1 Reducing Span of Merge Sort

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 $O(\lg n)$. 

1
Merge($B, C, n$)

1   \textbf{if} $n \leq 2$
2     \textbf{then} Concatenate the arrays in the right order and return.
3   \textbf{parallel} for $i \leftarrow 1$ to $n / \lg n$
4       do $b_i \leftarrow i \times \lg n$
5       $c'_i \leftarrow \text{BINARY SEARCH}(C, B[b_i])$
6   \textbf{parallel} for $i \leftarrow 1$ to $n / \lg n$
7       do $c_i \leftarrow i \times \lg n$
8       $b'_i \leftarrow \text{BINARY SEARCH}(B, C[c_i])$
9   \textbf{parallel} for $(j, k) \leftarrow \{(1, 1)\} \cup \{(b_i, c'_i)\} \cup \{(b'_i, c_i)\}$
10       do Use a sequential merge algorithm to merge the chunks starting at $B[j]$ and $C[k]$ and place the results in $A'$ starting at $A'[j + k - 1]$

Exercise 1 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 $O(\lg n)$.

2 Greedy Scheduling

In order to get good performance, not only do we need a work-efficient algorithm with ample parallelism, we also need to ensure that the strands are mapped onto the executing processors efficiently. In the model that we just discussed, a program can generate lots of strands on the fly, which can lead to a huge amount of parallelism, typically much more than the number of processors available when running. The model itself does not say how strands are mapped onto the processors, however. It is the job of a scheduler to decide how strands are mapped onto executing
processors (which and when).

Today, we are going to look at a simple scheduler, called greedy scheduler. We say that a scheduler is **greedy** if whenever there is a processor available and a strand ready to execute, then the strand will be scheduled on the processor and start running immediately.

Greedy schedulers have a very nice property that is summarized by the following:

**Theorem 1** On an ideal parallel computer with \( p \) processors, a greedy scheduler executes a multithreaded computation with work \( T_1 \) and span \( T_\infty \) in time

\[
T_p \leq \frac{T_1}{p} + T_\infty
\]

This is actually a very nice bound. For any scheduler, the time to execute the computation cannot be any better than \( \frac{T_1}{p} \), since we have a total of \( T_1 \) work to do, and the best we can possibly do is divide all the work evenly among the processors. Also note that the time to execute the computation cannot be any better than \( T_\infty \), since \( T_\infty \) represents the longest chain of sequential dependences. Therefore, the best we could do is:

\[
T_p \geq \max \left( \frac{T_1}{p}, T_\infty \right)
\]

We therefore see that a greedy scheduler does reasonably close to the best possible (within a factor of 2 of optimal). In particular, when \( \frac{W}{p} \gg S \) the difference between the two is very small. Indeed we can rewrite equation 1 above in terms of the parallelism \( P = \frac{W}{S} \) as follows:

\[
T_p < \frac{W}{p} + S \\
= \frac{W}{p} + \frac{W}{P} \\
= \frac{W}{p} \left( 1 + \frac{p}{P} \right)
\]

Therefore as long as \( P \gg p \) (the parallelism is much greater than the number of processors) then we get near perfect speedup (perfect speedup would be \( W/p \)).

**Proof of Theorem 1.** To prove this bound, let’s consider consider the execution dag \( G \) representing the computation with work \( T_1 \) and span \( T_\infty \). For convenience and without loss of generality, let’s assume each node in \( G \) represents one instruction (i.e., unit time). At each time step, either there are \( p \) or more nodes ready (call it a **complete step**), or there are less than \( p \) nodes ready.
(call it an **incomplete step**). We show this time bound by bounding the number of complete and incomplete steps a greedy scheduler takes before completing the execution.

Let’s first consider complete steps. In a complete step, a greedy scheduler executes $p$ nodes (picked arbitrarily among all the ready nodes), using all $p$ processors. The scheduler can’t take more than $\lfloor T_1/p \rfloor$ number of complete steps, because otherwise the total work performed in those complete steps would be greater than $T_1$.

Now we try to bound the number of incomplete steps. We assert that after each incomplete step, the longest path length decreases by one (we will come back to show this assertion later). Since $G$ starts out having the longest path length of $T_\infty$, and the longest path length decreases by one at each incomplete step, there can be at most $T_\infty$ number of incomplete steps.

What remains to be shown is the assertion that the longest path length decreases by one after each incomplete step. Let’s consider a complete step, and call the subdag that has yet to be executed at the beginning of this step as $G'$. Say $G'$ has longest path length $l$. The path(s) (there can be multiple longest paths having the same length) with length $l$ in $G'$ must starts with some node with in-degree 0 (i.e., a ready node), because otherwise $G'$ would have the longest path length $l + 1$. Since it’s an incomplete step, the greedy scheduler executes all ready nodes with in-degree 0, thereby decreasing the longest path length by one.

---

**In the real world:** No real schedulers are fully greedy. This is because there is overhead in scheduling the job. Therefore there will surely be some delay from when a job becomes ready until when it starts up. In practice, therefore, the efficiency of a scheduler is quite important to achieving good running time. Also the bounds we give do not account for memory affects. By moving a job we might have to move data along with it. Because of these affects the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.