A BOTTOM-UP MERGESORT ELIMINATING RECURSION

M. Abdullah-Al-Wadud[1], Md. Amiruzzaman[2], Oksam Chae[1]
[1]Department of Computer Engineering
Kyung Hee University, Seocheon, Kiheung, Yongin, Gyonggi, South Korea, 449-701
[2]Department of Computer Engineering
Sejong University, 98 Gunja-dong, Gwangjin-gu, Seoul, South Korea, 143-747
E-mail: awsujon@yahoo.com, m_amiruzzaman@email.com, oschae@khu.ac.kr

Abstract. *In this paper an improved mergesort technique is proposed by us. The recursive calls are removed by using a bottom-up strategy to select two lists to merge. Some earlier improvements on the merge procedure, which are done by different researchers, into an efficient merge procedure that requires less space for auxiliary memory and less number of conditions checking also combined.*

Keyword: *Mergesort, recursive calls, auxiliary array, bottom-up.*

1. Introduction

The mergesort algorithms to sort a sequence of n elements are based on divide and conquer theory [1-16]. Merge-sort technique recursively divides the list of elements into two sub-lists until it gets a single element. Then it merges two single elements into one sorted list. Two such lists are then merged to form a sorted list of four elements. This procedure is repeated with larger number of elements through the so called conquer steps until it merges two n/2 sized sorted lists into a sorted list of n elements.

Several approach has been proposed to improve the performance of the basic mergesort algorithm such as top-down mergesort [4], natural mergesort [5], queue mergesort [8], in-place mergesort [9] etc. [6] proposes a technique to improve the asymptotic average-case cost of mergesort for sorting link lists. Some approaches have been proposed for hardware based sorting such as merge-sort on a linear array with a reconfigurable pipelined bus system [10], parallel mergesort for binary tree on chip network [3]. [4] proposes a method to cut the auxiliary array down to half while [5] deals with reducing some condition checks in loops. In [1], authors have reduced the necessity of dividing the last step of splitting (until a single element) and thereby shown an improvement in running time requiring less number of recursive calls to the divide and conquer procedure. However, to the best of our knowledge, no significant work is done yet to remove the recursive function calls, which certainly add some overhead on the performance of mergesort.

In this paper we first combine the approaches in [4] and [5] to form a more efficient merge procedure, and then present a proposal to eliminate the recursive function calls completely.

The organization of the rest of the paper is as follows. In Section 2, basic mergesort algorithm is briefly described. Section 3 presents the proposed improvements of mergesort, while Section 4 presents some simulation results. Finally, Section 5 concludes the paper.

2. Basic Mergesort Algorithm

Mergesort is composed of three steps: divide the list of elements into two halves, recursively sort them and then combine (conquer) them into a single sorted list. The traditional mergesort algorithm found in textbooks [2] is presented in Fig.1. Here a contains the list of elements to be merged and b is an auxiliary list. The time complexity of traditional mergesort algorithm is O(nlog(n)) [2].

Fig. 2 presents an example depicting the steps of mergesort algorithm to sort the list {3, 18, 5, 9, 11, 1, 22, 4} in ascending order.

3 Improving Mergesort

The complexity of mergesort algorithm is O(nlog(n)) as found in textbooks. This includes only the comparisons among the elements being sorted. However, there are some other factors that are not ignorable. In this paper we focus on some of these issues.

```
void mergesort (int *a, int low, int high)
{
        if (low < high)
        {
            int mid = (low+high)/2;
            mergesort(a, low, mid);
            mergesort(a, mid+1, high);
            merge(a, low, mid, high);
        }
}
```

(a)

```
void merge(int *a, int low, int mid, int high)
{
        // copy to an auxiliary array b.
        for (i = low; i ≤ high; i++)
            b[i] = a[i];

        i = low; j = mid+1; k = low;
        while (i ≤ mid && j ≤ high)
        {
                if (b[i] ≤ b[j])
                {   a[k]= b[i]; k = k+1; i = i+1; }
                else
                {   a[k] = b[j]; k = k+1; j = j+1; }
        }

        // copy back remaining elements of first half (if any)
        while (i ≤ mid)
        {   a[k] = b[i]; k = k+1; i = i+1; }
        // copy back remaining elements of second half (if any)
        while (j ≤ high)
        {   a[k] = b[j]; k = k+1; j = j+1; }
}
```

