CSE341T/CSE549T **09/15/2014**

Merge Sort

Lecture 6

Today 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.

1 Reminder: Sequential Merge Sort

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

MergeSort(A, n)
1 **if** $n = 1$
2 **then return** A
3 Divide A into two A_{left} and A_{right} each of size $n/2$
4 A'_{left} =MERGESORT$(A_{left}, n/2)$
5 A'_{right} =MERGESORT$(A_{right}, n/2)$
6 Merge the two halves into A'
7 **return** A'

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

2 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.

MergeSort(A, n)

1 **if** $n = 1$
2 **then return** A
3 Divide A into two A_{left} and A_{right} each of size $n/2$
4 $A'_{left} \leftarrow$**spawn** MERGESORT$(A_{left}, n/2)$
5 $A'_{right} \leftarrow$ MERGESORT$(A_{right}, n/2)$
6 **sync**
7 Merge the two halves into A'
8 **return** A'

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

$$S_{\text{MergeSort}}(n) = S_{\text{MergeSort}}(n/2) + S_{\text{merge}}(n)$$

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.

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

Let us calculate work and span. The search takes $\Theta(\lg n)$ work and span.

Say $k = mid + s$. First, we notice that

$$\begin{aligned} k &= mid + s \\ &= m/2 + s \\ &\leq m/2 + l \\ &\leq n/4 + n/2 \\ &= 3n/4 \end{aligned}$$

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

$$\begin{aligned} W_{\text{Merge}}(n) &= W_{\text{Merge}}(k) + W_{\text{Merge}}(n-k) + \Theta(\lg n) \\ &= W_{\text{Merge}}(\alpha n) + W_{\text{Merge}}((1-\alpha)n) + \Theta(\lg n) \text{ For some } 1/4 \leq \alpha \leq 3/4 \\ &= \Theta(n) \end{aligned}$$

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:

$$\begin{aligned} S_{\text{Merge}}(n) &= \max\{S_{\text{Merge}}(k), S_{\text{Merge}}(n-k)\} + \Theta(\lg n) \\ &\leq S_{\text{Merge}}(3n/4) + \Theta(\lg n) \\ &= \Theta(\lg^2 n) \end{aligned}$$

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

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

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.

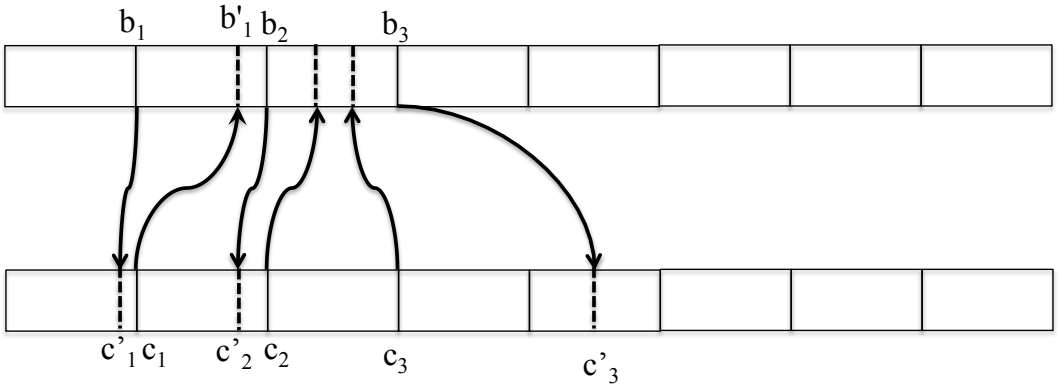

Figure 1: Merging two arrays.

4 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. Lets 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 Cinto $n/\lg n$ 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 $O(\lg n)$.

```
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           c'_i ← BINARYSEARCH(C, B[i])
 6  parallel_for i ← 1 to n/ lg n
 7        do c_i ← i × lg n
 8           b'_i ← BINARYSEARCH(B, 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 $O(\lg n)$.*