(n^2)$?*

[3]Note that it takes the maximum over $\binom{n}{2} \leq n^2$ values, but since $\log n^a = a \log n$, this is simply $O(\log n)$