Recall in last lecture, we looked at one way of parallelizing matrix multiplication. At the end of the lecture, we saw the reduce SUM operation, which divides the input into two halves, recursively calls itself to obtain the sum of these smaller inputs, and returns the sum of the results from those recursive calls. This is an example of applying the divide and conquer technique to solve a problem, which is a really useful technique, especially for parallel algorithms, so let's formalize it.

1 Divide and Conquer Algorithms

Divide and conquer algorithms generally have 3 steps: divide the problem into subproblems, recursively solve the subproblems and combine the solutions of subproblems to create the solution to the original problem.

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, 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 technique is even more useful for parallel algorithms. Generally, you can solve the subproblems in parallel. If the divide and combine step is inexpensive, then you are done. If either the divide and combine step (or both) is expensive, however, you may want to parallelize it (them), which can be difficult, depending on the algorithms.

Let us assume that the subproblems can be solved independently. Say the problem of size $n$ is broken into $k$ subproblems of size $n_1, \ldots, n_k$. How would you write the Cilk Plus program? What if $k = 2$?
F(n)
1  if n ≤ n₀
2    then Base-Case
3    return
4  Divide into 2 parts of size n₁ and n₂
5  spawn F(n₁)
6  F(n₂)
7  sync
8  Combine.

With the strategy above, the work is

\[ W(n) = W_{\text{divide}}(n) + W(n₁) + W(n₂) + W_{\text{combine}}(n) \]

And the span is

\[ S(n) = S_{\text{divide}}(n) + \max\{S(n₁), S(n₂)\} + S_{\text{combine}}(n) \]

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.

Now consider arbitrary \( k \). What is the pseudocode?

F(n)
1  if n ≤ n₀
2    then Base-Case
3    return
4  Divide into \( k \) parts of size \( n₁, n₂, \ldots, nₖ \)
5  parallel_for i ← 1 to k
6    do F(nᵢ)
7  Combine.

\[ W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(nᵢ) + W_{\text{combine}}(n) \]
And the span is

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

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—and learn how to derive a closed-form for them.

2 Solving matrix multiplication using divide and conquer

It turns out that, one can solve matrix multiplication using divide and conquer. You can divide your matrix into 4 quarters and get the following:

\[
\begin{align*}
C_{11} & = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} & = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} & = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} & = A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

This suggests a straightforward divide and conquer algorithm. You can compute all 8 parts in parallel and then add them:

```plaintext
MM(C, A, B, n)
1 if n = 1
2 then c_{11} ← a_{11}b_{11} return
3 partition A, B, and C, into 4 submatrices
4 create T, a temporary n × n matrix
5 spawn MM(C_{11}, A_{11}, B_{11}, n/2)
6 spawn MM(C_{12}, A_{11}, B_{12}, n/2)
7 spawn MM(C_{21}, A_{21}, B_{11}, n/2)
8 spawn MM(C_{22}, A_{21}, B_{12}, n/2)
9 spawn MM(T_{11}, A_{12}, B_{21}, n/2)
10 spawn MM(T_{12}, A_{12}, B_{22}, n/2)
11 spawn MM(T_{21}, A_{22}, B_{21}, n/2)
12 spawn MM(T_{22}, A_{22}, B_{22}, n/2)
13 sync
14 parallel_for i ← 1 to n
15 do parallel_for j ← 1 to n
16 do c_{ij} ← c_{ij} + t_{ij}
```
Then you must add the pairwise matrices (the combine step). Adding pair-wise matrices can also be done in parallel, either using 2 for loops or by dividing into 4 parts and doing it recursively. Either way, the work of adding $2 \times n$ matrices is $\theta(n^2)$ and the span is $\theta(\lg n)$.

Therefore, the work of the overall algorithm is $T_1(n) = 8T_1(n/2) + \theta(n^2) = \theta(n^3)$, and the span is $T_\infty(n) = T_\infty(n/2) + \theta(\lg n) = \theta(\lg^2 n)$.

How about if you didn’t want to create temporary matrices? You can do 4 recursive calls, sync, and the do the remaining 4 recursive calls. Doing so rid of the combine step. You get work $T_1(n) = 8T_1(n/2) + \theta(1)$, which is still $\theta(n^3)$ and span $T_\infty(n) = 2T_\infty(n/2) + \theta(1) = \theta(n)$, which is less but generally adequate parallelism. Even though this version of matrix multiply has less parallelism, but in practice, $\theta(n^2)$ is plenty of parallelism — even on a small input (e.g., 1000 by 1000 matrices), the amount of parallelism is already $10^6$. With even larger input, you would have more parallelism than you know what to do with. Thus, in practice, it’s actually better to use the version without the temporary, because it actually has smaller work overhead, albeit by constant amount due to less memory usage, that constant indeed makes a difference in running time in actual implementation.

