Finding the Median of Two Sorted Arrays Efficiently

Merge operations integrate sorted arrays by triaging entries based on a median value. Finding that value accurately and efficiently requires careful work.

October 28, 2014
URL:http://www.drdobbs.com/parallel/finding-the-median-of-two-sorted-arrays/240169222

Parallel Merge and Parallel In-Place Merge algorithms merge two already sorted arrays (blocks) into a single sorted array using a similar divide-and-conquer strategy. The strategy picks the mid-element (X) within the larger of the two sorted blocks and uses binary search to split the smaller block into two sections: one with elements that are all smaller than (X), and the other with elements larger or equal to X, as shown in Figure 1:

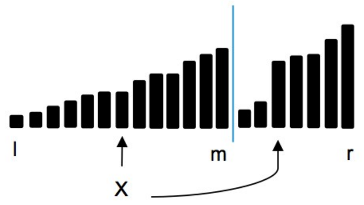

Figure 1: Merging two already sorted arrays.

The block on the left (between l and m indexes) is sorted and has 13 elements, with indexes 0 to 12. The mid-element X is at index 6. The block on the right (between m+1 and r indexes) has 7 elements. The value of X is used in the binary search to find an index where all elements to the left are smaller than X.

In the above example, there are 9 elements that are smaller or equal to X: 6 from the left block, 2 from the right block, and X itself. Also, 11 elements are larger or equal to X: 6 from the left block and 5 from the right block. Thus, using the mid-element X of the left block as the partition resulted in an uneven split: 9 elements smaller and 11 elements larger.

In the worst case, this "uneven-ness" of the split can be significantly worse, with one quarter of the elements being smaller and three quarters being larger, or vice versa. The worst case occurs when the two blocks are of equal size, and the split is such that either all elements of the right block are smaller than X, or all elements are larger or equal to X. For instance, if both blocks have M elements, and M is odd, then within the left block, M/2 elements will be smaller than or equal to X, and M/2 elements will be greater than or equal X. If the split is such that all elements of the right block are smaller than X, then overall we have M + M/2 elements that are smaller or equal to X and M/2 elements that are larger or equal to X. The opposite worst case split would occur if all elements of the right array are larger than or equal to X.

If one of these two worst cases occurred every time, then one side of the divide-and-conquer algorithm would be working on M/2 elements, while the other side would be working on M + M/2 elements — that is, one quarter of all elements and three quarters of overall elements, respectively. This uneven split will result in an unbalanced recursive binary tree during divide-and-conquer, performing deeper recursion than with an even split.

In this article, a better strategy is developed that always splits the two sorted blocks in a more optimal way by finding an overall median of the two blocks. Using this median in the same divide-and-conquer algorithm creates a more balanced recursive binary tree (as suggested in [1]).

Searching for the Median

The median of a single sorted array is trivial to find and is *O*(1) constant time. For example, a sorted array A = [5, 7, 9, 11, 15], which has an odd number of elements, has a unique median element 9 at index 2. Elements [5, 7] to the left are smaller than or equal to the median. Elements [11, 15] to the right are bigger than or equal to the median. The array has 5 elements with indexes in the range of 0 to 4, and the median element is at index (5-1)/2 = 2. In general, the median is at index (n-1)/2 if the number of elements in an array (n) is odd.

For a sorted array with an even number of elements, two elements in the middle are medians. For example, A = [5, 7, 9, 11] has medians of 7 and 9 at indexes of 1 and 2. In general, the medians are at index `floor`((n-1)/2) and at n/2. We could just pick the lower median, which leads to a single index computation no matter whether the array has an odd or an even number of elements, to wit, `floor`((n-1)/2).

Finding a median of two sorted arrays is more difficult and is no longer constant time. For example, two sorted arrays A = [5, 7, 9, 11, 15] and B = [1, 8] could be merged into a single sorted array in *O*(N) time to produce C = [1, 5, 7, 8, 9, 11, 15]. This resulting array has 7 elements, with a median of 8 at index `floor`((7-1)/2) = 3. It's possible to do better than *O*(N), by using a modified binary search leading to *O*(lgN) performance.