(b)

Fig. 1 Traditional mergesort algorithm. (a) Main control of the algorithm, (b) The merge procedure

For a single processor based system the key points, which have drawn attention of most researchers, to improve the performance of the mergesort algorithm mainly include reducing the number of comparisons (of course) and cutting down the size of required auxiliary array. In this connection, *[4]* proposes a method to cut the auxiliary array down to half while *[5]* deals with reducing some condition checking in loops. However, to the best of our knowledge, little attention is given to remove the recursive function calls, which certainly add some overhead on the performance of mergesort. Here we first merge the approaches in *[4]* and *[5]*, and then present a proposal to eliminate the recursive function calls completely.

3.1 Reducing Auxiliary Memory and Loop-Condition Checking

In *[4]*, authors notice that it is not necessary to copy the second half of array *a* to the auxiliary array *b* (in Fig. 1(b)). Doing so, it cuts the auxiliary array as well as the necessary copy operations to half of that needed in the basic approach. Moreover, if all elements of the first half have been copied back to *a*, the remaining elements of the second half need not be moved anymore since they are already at their proper places. Hence the improved version if merge() function may look like which is presented in Fig. 3.

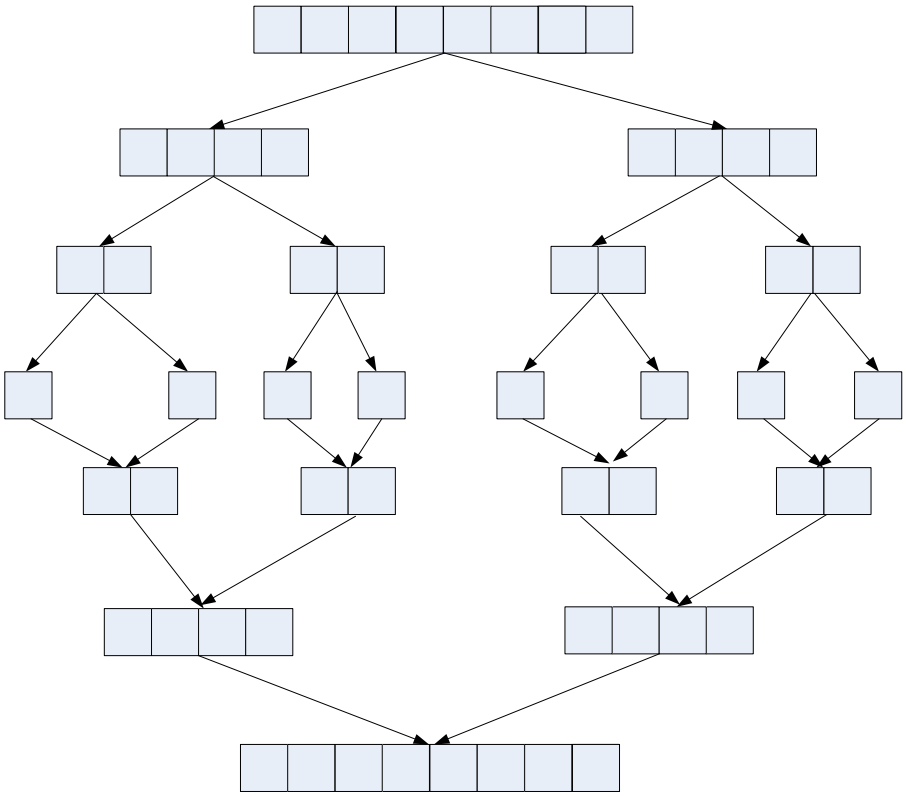

Fig. 2 An example showing the steps of mergesort

```
void merge_improved(int *a, int low, int mid, int high)
{
        i = 0; j = low;
        // copy first half of array a to auxiliary array b
        while (j ≤ mid)
        {   b[i] = a[j]; i = i+1; j = j+1; }

        i = 0, k =low;
        // copy back the next-largest element at each time
        while (k < j && j ≤ high)
        {
                if (b[i] ≤ a[j])
                {   a[k] = b[i]; k = k+1; i = i+1; }
                else
                {   a[k] = a[j]; k = k+1; j = j+1; }
        }
        // copy back remaining elements of first half (if any)
        while (k < j)
        {   a[k] = b[i]; k = k+1; i = i+1; }

}
```

Fig. 3 First Improvement of merge()

The second while-loop of the algorithm in Fig. 3 checks whether any of the two lists are ended. However, both the list will never be ended at the same time *[5]*. Hence, checking only the list that will end earlier is sufficient and cuts down almost half of the CPU time spent in checking the while-loop condition. The improved algorithm, along with the improvement done in Fig. 3, then looks like that in Fig 4.

```
void merge_final(int *a, int low, int mid, int high)
{
        if (a[mid] > a[high])
        {
                i=0; j=low;
                while (j<=mid)
                {   b[i]=a[j]; i=i+1; j = j+1; }

                 i=0; k=low;

                 while (j<=high)
                // no need to check whether the left list is finished
