Introduction to Algorithms December 14, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 35

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 $\Theta(n/\lg^2 n)$. 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, ⌈n/2⌉ − l), min(m, ⌈n/2⌉))
 3  i ← ⌊(left + right)/2⌋
 4  j ← ⌈n/2⌉ − i
 5  if (j = 0 ∨ A[i] > B[j]) and (j = m ∨ A[i] ≤ B[j + 1])
 6     then return (A, i)                          ▷ median = A[i]
 7  elseif (j = 0 ∨ 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

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

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 *left* $>$ *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 $right - left$ 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 $2\lceil \lg n \rceil$ 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:

```
    P-MERGE(A[1..l], B[1..m], T[1..n])
 1  if n ≤ 0
 2     then return
 3  (A, k) ← MEDIAN-SEARCH(A, B)        ▷ wlog assume median is in A
 4  if n = 1
 5     then T[1] ← A[1]
 6     else spawn P-MERGE(A[1..k−1], B[1..⌈n/2⌉ − k − 1], T[1..⌈n/2⌉ − 1)
 7          spawn P-MERGE(A[k..l], B[⌈n/2⌉ − k..m], T[⌈n/2⌉..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 $T_1/T_\infty = \Theta(n/\lg^2 n)$.

(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 $s \leq r+1$ ▷ 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** $k \leq s+q-p-r+2$ ▷ kth element is less than or equal to the median element
4 **then if** $A[\lceil \frac{p+q}{2} \rceil] < B[\lceil \frac{r+s}{2} \rceil]$
5 **then** **return** ORDER-STATISTICS$(A[p..q], B[r..\lfloor \frac{r+s}{2} \rfloor], k)$
6 **else** **return** ORDER-STATISTICS$(A[p..\lfloor \frac{p+q}{2} \rfloor], B[r..s], k)$
7 **if** $k > s+q-p-r+2$ ▷ kth element is greater than the median element
8 **then if** $A[\lfloor \frac{p+q}{2} \rfloor] < B[\lfloor \frac{r+s}{2} \rfloor]$
9 **then** **return** ORDER-STATISTICS $(A[\lceil \frac{p+q}{2} \rceil ..q], B[r..s], k - \lfloor \frac{p+q}{2} \rfloor + p - 1)$
10 **else** **return** ORDER-STATISTICS$(A[p..q], B[\lceil \frac{r+s}{2} \rceil ..s], k - \lfloor \frac{r+s}{2} \rfloor + r - 1)$

We now use this subroutine to merge two arrays. Instead of dividing arrays into 2 subarrays, 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 merging 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 $T_\infty(n) = T_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$.