Modified Binary Search

In the case above, the median of value 8 came from the smaller of the two arrays. If we use different array values, A = [1, 2, 3, 4, 7] and B = [0, 5, 6, 9], then the combined sorted array C = [0, 1, 2, 3, 4, 5, 6, 7, 9] has a median of 4 at index 4. In this case, the median came from the larger array. This shows that the median can come from either array, no matter the size of either. Even if one of the arrays has a single element, that element could be the overall median. Thus, when we search for the median, both arrays must be involved in the search, otherwise we could potentially miss the true overall median.

Because both arrays are sorted, binary search can be used to search each of the arrays quickly, in *O*(lgN) time. The trick is to develop a test to determine quickly whether the selected element is the median or not — ideally, in constant time. Using a binary search, the mid-element of the sorted array A is selected, which is 3 at index 2. Is this element an overall median? What kind of a test is necessary?

Figure 2 shows two sorted arrays on the top line, in which an overall median is to be found. On the lower line, the two arrays have been merged into a single sorted array with the median shown, as the mid-element at offset 4 labeled with an M. Using a binary search within the first array, element G (for "guess") is selected as a guess for the overall median. Two elements on the right of G must be greater than or equal to G, since the array A is sorted. For G to be the overall median (M), it must be less than or equal to 4 elements, within the overall sorted array C. Two of these 4 elements come from array A, which implies that the rest of the elements (two more) must come from array B. Also, G must be greater than or equal to the rest of the elements in array B, i.e. B[0-1]. Since B is a sorted array, G only has to be greater than or equal to B[1], as B[0] ≤ B[1]. In other words, if B[1] ≤ G ≤ B[2], then G is the overall median M.

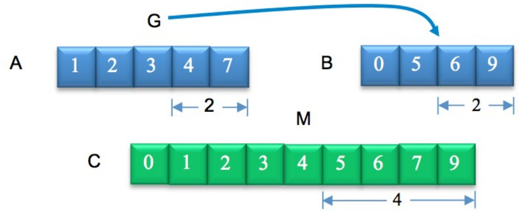

Figure 2: Two sorted arrays and a merge.

In Figure 2, the first guess G has a value of 3, and needs to satisfy B[1] ≤ 3 ≤ B[2], where B[1]=5 and B[2]=6. This guess G of 3 cannot be the overall median, since it fails to satisfy 5 ≤ 3 comparison. As G failed to be bigger than B[1], the modified binary search algorithm moves the guess to the right within A. An additional explanation video is available here.

A simple and quick, constant time, test has now been developed to see whether an element is an overall median. In principle, the algorithm consists of a binary search within the first sorted array A, where the split of array B is tested to see if the current guess G is an overall median. If the search fails within array A, then array B is searched, since the median could be in either array. Thus, in the worst case, both arrays are binary searched, resulting in O(lg(|A|) + lg(|B|)), where |A| is the size of A..

Listing One shows the implementation of the "Median of Two Sorted Arrays" algorithm.

The algorithm is broken into two parts: the outer part and the inner part. The outer part (`medianOfTwoSortedArrays_m2`), which is called by the user, performs setup, handles the special cases of one or both sorted arrays having zero elements, leaving the rest of the work for the inner algorithm. Interestingly, the same inner routine is performed twice: once for array A, and the second time for array B, but only if the overall median was not found in A.

The inner algorithm (`medianOfTwoSortedArrays_m2_inner` on line 8) performs a modified binary search on one of the arrays. It starts out with a full range over that array, and chooses the mid-element within that range. It then computes the split point within the other array and performs the comparisons (if possible) for the median test. Note the condition of the split being outside the bounds of the second array, or right at the edge, are handled graciously (lines 20 and 22). In case of being right at the boundary, only a single comparison is possible (lines 22 and 33).

