We came up with a parallel merge last time, but did not finish analyzing it.

1 Parallel Merge

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

\[
W_{\text{Merge}}(n) = W_{\text{Merge}}(k) + W_{\text{Merge}}(n - k) + \Theta(\lg n) \\
= W_{\text{Merge}}(\alpha n) + W_{\text{Merge}}((1 - \alpha)n) + \Theta(\lg n) \quad \text{for some } 1/4 \leq \alpha \leq 3/4 \\
= \Theta(n)
\]
Exercise 1.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 $\Theta(\lg^2 n)$.

Work and Span of Parallel Merge Sort using Parallel Merge

We can use this parallel merge procedure as a subroutine of merge sort and our work remains $\Theta(n \lg n)$. If we substitute the span of merge back into the Merge Sort equation, we get

$$S_{\text{MergeSort}}(n) = S_{\text{MergeSort}}(n/2) + S_{\text{Merge}}(n)$$
$$= S_{\text{MergeSort}}(n/2) + \Theta(\lg^2 n)$$
$$= \Theta(\lg^3 n)$$

Therefore, the parallelism of this new merge sort procedure is $\Theta(n \lg n / \lg^3 n) = \Theta(n / \lg^2 n)$. Therefore, we now have polynomial amount of parallelism.

2 Just for Kicks: Further Reduce the Span

We now see if we can further reduce the span of the Merge procedure. In practice, you probably don’t want to use this algorithm since generally, the previous algorithm has ample parallelism. However, this algorithm will teach you a technique for parallelizing algorithms that is quite general and quite interesting.

Our goal is to design a merge algorithm that has $O(n)$ work and $O(\lg n)$ span. In our previous algorithm, we divided the array into 2 parts and merged them recursively. Let’s try a different extreme. How about if I had arrays $B[1..n/2]$ and $C[1..n/2]$ and wanted to merge them. One very easy thing I can do is search for every element in $C$ within $B$. The span of this procedure is $O(\lg n)$, but the work is $O(n \lg n)$. So we managed to reduce the span at the cost of increasing the work, which is no good.

So how many binary searches can we afford to do? Each costs $O(\lg n)$ work and we can only afford $O(n)$ total work, so we can afford to do $O(n / \lg n)$ binary searches. So that’s what we will do.
We can divide $B$ and $C$ into $n/\lg n$ chunks each of size $O(\lg n)$. Say the boundary elements of these chunks are $b_1, b_2, ..., b_{n/\lg n}$ and $c_1, c_2, ..., c_{n/\lg n}$. We search for these boundary elements in the other array. That is we search for $b_1, b_2, ..., b_{n/\lg n}$ in $C$ using binary search. Since all of these binary searches can happen in parallel, the work of this step is $O(\lg n \times n/\lg n) = O(n)$ and the span is $O(\lg n)$. These searches lead to new boundary elements as shown in Figure 1.

Each of the chunks in the two arrays are size at most $\lg n$. Therefore, the corresponding chunks can now be merged sequentially with work and span $O(\lg n)$, and all the merges can occur in parallel. Therefore the total work is $O(n/\lg n)$ binary searches, each with cost $O(\lg n)$ added to $O(n/\lg n)$ sequential merge operations, each with the cost $O(\lg n)$, for a total work of $O(n)$. Similarly, span is the cost of the binary search added to the cost of a sequential merge of two arrays each of size $O(\lg n)$. Therefore, the total span is $O(\lg n)$.

**Figure 1**: Merging two arrays.

```plaintext
Merge(B, C, n)
1    if n ≤ 2
2    then Concatenate the arrays in the right order and return.
3    parallel_for i ← 1 to n/\lg n
4      do b_i ← i × \lg n
5    parallel_for i ← 1 to n/\lg n
6      do c_i ← i × \lg n
7    parallel_for (j, k) ← {(1, 1)} ∪ {(b_i, c'_i)} ∪ {(b'_i, c_i)}
8    parallel_for (j, k) ← {(1, 1)} ∪ {(b_i, c'_i)} ∪ {(b'_i, c_i)}
9    parallel_for (j, k) ← {(1, 1)} ∪ {(b_i, c'_i)} ∪ {(b'_i, c_i)}
10   do Use a sequential merge algorithm to merge the chunks starting at B[j] and C[k] and place the results
```

**Exercise 2** I have swept something under the rug here. How do you find out where your corresponding chunk, which starts at $B[j]$, $C[k]$ ends? Try to write the full pseudocode of this algorithm.
with all the gory details while keeping the work $O(n)$ and span $O(\lg n)$.

3 Quicksort

Quicksort is another recursive sorting algorithm. It is a randomized algorithm, however. The basic algorithm picks a pivot, partitions the array around the pivot and then sorts the two parts recursively.

```plaintext
QuickSort(A, p, q)
1  if p = q
2    then return
3  r ← Partition(A, p, q)
4  spawn QuickSort(A, p, r − 1)
5  QuickSort(A, r + 1, q)
6  sync
7  return
```

The sequential partitioning algorithm simply picks a random element in the sequence and places all the elements smaller than the pivot to the left of the pivot and all the elements larger than the pivot to the right of the pivot.

If we use sequential partition algorithm, then the work recurrence is $W(n) = W(k) + W(n − k − 1) + \Theta(n)$ where $k$ is the rank of the partition element. The worst case work is when we always pick $k = 0$ or $k = n − 1$. In this case, the work is $\Theta(n^2)$. The span recurrence is $S(n) = \max\{S(k), S(n − k − 1)\} + \Theta(n)$, which is also solves to $\Theta(n^2)$ in the worst case.

Since it is a randomized algorithm, we analyze the expected work and expected span. We analyzed the expected work in CSE241, and it was $\Theta(n \lg n)$; we won’t repeat the analysis here. The span recurrence is $S(n) = \max\{S(k), S(n − k − 1)\} + \Theta(n)$. Without even solving it, we can see that $E[S(n)] = \Omega(n)$. Actually, this is the same recurrence time recurrence for the randomized select algorithm that you saw in CSE241 also. Therefore, without solving it, let me just assert that it the $E[S(n)] = \Theta(n)$. Therefore, the parallelism is $\Theta(\lg n)$ which is not very high. Again, as in merge sort, we must parallelize the partition algorithm.

So what is the problem with parallelizing partition? We can easily compare all elements with the pivot in parallel. However, we don’t know where to put them in the partitioned array. We have to learn another algorithm called scan to learn how to do this.
4 Scan

Scan is a very useful primitive for parallel programming. We will use it all the time in this class. First, let’s start by thinking about what other primitives we have learned so far? The most important one was REDUCE — given \( n \) elements, you can return the sum (or really, apply any associative operator) of all elements in \( O(n) \) work and \( O(\lg n) \) span.

Today, we will learn a much more powerful primitive. Given a sequence \( A[1..n] \), the scan operation returns another sequence \( B[0..n] \) such that

\[
B[i] = \sum_{j=1}^{i} A[j]
\]

Where the sum here can be any associative operation. That is, you can have any operator or function \( f \), such that \( f(f(x, y), z) = f(x, f(y, z)) \). Since it computes prefix of each index, this operation is sometimes called prefix sums. Note that \( B[0] \) always contains the identity of the function you are applying and \( B[n] \) contains the answer to REDUCE(\( A \)).

One way to do a scan is to do it sequentially with \( O(n) \) work and \( O(n) \) span. Another way is to do \( n \) reduce operations \( O(n \lg n) \) work and \( O(\lg n) \) span. We want the best of both worlds.