Problem Set 9 Solutions

Problem 9-1. More parallel merge sort

In this problem we will improve the parallel merge-sort algorithm from lecture. The algorithm
described in class has work $\Theta(n \lg n)$ and parallelism $\Theta(n/\lg n)$. We shall develop an algorithm
with the same work, but higher parallelism.

(a) Given two sorted arrays containing a total of $n$ elements, give an algorithm to find the
median of the $n$ elements in $\Theta(\lg n)$ time on one processor.

Solution: The basic idea is that if you are given two arrays $A$ and $B$ and know
the length of each, you can check whether an element $A[i]$ is the median in constant
time. Suppose that the median is $A[i]$. Since the array is sorted, it is greater than
exactly $i - 1$ values in array $A$. Then if it is the median, it is also greater than exactly
$j = \lceil n/2 \rceil - (i - 1)$ elements in $B$. It requires constant time to check if $B[j] \leq A[i] \leq B[j + 1]$. If $A[i]$ is not the median, then depending on whether $A[i]$ is greater
or less than $B[j]$ and $B[j + 1]$, you know that $A[i]$ is either greater than or less than
the median. Thus you can binary search for $A[i]$ in $\Theta(\lg n)$ worst-case time. The
pseudocode is as follows:

```
MEDIAN-SEARCH(A[1..l], B[1..m], left, right)
1  if left > right
2     then return MEDIAN-SEARCH(B, A, max(1, \lceil n/2 \rceil - l), min(m, \lceil n/2 \rceil))
3     i ← (left + right)/2
4     j ← \lceil n/2 \rceil - i
5     if (j = 0 \lor A[i] > B[j]) and (j = m \lor A[i] \leq B[j + 1])
6         then return (A, i) ▷ median = A[i]
7     elseif (j = 0 \lor A[i] > B[j]) and j ≠ m and A[i] > B[j + 1]
8         then return MEDIAN-SEARCH(A, B, left, i - 1) ▷ median < A[i]
9     else return MEDIAN-SEARCH(A, B, i + 1, right) ▷ median > A[i]
```

Let the default values for `left` and `right` be such that calling `MEDIAN-SEARCH(A, B)`
is equivalent to

```
MEDIAN-SEARCH(A[1..l], B[1..m], max(1, \lceil n/2 \rceil - m), min(l, \lceil n/2 \rceil))
```

The invariant in `MEDIAN-SEARCH(A, B)` is that the median is always in either $A[left..right]
or $B$. This is true for the initial call because $A$ and $B$ are sorted, so by the definition
of median it must be between $\max(1, \lceil n/2 \rceil - m)$ and $\min(l, \lceil n/2 \rceil)$, inclusive. It is
also true the recursive calls on lines 8 and 9, since the algorithm only eliminates parts of the array that cannot be the median by the definition of median. The recursive call on line 2 also preserves the invariant since if \( \text{left} > \text{right} \) the median must be in \( B \) between the new left and right values. When the algorithm terminates, the return value is the median by the definition of median. This algorithm is guaranteed to terminate because at each step \( \text{right} - \text{left} \) decreases and the median must be either in \( A \) or \( B \). The asymptotic worst-case running time of MEDIAN-SEARCH is the same as for performing two binary searches, which is \( \Theta(\lg n) \).

Alternatively, you can find the median by repeatedly comparing the median of the two arrays and discarding the part that cannot contain the median. It runs for at most \( 2\lfloor \lg n \rfloor \) iterations because each iteration discards half of one of the input arrays. Therefore this algorithm’s running time is also \( \Theta(\lg n) \).

(b) Using the algorithm in part (a) as a subroutine, give a multithreaded algorithm to merge two sorted arrays. Your algorithm should have \( \Theta(n) \) work and \( \Theta(n/\lg^2 n) \) parallelism. Give and solve the recurrences for work and critical-path length, and show that the parallelism is \( \Theta(n/\lg^2 n) \), as required.