When the current guess is found to be too small and needs to be increased for the next guess, the lower range boundary is adjusted to the current location plus one, to remove the current guess location, which failed to be the median from further consideration. When the current guess is found to be too large, the high boundary is adjusted similarly (lines 28 and 50). The search continues in this fashion, until either the overall median is found or the low and high boundaries cross, and thus there are no more elements that can be searched for within the current array.

Further Optimization

Let's say that we have 101 elements in sorted array A and only 2 elements in sorted array B. If the overall median is within A, then it's going to be close to the median of A itself, no matter what the values are in B. Thus, there is no reason to search all of A for the overall median. Narrowing the search range should speed-up the search. How much should the range be around the median of A? Maybe +/- length of B, or possibly even less…

There are only a few possible scenarios. In the first, all elements of B are smaller than median of A. In the second, all elements of B are larger than median of A. Or elements of B straddle the median of A. Of course, this is the case when B has only two elements. Otherwise, there are many more straddling possibilities. If all elements of B are smaller than median of A (first case), then the overall median starts at the median of A and moves higher by half the length of B. In case two, the overall median starts at the median of A and moves lower by half the length of B. The third case is somewhere in between cases one and two. Thus, it's only necessary to search +/- (1/2 length of B) away from the median of A, where A is the larger of the two arrays.

The order of the resulting algorithm is O(2*log(min(|A|,|B|))), since both arrays still need to be searched, but now reduced to the range of the smallest array, both times.

Conclusion

The algorithms presented in Parallel Merge and Parallel In-Place Merge are both *O*(log(min(|A|,|B|))), where |A| is the size of A, since the binary search is performed within the smaller array and is *O*(lgN). The algorithm presented in this article is *O*(log(|A|+|B|)), which is slightly worse. With further optimizations the order was reduced to O(log(2*min(|A|,|B|))), which is better, but is 2X more work, since both arrays may have to be searched. All algorithms are logarithmic.

Two binary searches were necessary to find an even split that produced two equal or nearly equal halves. Luckily, this part of the merge algorithm is not performance critical. So, more effort can be spent looking for a better split. Using this algorithm in the two parallel merges should balance the recursive binary tree of the divide-and-conquer and improve the worst-case performance of parallel merge sort.

I'll explore its integration into parallel merge and parallel merge sort in my next article.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Third Edition, MIT Press, Sept. 2009, p. 223, 804.

Related Articles

Parallel Merge Sort

Parallel Merge

Parallel In-Place Merge

Parallel In-Place Merge Sort

Victor Duvanenko is a frequent contributor to Dr. Dobb's *on algorithms for parallel programming.*