                {
                     if (b[i]<=a[j])
                     {    a[k]=b[i]; k = k+1; i = i+1; }
                     else
                     {    a[k] = a[j]; k = k+1; j = j+1; }
                }
                while (k < j)
                {    a[k]=b[i]; k = k+1; i = i+1; }
        }
        else
        {
                  i = 0; j = low;
                  while (j ≤ mid)
                   {   b[i] = a[j]; i = i+1; j = j+1; }

                  i = 0; k = low;

                  while (k < j)
                  // no need to check whether the right list is finished
                  {
                      if (b[i] ≤ a[j])
                      {    a[k] = b[i]; k = k+1; i = i+1; }
                      else
                      {    a[k] = a[j]; k = k+1; j = j+1; }
                  }
        }
}
```

Fig. 4 Further Improvement of merge()

3.2 Eliminating Recursive Calls

The overheads that may be associated with a function call are:

- *Space*: Every invocation of a function call may require space for parameters (and local variables), and for an indication of where to return when the function is finished. Typically this space is allocated on the stack and is released automatically when the function returns.
- *Time*: The operations involved in calling a function include allocating (and later releasing) local memory, copying values into the local memory for the parameters, branching to (and returning from) the function. All these operations contribute to the time overhead.

Hence, a recursive algorithm may need space and time proportional to the number of nested calls to the same function. Too much recursion may cause a stack overflow.

Thinking of these overhead, reducing the recursive calls in mergesort surely achieves some performance gain *[1]*. To sort a list of n elements, traditional mergesort algorithm calls (recursively) the function mergesort(), in Fig. 1(a), $2n$-1 times. Therefore, by eliminating all these calls, we can avoid good amount of operations.

Traditional mergesort algorithm works on a so called top-down fashion. Each time is divides n elements into two $n/2$ lists, recursively calls the same function to sort each of them and then merge them into a single list. To avoid the recursion, we proceed in a bottom-up approach. We start by merging two neighboring elements into sorted blocks of two elements. Each neighboring pair of such blocks are then merged to make blocks of four sorted elements.

This procedure continues until it merges all the elements into a single sorted block. One example of this approach is presented in Fig. 5. In fact here we cut down the upper half of the execution flow of basic mergesort (presented in Fig. 2). The algorithm of this modified approach is presented in Fig. 6. Here *levels* holds total number of levels to traverse to cover the upside down binary tree in Fig. 5. *block_size* is the size of two blocks that are going to be merged in current level.

