We will look at another divide and conquer algorithm. Then we start looking at sorting.

1 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 \( \langle s_i, s_{i+1}, \ldots, s_j \rangle \).

\[
\text{MCSS}[1..n]
\]

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

**Exercise 1** Can you improve the work of the naïve algorithm to $O(n^2)$? What does this do to 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:

\[
\langle \quad L \quad || \quad R \quad \rangle
\]

\[
\downarrow
\]

\[
L = \langle \ldots \rangle \quad \text{mcss}=56
\]
\[
R = \langle \ldots \rangle \quad \text{mcss}=17
\]

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.

\[
\text{MCSS}(S, i, j)
\]

1. if $i = j$
2. then return $S[i]$
3. $\mid mid \leftarrow \left\lceil \frac{\text{end}-\text{start}}{2} \right\rceil$
4. $L \leftarrow \text{spawn MCSS}(S, i, mid)$
5. $R \leftarrow \text{spawn MCSS}(S, mid + 1, j)$
6. sync
7. $LS \leftarrow \text{spawn \text{MAX\text{PREFIX\text{SUM}}}}(S, i, mid)$
8. $RP \leftarrow \text{MAX\text{PREFIX\text{SUM}}}(S, mid + 1, j)$
9. sync
10. return $\max(L, LS + RP, R)$
How would you calculate the suffix and the prefix? If you do it in the naive sequential way (e.g.
keep the running prefix sum, and update the best as you encounter each element), then its work
and span is $\theta(n)$. Then the work of the MCSS algorithm is

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

and the span is

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

This is not great, since the parallelism is only $\theta(\log n)$. We will show next week how to compute
the max prefix and suffix sums in parallel, but for now, we’ll take it for granted that they can be
done in $\theta(n)$ work and $\theta(\log n)$ span. This reduces the span to

$$T_\infty(n) = T_\infty(n/2) + \theta(\log n) = \theta(\log^2 n)$$

**Exercise 2** Solve the work and span recurrences without using the master method. That is, use
the recursion tree method you learned in CSE 241. Also prove the bounds using induction, as you
learned in CSE 241.

**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 3** Write the Cilk Plus code (or detailed pseudocode) to compute MCSS using the algo-
rum we have abstractly described here. Then, figure out the work and span for this algorithm.
2 Reminder: Sequential Merge Sort

We are going to turn to the classic problem of sorting. Recall that given an array of numbers, a sorting algorithm permutes this array so that they are arranged in an increasing order. You are already familiar with basics of the sorting algorithm we are learning today, but we will see how to make it parallel today.

As a reminder, here is the pseudo-code for sequential Merge Sort.

\begin{verbatim}
MergeSort(A, n)
    if n = 1 then return A
    Divide A into two A_{left} and A_{right} each of size n/2
    A'_{left} ← MERGE_SORT(A_{left}, n/2)
    A'_{right} ← MERGE_SORT(A_{right}, n/2)
    Merge the two halves into A'
    return A'
\end{verbatim}

The running time of the merge procedure is \(\Theta(n)\). The overall running time (work) of the entire computation is \(W(n) = 2W(n/2) + \Theta(n) = \Theta(n \log n)\).

3 Let’s Make Mergesort Parallel

The most obvious thing to do to make the merge sort algorithm parallel is to make the recursive calls in parallel.

\begin{verbatim}
MergeSort(A, n)
    if n = 1 then return A
    Divide A into two A_{left} and A_{right} each of size n/2
    A'_{left} ← spawn MERGE_SORT(A_{left}, n/2)
    A'_{right} ← MERGE_SORT(A_{right}, n/2)
    sync
    Merge the two halves into A'
    return A'
\end{verbatim}

The work of the algorithm remains unchanged. What is the span? The recurrence is

\[ S_{MergeSort}(n) = S_{MergeSort}(n/2) + S_{merge}(n) \]
Since we are merging the arrays sequentially, the span of the merge call is $\Theta(n)$ and the recurrence solves to $S_{\text{MergeSort}}(n) = \Theta(n)$. Therefore, the parallelism of this merge sort operation is $\Theta(\lg n)$, which is very small. In general, you want the parallelism to be polynomial in $n$, not logarithmic in $n$.

What is the problem? It is the merge operation — doing merge sequentially is the bottleneck.

4 Let’s Parallelize the Merge in Mergesort

In Mergesort, we generally merge two arrays of the same size. However, in order to get this parallel merge to work, we have to be more general. We must learn how to merge two arrays which can be different in size.

Problem Statement: Given two arrays $B[1..m]$ and $C[1..l]$, each of which is sorted, we want to merge them into a sorted array $A[1..n]$ where $n = m + l$. Without loss of generality say that $m > l$. Here’s the procedure.

ParallelMerge$(B, m, C, l)$

1. if $m < l$
2. then return $\text{MERGE}(C, l, B, m)$
3. if $m = 1$,
4. then Concatenate the arrays in the right order and return.
5. $\text{mid} \leftarrow \lfloor m/2 \rfloor$
6. $s \leftarrow \text{SEARCH}(C, B[\text{mid}])$.
7. $A'_{\text{left}} \leftarrow \text{spawn } \text{MERGE}(B[1..\text{mid}], \text{mid}, C[1..s], s)$
8. $A'_{\text{right}} \leftarrow \text{spawn } \text{MERGE}(B[\text{mid}+1..m], m - \text{mid}, C[s+1..l], l - s)$
9. sync
10. Concatenate $A'_{\text{left}}$ and $A'_{\text{right}}$ and return

Let us calculate work and span. The search takes $\Theta(\lg n)$ work and span. Say $k = \text{mid} + s$. First, we notice that

\[
\begin{align*}
k &= \text{mid} + s \\
&= m/2 + s \\
&\leq m/2 + l \\
&\leq n/4 + n/2 \\
&= 3n/4
\end{align*}
\]

Also, we know that $k = m/2 \geq n/4$. Therefore, we have $n/4 \leq k \leq 3n/4$. 

5
\[ W_{\text{Merge}}(n) = W_{\text{Merge}}(k) + W_{\text{Merge}}(n - k) + \Theta(\lg n) \]
\[ = W_{\text{Merge}}(\alpha n) + W_{\text{Merge}}((1 - \alpha) n) + \Theta(\lg n) \quad \text{For some } 1/4 \leq \alpha \leq 3/4 \]
\[ = \Theta(n) \]

**Exercise 4** Show using induction that the recurrence \( W(n) = W(\alpha n) + W((1 - \alpha) n) + \Theta(\lg n) \) solves to \( \Theta(n) \).

For span, we have:
\[ S_{\text{Merge}}(n) = \max \{ S_{\text{Merge}}(k), S_{\text{Merge}}(n - k) \} + \Theta(\lg n) \]
\[ \leq S_{\text{Merge}}(3n/4) + \Theta(\lg n) \]
\[ = \Theta(\lg^2 n) \]

Note that parallelizing the Merge procedure did not increase its work — which is exactly what we want. It is a work-efficient algorithm. But we reduced the span from \( \Theta(n) \) to \( \Theta(\lg^2 n) \).