Listing One

```
// The median check consists of two checks:
// 1. Is the element to the right of this element equal to or larger
// 2. Is the element to the left  of this element equal to or smaller
// Returns a boolean to indicate whether the median was found in array A.
template< class _Type >
inline boolean medianOfTwoSortedArrays_m2_inner( const _Type* a, long a_left, long a_right, const _Type* b, long b_left, long b_right,
                                                                                                 long num_elements_larger_than_median_in_total, long* median )
{
        long low   = a_left;
        long high  = __max( a_left, a_right );
        while( low <= high )                                    // termination means no overall median within A
        {
                long overall_median           = ( low + high ) / 2;             // guess for the median as middle of A
                long num_larger_elements_in_a = a_right - overall_median;
                long num_larger_elements_in_b = num_elements_larger_than_median_in_total - num_larger_elements_in_a;
                long i_larger_in_b            = b_right - num_larger_elements_in_b + 1; // starting index in array B of elements larger than median of A
                // Compare with the left element in B if there is one
                if ( i_larger_in_b <= b_left )                          // no elements to the left in B. #2 check can't be performed
                {
                        if ( a[ overall_median ] <= b[ b_left ] && i_larger_in_b == b_left )            // satisfies being an overall median. check the element to the right (check #1)
                        {
                                *median = overall_median;
                                return true;                                            // median was found in A
                        }
                        else {  // a[ overall_median ] > b[ i_larger_in_b = 0 ] => guess of overal median is too big to be the overall median. Need to move overall_median within A to become smaller
                                high = overall_median - 1;                      // do not include it in our further searches, since it's not the overall median
                        }
                }       // i_larger_in_b > 0  => have a left element to compare with. Compare with the right element in B if there is one
                else if ( i_larger_in_b > b_right )                     // no elements to the right in B. #1 check can't be performed
                {
                        if ( b[ b_right ] <= a[ overall_median ] && ( i_larger_in_b - 1 ) == b_right )  // satisfies being an overall median
                        {
                                *median = overall_median;
                                return true;                                            // median was found in A
                        }
                        else {  // a[ overall_median ] < b[ i_larger_in_b = b_right ] => guess of overall median is too small to be the overall median. Need to move overall_median within A to become smaller
                                low = overall_median + 1;                       // do not iclude it in our further searches, since it's not the overall median
                        }
                }       // 0 < i_larger_in_b <= b_right => have two elements within B to compare with a[ overall_median ]
                else if ( b[ i_larger_in_b - 1 ] <= a[ overall_median ])        // check the element to the left  (check #2)
                {
                        if ( a[ overall_median ] <= b[ i_larger_in_b ])                 // check the element to the right (check #1)
                        {
                                *median = overall_median;
                                return true;                                            // median was found in A
                        }
                        else {
                                high = overall_median - 1;      // do not include it in our further searches, since it's not the overall median
                        }
                }
                else {  // a[ overall_median ] < b[ i_larger_in_b - 1 ] => overall median guess is too small and needs to be increased
                        low = overall_median + 1;               // do not iclude it in our further searches, since it's not the overall median
                }
        }
        *median = -1;
        return false;   // median was not found in A
}
// Searches both arrays, since the median can be in either array no matter the size of either array. Even if one of the arrays has a single element, that element could be the median.
// Both array bounds (left and right) are inclusive.
// Starts by searching within the first array and if the median is not found within the first array, then the search continues within the second array.
template< class _Type >
inline void medianOfTwoSortedArrays_m2( const _Type* a, long a_left, long a_right, const _Type* b, long b_left, long b_right, long* which_array, long* median )
{
        *which_array =  0;
        *median      = -1;              // by default set return median index to be invalid
        long a_length = a_right - a_left + 1;
        long b_length = b_right - b_left + 1;
        long total_length    = a_length + b_length;
        long median_in_total = ( total_length - 1 ) / 2;        // works for even or odd number of elements
        long num_elements_larger_than_median_in_total = ( total_length - 1 ) - median_in_total;         // high element in total minus median index
        // There may be a way to eliminate this special case of handling either one or both input arrays of length 0
        if ( a_length == 0 )
        {
                if ( b_length == 0 ) {
                        *which_array =  0;                      // both arrays are of zero length, return an invalid overall median index
                        *median      = -1;
                } else {        // b_length != 0
                        *which_array = 1;                       // median was found in B immediately since A has no elements
                        *median      = b_left + median_in_total;
                }
                return;
        }
        else if ( b_length == 0 ) {
                *which_array = 0;                               // median was found in A immediately since B has no elements
                *median      = a_left + median_in_total;
                return;
        }
        // a_length > 0 and b_length > 0
        if ( medianOfTwoSortedArrays_m2_inner( a, a_left, a_right, b, b_left, b_right, num_elements_larger_than_median_in_total, median ))
        {
                *which_array = 0;
                return;
        }
        // Search within B, since we didn't find it within A. It has to exist within B, since there is always an overall median.
        if ( medianOfTwoSortedArrays_m2_inner( b, b_left, b_right, a, a_left, a_right, num_elements_larger_than_median_in_total, median ))
        {
                *which_array = 1;
                return;
        }
        *which_array = -1;
        *median      = -2;      // There has to be an overall median at this point, flag an error condition if it wasn't found.
        return;
}
```