**Strassen’s method**

Another interesting matrix multiplication algorithm is Strassen’s method. First, you divide input matrices into 4 submatrices respectively. Then you create 10 temporary matrices by adding or subtracting the input submatrices (think of this as part of divide step):

\[
\begin{align*}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{align*}
\]
Then you can recursively spawn the following subcomputations:

\[
\begin{align*}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{align*}
\]

Finally, you combine the results from the subcomputations:

\[
\begin{align*}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{align*}
\]

Now you have only 7 matrix multiplications and a bunch of additions. Therefore, the work is
\[T_1(n) = 7T_1(n/2) + \theta(n^2) = \theta(n^{\log_2 7}).\]

**Exercise 1** Parallelize Strassen’s algorithm and compute its span.

### 3 Maximum contiguous subsequence sum problem (MCSS)

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving. 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. Let’s take a look at an example: the maximum contiguous subsequence sum (MCSS) problem. MCSS can be defined as follows:

**Definition 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).

For example, the MCSS of a sequence \(\langle 2, -5, 4, 1, -2, 3 \rangle\) is is 6, via the subsequence \(\langle 4, 1, -2, 3 \rangle\).
Algorithm 1: Brute Force

The brute force 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 $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.

MCSS$[1..n]$
1. parallel_for all tuples $(i, j)$ such that $i \leftarrow 1$ to $n$ and $j \leftarrow i$ to $n$
2. do $A[i, j] \leftarrow$ spawn SUM$(i, j)$
3. MAX$(A)$

The total work is $O(n^3)$. We have learned in last lecture that we can use reduction to compute $\text{SUM}(i, j)$ in parallel, with the work of $\theta(n)$ and span of $\theta(\log n)$. Similarly, you can compute $\text{MAX}$ using reduction with the same asymptotic work and span. That means, the total work of this Brute Force algorithm is $\theta(n^3)$, and the span is $\theta(\log n)$.

Exercise 2 Can you improve the work of the naïve algorithm to $O(n^2)$? What does this do the span?

Algorithm 2: Divide And Conquer — Version 1.0

We’ll design a divide-and-conquer algorithm for this problem. If you took CSE241 with me, then you already know the basic algorithm.

Divide: Split the sequence in half.
Conquer: Recursively solve for both halves.
Combine: This is the most interesting step. For example, imagine we split the sequence in the middle and we get the following answers:

$\langle \ldots L \ldots || \ldots R \ldots \rangle$
\[
\downarrow
\]
$L = \langle \ldots \rangle_{\text{mcss=56}}$
$R = \langle \ldots \rangle_{\text{mcss=17}}$

There are 3 possibilities: (1) the maximum sum lies completely in the left subproblem, (2) the maximum sum lies completely in the right subproblem, and (3) the maximum sum spans across the split point. The first two cases are easy. The more interesting case is when the largest sum goes between the two subproblems. The maximum subsequence that spans the middle is equal to the largest sum of a suffix on the left and the largest sum of a prefix on the right.
Algorithm 3: Divide And Conquer — Version 2.0

As it turns out, we can do better than $O(n \log n)$ work. The key is to strengthen the (sub)problem—i.e., solving a problem that is slightly more general—to get a faster algorithm. Looking back at our previous divide-and-conquer algorithm, the “bottleneck” is that the combine step takes linear work. Is there any useful information from the subproblems we could have used to make the combine step take constant work instead of linear work?

In the design of our previous algorithm, we took advantage of the fact that if we know the max suffix sum and max prefix sums of the subproblems, we can produce the max subsequence sum in
constant time. The expensive part was in fact computing these prefix and suffix sums—we had to spend linear work because we didn’t know how generate the prefix and suffix sums for the next level up without recomputing these sums. Can we easily fix this?

The idea is to return the overall sum together with the max prefix and suffix sums, so we return a total of 4 values: the max subsequence sum, the max prefix sum, the max suffix sum, and the overall sum. Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. More specifically, we strengthen our problem to return a 4-tuple \((mcss, \text{max-prefix}, \text{max-suffix}, \text{total})\), and if the recursive calls return \((m_1, p_1, s_1, t_1)\) and \((m_2, p_2, s_2, t_2)\), then we return

\[
(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2)
\]

Exercise 4 Write the Cilk Plus code (or detailed pseudocode) to compute MCSS using the algorithm we have abstractly described here. Then, figure out the work and span for this algorithm.