RESEARCH ARTICLE OPEN ACCESS

A Fast Merge Sort

D.Abhyankar
D. Abhyankar, School of Computer Science, D.A. University, Indore M.P. India

Abstract
Merge sort is one of the most efficient ways to solve sorting problem. Our research suggests that it is still responsive to clever optimizations. A considerably fast variation of Merge sort has been proposed in this paper. Proposed algorithm uses some novel optimizations to improve the speed of the proposed algorithm. Proposed algorithm not only applies some novel optimizations but also retains most of the old optimizations which were effective in reducing space and time. In addition to time and space efficiency, proposed algorithm offers the benefit of elegant design. Profiling has been carried out to verify the effectiveness of proposed algorithm. Profiling results reinforce the fact that proposed algorithm is significantly faster than existing best algorithm.

I. Introduction

Sorting is one of the most fascinating computational problems that demands a lot of computer time in practical applications. Merge sort is a stable divide and conquer algorithm that solves sorting efficiently. It requires an extra array of size Θ(n) to sort and thus it is not an in-place sorting algorithm. It spends running time Θ(n log n) to sort the input array [1]. Although a lot of research work has been carried out to improve the performance of Merge sort [2], it is still responsive to novel optimizations. Our research offers a better implementation of Merge sort that saves a lot of time and space. Proposed algorithm provides an extremely compact and fast Merge function. Also, it retains the advantages of existing Merge sort variations. This Section is followed by Section 2 that presents the proposed algorithm. Section 3 presents the comparative profiling statistics of the proposed implementation and the existing best Merge sort. In the end, Section 4 concludes and presents the essence of the paper.

II. Proposed Implementation

The most novel point in proposed algorithm is the amazingly clever sentinel trick accomplished by making the recursively sorted subarrays overlap. This trick eliminates a test in the inner loop by assuring right side of the array contains the last element. That assurance has a cost: lengthening the first of the two recursive sorts so they overlap. It is important to observe that the trick reduces the code size of Merge function. Actual improvement in terms of time has been assessed by profiling in Section 3. It is important to note that proposed implementation retains the existing optimizations on Merge sort. For instance, proposed implementation is a hybrid of Merge sort and Insertion sort. Also, the data move count has been reduced to an extremely low level. Following C++ code formally asserts the implementation.

```
void Merge(int*& a, int& h, int*& buf);
void MergeSort(int* a, int n, int* b, int m);
void Copy(int* s, int n, int* d);
void InsertionSort(int* a, int n);

 void Sort(int* a, int n)
    {
     if(n > 5)
       {
         int h = (n+1)/2;
         int* b = new int[h+1];
         MergeSort(b,h+1,a,1);
         a[h]=b[h];
         MergeSort(&a[h],n-h,a,0);
         Merge(a,h,b);
         delete[] b;
       }
       else{
             InsertionSort(a,n);
            }
      }
void MergeSort(int* a, int n, int* b, int m)
    {
      if(n > 5)
         {
           int h = (n/2);
           MergeSort(b,h+1,a,m+1);
           a[h]=b[h];
           MergeSort(&a[h],n-h,&b[h],m);
           Merge(a,h,b);
         }
       else{
           if((m%2)==1)
           Copy(b,n,a);
           InsertionSort(a,n);
            }
    }
inline void Merge(int*& a, int& h, int*& buf) // An ultimate Merge function
    {
    int i = 0; int k = 0; int j = h;
    while(1)
      {
          if(buf[i] <= a[j])
             {
             a[k] = buf[i];
              i++;
              if(i == h)
               return;
             }
           else{
               a[k] = a[j];
               j++;
               }
          k++;
      }
    }
void InsertionSort(int* a, int n) // Reduce function call count
     {
        int j = 1;
        while(j < n)
           {
             int x = a[j];  // Element to be inserted.
             int i = j - 1;
             while(i >=0)
                {
                 if(a[i] > x)
                   {
                      a[i+1] = a[i];
                       i--;
                   }
                 else break;
                }
            a[i+1]=x;  // Insertion
            j++;
         }
     }
```

III. Profiling Results

In order to measure the actual time improvement achieved by the proposed algorithm, profiling of the proposed algorithm and existing best algorithm was carried out. Visual Studio 2013 Ultimate software was used to measure the performance of the proposed and existing best algorithm. Details of existing best algorithm can be found in literature [2]. Profiling experiments were carried out on random data sets. Results suggest that proposed algorithm is considerably faster than the existing best algorithm. Following Table 1 and Figure 1 present the profiling statistics.

