IJCES International Journal of Computer Engineering Science , Volume1 Issue 3, December 2011
ISSN : 2250:3439
https://sites.google.com/site/ijcesjournal
http://www.ijces.com/

A Novel Mergesort

D.Abhyankar[1], M.Ingle[2]

[1]D. Abhyankar, School of Computer Science, D.A. University, Indore M.P. India
Email-id: deepak.scsit@gmail.com
[2]M.Ingle, School of Computer Science, D.A. University, Indore M.P. India
Email-id: maya_ingle@rediffmail.com

Abstract. Sorting is one of the most frequently needed computing tasks. Mergesort is one of the most elegant algorithms to solve the sorting problem. We present a novel sorting algorithm of Mergesort family that is more efficient than other Mergesort algorithms. Mathematical analysis of the proposed algorithm shows that it reduces the data move operations considerably. Profiling was done to verify the impact of proposed algorithm on time spent. Profiling results show that proposed algorithm shows considerable improvement over conventional Mergesort in the case of large records. Also, in the case of small records, proposed algorithm is faster than the classical Mergesort. Moreover the proposed algorithm is found to be more adaptive than Classical Mergesort.

1 Introduction

Sorting is one of the fundamental problems in Computer Science that spends a lot of computer time in the real world applications. Mergesort, which was invented by John Von Newman, is one of the most elegant algorithms to appear in the sorting literature. It is the first sorting algorithm to have O(n log n) time complexity bound. It is stable i.e. it maintains original order of records on equal keys [1, 2, 4]. It is not in place because it needs extra space in the form of an array and stack. It is important to observe that Mergesort spends a lot of time on data transfer operations. In fact standard Mergesort incurs about 2n data move operations. This count is improved by Ping Pong strategy, a variation of bottom up Mergesort, to the level of roughly n data moves. It is interesting to note that ping pong strategy is previous best in terms of data move operations.

Our study suggests that data move count can further be reduced to a new low. We have proposed a novel Mergesort that reduces the data move count by 50% in best case and 25 % in average case. This indicates a clear improvement over other Mergesort algorithms. Moreover, it was found that impact of this reduction on total time is significant when it is run on large records. On large records total time is reduced by approximately 60% in our experiments. This Section is followed by Section 2 that paraphrases the standard Mergesort. Section 3 contains the proposed algorithm, whereas Section 4 provides the analysis of proposed algorithm. Section 5 provides the empirical results, and Section 6 concludes and presents the essence of the paper.

2. Conventional Mergesort

Standard Mergesort is an O (n log n) time complexity, comparison based, internal sorting algorithm. Conceptually it works as follows.

1. If the array is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted array into two sub arrays of about half the size.
3. Sort each sub array recursively.
4. Merge the two sub arrays back into one array.

In simple C++ code, algorithm is formally presented here.

```
void Merge(int* a, int l, int n);

void MergeSort(int* a, int n)
      {
      if( n > 1)
            {
            int  l = (n/2);
            int  r = n - l;
            MergeSort(a, l);
            MergeSort(&a[l],r);
            Merge(a,l,n);
             }
}

void Merge(int* a, int l, int n)
{
int* t = new int[n];
int k = 0;
while(k < n)  // First loop
{
      t[k] = a[k];
      k++;
}
int i = 0; int j = l;  k = 0;
while(j < n)  // Second loop
{
      If(t[i] <= t[j])
            {
            a[k] = t[i];
            i++;
            if(i==l) return;
            }

      else{
      a[k] = t[j];
      j++;
      }
k++;
}
while(i < l)  // Third Loop
{
      a[k] = t[i];
      i++; k++;
}
}
```