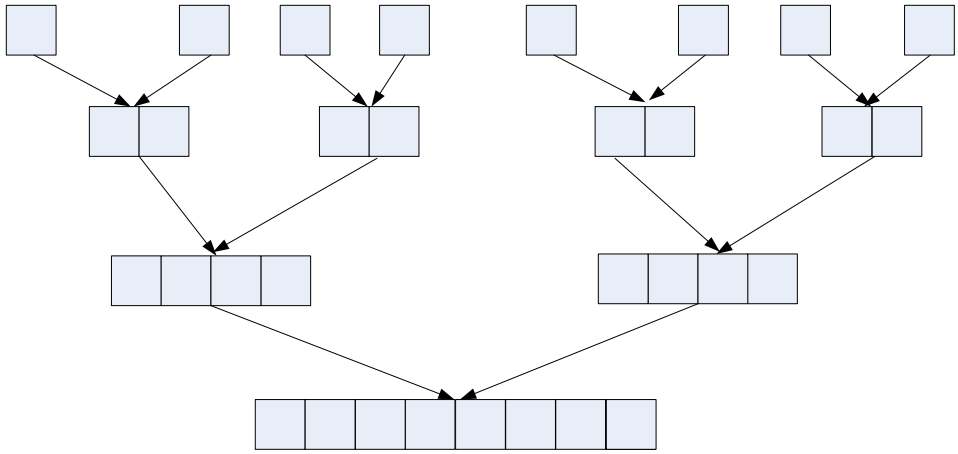

Fig. 5 An example showing the steps of mergesort without using any recursive call

```
void MergeSort(int *a, int n){
        levels = ⌈log2n⌉;
        levelCount=0, block_size=1;
        while (levelCount < levels)
        {
                left=0;
                while(1)
                {
                        mid = left+block_size-1;
                        right = mid+block_size;
                        if(right ≥ total)
                        {
                                right = total-1;
                                if(mid>=right)
                                        break;
                        }
                        merge_final(numbers_our,  left, mid, right);
                        left = right+1;
                }
                levelCount  = levelCount +1;
                block_size = block_size *2;
        }
}
```

Fig. 6 Algorithm of mergesort without using any recursive call

4 Experimental Results

For performance measurement, we have executed our proposed mergesort and the basic mergesort algorithm on a PC having Intel Pentium (R) D CPU 3.40 GHz and 2.00 GB of RAM. We have run them on same randomly generated data sets of different sizes. We have generated ten different data sets of same size, run the algorithms on them and taken the average time needed to sort the sets. Then we have plotted these data to have the graph in Fig. 7. It clearly shows better performance of the proposed algorithm. Moreover, also notice that the more data is given, the better is the performance. This also advocates for the proposed method.