IV. Conclusion

This research work suggests that proposed algorithm is considerably faster than the existing best algorithm. It is important to restate that proposed algorithm retains the advantages of existing research work. For instance, the space requirement is just 50 % of what is required by classical Merge sort. Also, the proposed algorithm is more adaptive than its classical counterpart. Proposed algorithm delivers better performance on almost sorted data sets.

Table 1: Comparison on Random Input

N	Existing Best Algorithm (in ms)	Proposed Algorithm (in ms)
10000	5.74	2.95
20000	19.19	8.28
30000	22.19	11.92
40000	28.94	18.59
50000	35.91	19.21
60000	36.60	15.95
70000	38.81	22.11
80000	47.66	26.78
90000	759	29.68
100000	1119.80	34.09

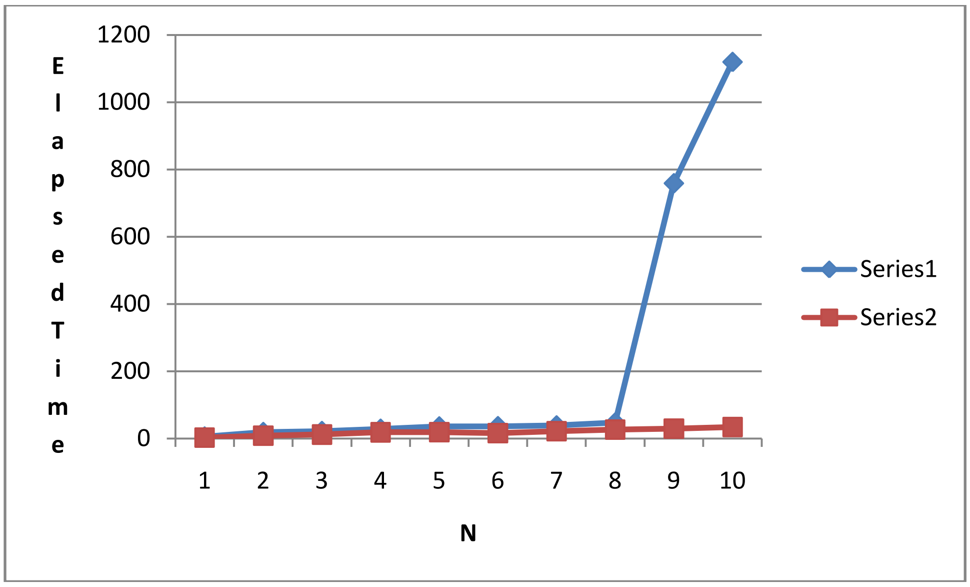

Figure 1: Comparative Performance

References

[1] Donald E. Knuth, The Art of Computer Programming, Vol. 3, Pearson Education, 1998.
[2] http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergef.htm.
[3] S. Baase and A. Gelder, Computer Algorithms:Introduction to Design and Analysis, Addison-Wesley, 2000.
[4] J. L. Bentley, "Programming Pearls: how to sort," Communications of the ACM, Vol. Issue 4, 1986, pp. 287-ff.
[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001.
[6] P. Biggar, N. Nash, K. Williams, D. Gregg, “An Experimental Study of Sorting and Branch Prediction ,” Journal of Experimental Algorithmics (JEA), Vol. 12, 2008.
[7] G. Graefe, “Implementing Sorting in Databases,” Computing Surveys (CSUR), Vol. 38, Issue 3. 2004.
[8] T. J. Rolfe, “List Processing: Sort Again, Naturally,” SIGCSE Bulletin ACM , Vol. 37, Issue 2, 2005.