3. Proposed Algorithm

Proposed algorithm is based on mutual recursion and it is comprised of 2 two mutually recursive Mergesort routines named as Mergesort1 and Mergesort2. Mergesort1 sorts in the same array whereas Mergesort2 puts the sorted data in a buffer. Following C++ code formally express the proposed algorithm.

```
void  MergeSort1(int* a, int n);  // Final sorted output will be in array a

void  MergeSort2(int* s, int n, int* d); // Final sorted output will be in
array d
void Merge(int* Left, int l, int n, int* a); // A better Merge function
void MergeSort1(int* a, int n)
{
if(n > 1)
      {
      int l = (n/2);  // Size of Left Subarray is l = (n/2)
      int r = n - l;  // Size of Right Subarray r = n - l
      MergeSort1(&(a[l]), r);  // Sort the Right Subarray
      int* t = new int[l];          // Allocate Memory to temporary array
      MergeSort2(a, l, t);        // Sort the Left Subarray
      Merge(t,l,n,a);                 //  After merging entire sorted output
will be in a
      delete[] t;                      // free the memory of temporary array
t
      }
}
void MergeSort2(int* s, int n, int* d)
{
if(n > 1)
      {
      int l = (n/2);  // Size of Left Subarray is l = (n/2)
      int r = n - l;  // Size of Right Subarray r = n - l
      MergeSort1(s,l); // Sort the Left Subarray
      MergeSort2(&(s[l]), r, &d[l]); // Sort the Right Subarray
      Merge(s,l,n,d);  //  After merging entire sorted output will be in d
      }
    else    {
      d[0] = s[0];   // Trivial case
      }
}
void Merge(int* Left, int l, int n, int* a) // An exceptionally efficient
Merge function
      {
      int i = 0;
      int j = l;
      int k = 0;
      while(i < l)  // While Left Subarray is not consumed
            {
            //only loop with at most n data moves.
                  if(Left[i] <= a[j])
                  {
                  a[k] = Left[i];
                  i++;
                  }
                  else{
                  a[k] = a[j];
                  j++;
                  if(j == n)
                        {
                         j--;
                        a[j] = a[l-1];
```

```
                }
            }
            k++;
        }

    }
```

4. Mathematical Analysis

Proposed algorithm is an instance of mutual recursive design. This mutual recursive design allows an unusually efficient, light weight Merge function that is not technically feasible in standard Mergesort design. Our Merge function avoids one of the two array copy loops, and thereby saves n data move operations. It is plain from the code that proposed algorithm incurs n data move operations in the worst case. It can be observed that on reverse order sorted data the worst case will occur. In the best case proposed algorithm will incur (n/2) data moves. It can be seen that best case will occur on sorted data. If we assume all inputs equally likely and then take the average we find that the proposed algorithm costs approximately (3n/4) data moves on average. It is interesting to note that the Conventional Merge sort incurs double the operations incurred by proposed algorithm. Even code size has been reduced to a single loop, whereas Merge routines of other variations of Mergesort involve code that contains three loops. Compact Merge function is a virtue of the proposed work because it has a synergistic effect on the overall performance of Mergesort.

5. Profiling Results

In order to measure the actual time improvement achieved by the proposed algorithm, profiling of the proposed algorithm was carried out. Aqtime software was used to measure the performance of the proposed and existing algorithms. This algorithm was found extremely useful in cases where one has to deal with large records. This paper contains the performance statistics of the proposed and existing algorithm on large records. It was found that for large records proposed algorithm is approximately 3 times faster than the classical Mergesort. In the experiment, record size of 1024 bytes was chosen. Following Table 1 and Fig. 1 show the time statistics for large records. Moreover experiment was done on almost sorted data and performance was found to be better than conventional Mergesort even on almost sorted input or completely sorted input. Therefore proposed algorithm is comparatively more adaptive than classical Merge Sort.

Table 1 Comparison on Large Records and Random Input

N	Classical Mergesort (in ms.)	Proposed Mergesort (in ms.)
100	1.66	0.35
200	2.53	0.77
300	13.58	1.38
400	5.70	2.12
500	7.42	3.62
600	10.83	4.92
700	20.49	5.08
800	13.98	4.82