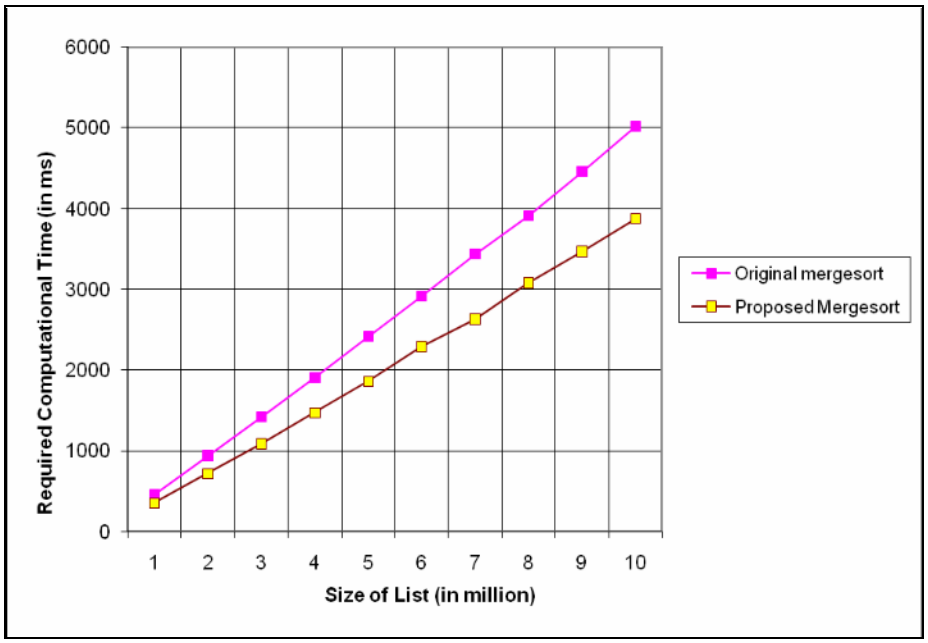

Fig. 7 Performance comparison of the proposed algorithm with traditional mergesort

5 Conclusions

In this paper we have presented an improved mergesort algorithm. We have eliminated the need of recursive calls by making the algorithm bottom-up. We also have combined two approaches to cut the auxiliary array in the traditional mergesort and to reduce some loop-conditions to speed up the computational time. Though the computational complexity remains the same as traditional one, it surely runs faster than the traditional implementations and the experimental results also support this claim.

References

[1] Hossain, N., Alma, M.G.R., Amiruzzaman, M., Qadir, S.M.M., "An Efficient Merge Sort Techniques that reduces both Times and Comparisons". Damascus, Syria, *ICCTA'04*, April 19-23, 2004, pp.537-538

[2] Azad, A.K.M., Kaykobad, M., "A Variation of Merge Sort Algorithm Requiring Fewer Comparisons", *Nat. Conf. Comp. & Info. Syst*. Dhaka, Dec 9-10, 1997

[3] Wong, S., Vassiliadis, S., Hur, J. Y., "Parallel Merge Sort on a Binary Tree On-Chip Network", *Proceedings ProRISC*, 2005

[4] SOMMERLAD, P., (Private Korrespondenz). Sommerlad, P., Rapperswil, H. f. T., Schweiz, 2004

[5] Katajainen, J., Träff, J. L., "A Meticulous Analysis of Mergesort Programs", *Proceedings of the Third Italian Conference on Algorithms and Complexity*, Vol. 1203, Lecture Notes In Computer Science, 1997, pp.217–228

[6] Roura, S., "Improving Mergesort for Linked Lists", J. Nˇesetˇril (Ed.): ESA'99, *Lecture Notes in Computer Science* 1643, 1999, pp.267–276

[7] Roura, S., "An improved master theorem for divide-and-conquer recurrences". *Proc. of the 24th International Colloquium (ICALP-97),* volume 1256 of *Lecture Notes in Computer Science*, Springer, 1997, pp.449–459.

[8] Golin, M.J., Sedgewick, R., "Queue-mergesort", *Information Processing Letter*, 48, 1993, pp. 253-259

[9] Katajainen, J., Pasanen, T., Teuhola J., "Practical in-place mergesort", *Nordic Journal of Computing*, 1996, pp.27-40

[10] Datta, A., Soundaralakshmi, S., Owens, R., "Fast Sorting Algorithms on Linear Array with a Reconfigurable Pipelined Bus Sytem", *IEEE Transactions on Parallel and Distributed Systems*, Volume 13, No. 3 March, 2002

[11] Knuth, D.E., *The art of Computer Programming: Sorting and Searching*, Volume 3, Addison Wesley, Reading, MA, 2nd edition, 1998

[12] Horowitz, E., Sahni, S., *Fundamentals of Computer Algorithms*, Galgotia Publication (p) Ltd., New Delhi, 1995

[13] Sedgewick, R., *Algorithms*, 3rd Edition, Addison Wesley, 1946

[14] Kleinberg, J., Tardos, E., *Algorithm Design*, Pearson International Edition, Addison Wesley, 2006, pp.210

[15] Sedgewick, R., *Algorithms*, 2nd edition, Addison-Wesley Publishing Company, Reading, Mass, 1988

[16] Knuth, D.E., *The Art of Computer Programming*, Vol.1: Fundamental Algorithms, Addison-Wesley Publishing Company, Reading, Mass, 1968