Solution:

We can write a multithreaded algorithm for merging in which the smaller and larger halves of the sorted arrays are merged recursively in parallel:

\[
\text{P-MERGE}(A[1 \ldots l], B[1 \ldots m], T[1 \ldots n])
\]

1 if \( n \leq 0 \)
2    then return
3 \( (A, k) \leftarrow \text{MEDIAN-SEARCH}(A, B) \)  \( \triangleright \) wlog assume median is in \( A \)
4 if \( n = 1 \)
5    then \( T[1] \leftarrow A[1] \)
6 else spawn P-MERGE(\( A[1 \ldots k - 1], B[1 \ldots \lfloor n/2 \rfloor - k - 1], T[1 \ldots \lfloor n/2 \rfloor - 1] \))
7      spawn P-MERGE(\( A[k \ldots l], B[\lfloor n/2 \rfloor - k \ldots m], T[\lfloor n/2 \rfloor \ldots n] \))
8 sync

This algorithm merges two sorted arrays into a buffer \( T \). We can assume without loss of generality that the median is in \( A \); if it is in \( B \), we can just swap \( A \) and \( B \) in the remainder of the algorithm.

The work of this algorithm is \( T_1(n) = 2T_1(n/2) + \Theta(\lg n) = \Theta(n) \). The critical path length \( T_\infty(n) = T_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n) \). Therefore the parallelism is \( \frac{T_1}{T_\infty} = \Theta(n/\lg^2 n) \).

(c) Optional: Generalize the algorithm in part (a) to find an arbitrary order statistic. Using this algorithm, describe a merge-sorting algorithm with \( \Theta(n \lg n) \) work that achieves a parallelism of \( \Theta(n/\lg n) \).
Solution:
We first need a subroutine to find arbitrary order statistics. The following algorithm
finds kth order statistic of all the elements from two sorted arrays.

ORDER-STATISTICS (A[p..q], B[r..s], k)
1 if \( q \leq p + 1 \) or \( s \leq r + 1 \)  \( \triangleright \) One of the arrays has less than 3 elements
2 then Find the kth order statistic in constant time trivially and return it.
3 if \( k \leq s + q - p - r + 2 \) \( \triangleright \) kth element is less than or equal to the median element
4 then if \( A\left[\left\lfloor \frac{p+q}{2} \right\rfloor\right] < B\left[\left\lfloor \frac{r+s}{2} \right\rfloor\right] \) 
5 then return ORDER-STATISTICS (A[p..q], B[r..r+s], k)
6 else return ORDER-STATISTICS (A[p..p+q], B[r..s], k)
7 if \( k > s + q - p - r + 2 \) \( \triangleright \) kth element is greater than the median element
8 then if \( A\left[\left\lfloor \frac{p+q}{2} \right\rfloor\right] < B\left[\left\lfloor \frac{r+s}{2} \right\rfloor\right] \) 
9 then return ORDER-STATISTICS (A[\(\left\lfloor \frac{p+q}{2} \right\rfloor\) ..q], B[r..s], k - \( \left\lfloor \frac{p+q}{2} \right\rfloor + p - 1 \))
10 else return ORDER-STATISTICS (A[p..q], B[r..s], k - \( \left\lfloor \frac{r+s}{2} \right\rfloor + r - 1 \))

We now use this subroutine to merge two arrays. Instead of dividing arrays into 2 sub-
arrays, we instead find \( \sqrt{n} - 1 \) equally spaced order statistics (for \( k = \sqrt{n}, 2\sqrt{n}, \ldots, (\sqrt{n} - 1)\sqrt{n} \)) and merge \( \sqrt{n} \) subarrays in parallel. Therefore, the critical path of the merg-
ing subroutine is \( M_\infty(n) = M_\infty(\sqrt{n}) + \lg n \) and the solution of this recurrence is
\( M_\infty(n) = \Theta(\lg n) \). The work of the merge is \( M_1(n) = \sqrt{n}M_1(\sqrt{n}) + \Theta(\sqrt{n} \lg n) \),
whose solution is \( M_1(n) = \Theta(n) \). Therefore, the work of merge-sort remains \( \Theta(n) \),
while the critical path reduces to \( T_\infty(n) = T_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n) \).