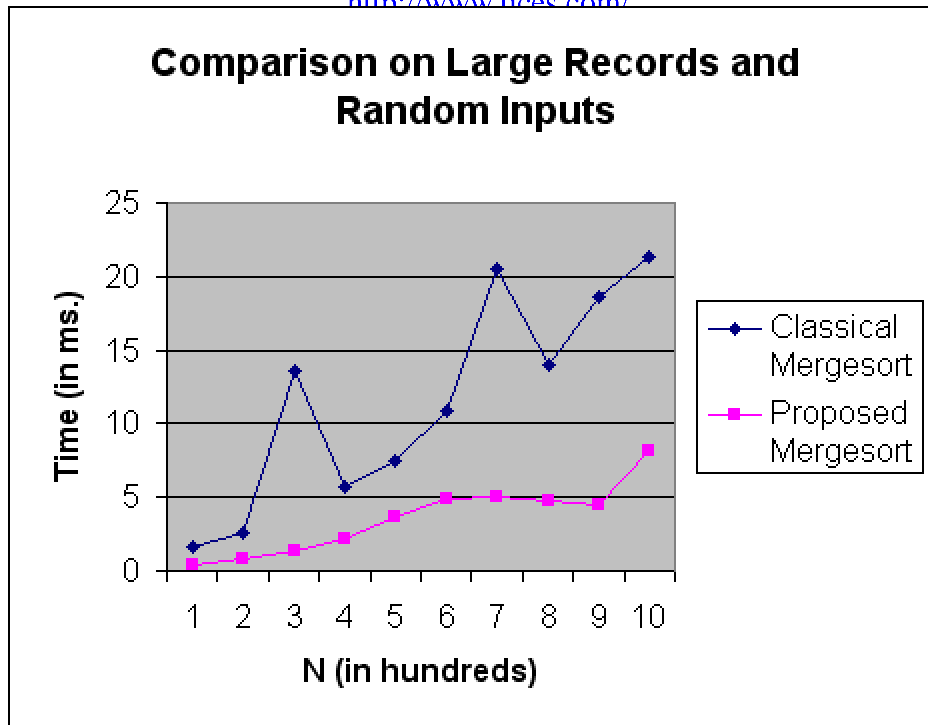

Fig. 1

Also, profiling on small records was done that suggests the performance improvement over classical Mergesort. Fig. 2 shows the comparative performance statistics of the two variations of proposed Mergesort with the classical top down Mergesort in a graphical form.

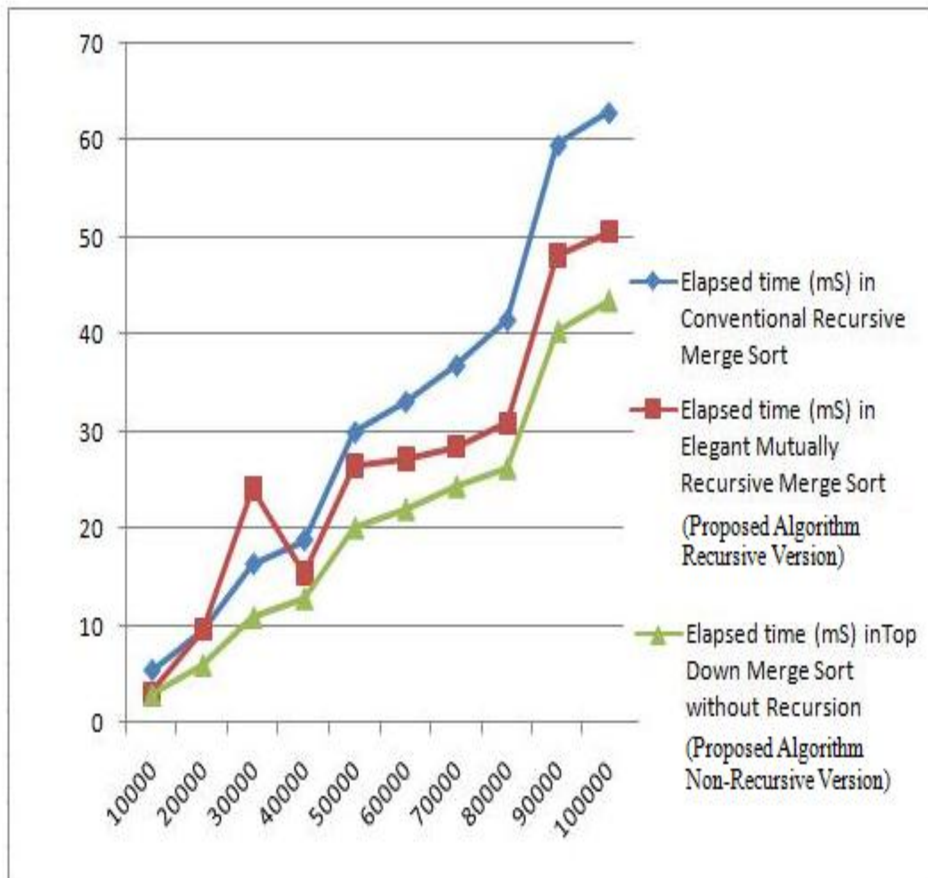

Fig. 2

Moreover, profiling was carried out on almost sorted data and performance was found to be better than conventional Mergesort even on almost sorted input and completely sorted input. Therefore, proposed algorithm is comparatively more adaptive than classical Merge Sort.

6. Conclusion

This study finds that proposed algorithm shows significant performance improvement over classical Mergesort. Proposed algorithm has a fast inner loop. Data move count is almost half than what is incurred by classical Mergesort. In fact, data move count is lesser than the same count experienced by Ping Pong strategy. It is evident that on large records the proposed algorithm does better than any other member of Mergesort family. It is not unusual for database applications to have large records and that is where the proposed algorithm will be effective. Even on small records, algorithm is faster than any other existing Mergesort. Also, the proposed algorithm delivers better performance on almost sorted data.

References

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