Advanced Pattern Based Merge Sort With Pooling

Shanti Swaroop Moharana, Satya Prakash Sahoo, Dr. Manas Ranjan Kabat
1,2,3Dept. of Computer Science & Engineering, Veer Surendra Sai University of Tech. Burla, India

Abstract

Sorting is the arrangement of records or elements in some mathematical or logical order. Merge sort is one among many sorting techniques. Several improvements on the merge sort procedure are done by different researchers that require less space for auxiliary memory and less number of conditions checking. In this paper, a new merge-sort technique is proposed which uses the design patterns to reduce computational complexity of swapping operation and memory usage. The order of settlement of elements is recorded by various design patterns and then merging of one phase of elements with another is replaced with chunk merging. The proposed technique uses the design patterns to create a memory pool of data to sort data. This algorithm has been implemented in JAVA and it is observed that the proposed algorithm performs better than all existing merge sort algorithms.

Keywords

Sorting, Pooling, Heuristic, Pattern Based

I. Introduction

Sorting [7, 16], is the arrangement of records or element in some mathematical or logical form. Proper arrangement of elements makes searching and selecting a data element easy and hence it saves time. There are many fundamental and advance sorting algorithms in computer science. These sorting algorithms are classified by various research papers [1-3], which are listed as follows:-

- Swap based- Sorts begin conceptually with the entire list, and exchange of particular pairs of elements (adjacent elements or elements with certain step like in Shell sort) takes place.
- Merge Based Swap. eg Merge Sort.
- Tree based sorts. eg Heap Sort.
- Other. eg Radix Sort or Bucket Sort.

Although many people consider sorting as a solved problem, but useful new sorting algorithms are still being invented for example, library sort in 2004, Bottom up merge sort in 2008 and Heuristic merge Sort in 2010. Mergesort [4-5], is a sorting algorithm which is used for sorting data stored either in internal or external storage device. This algorithm is based on divide and conquer theory.

- Merge-sort technique recursively divides a list of ‘n’ elements into two sub-lists n1 and n2 until it gets a single element.
- Then it merges two single elements into one sorted list and then two elements and proceeds in this way. Two such lists are then merged to form a sorted list of four elements.

The above 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 approaches have been proposed to improve the performance of the merge sort algorithm such as top down merge sort [8], natural merge sort [10], queue merge sort [9], in place merge sort [6], bottom up merge sort [11], heuristic merge sort [12], etc.

Top-Down merge sort [8], uses a technique to cut the auxiliary array down to half. However no significant effort has been made in top down merge sort [8], to reduce the recursive merge function calls which may increase the execution time. Another sorting technique natural merge sort [10], reduces some condition checks in loops. However the recursive merge function calls still exist in natural merge sort [10], which may increase the execution time. Bottom-Up merge sort [11], uses a non recursive procedure for sorting numbers which eliminates the divide procedure completely. However, although elimination of recursive function calls decreases time but actually the sorting time increases. This increase in sorting time occurs because the algorithm actually works similar to the worst case merge sort algorithm, comparing each and every number with the other number. Heuristic Merge Sort [12], algorithm searches a specific pattern (consecutive ascending or consecutive descending) of numbers in the given list of numbers. It extracts sub lists of sorted data, and then uses the merge procedure to merge all the extracted sub lists. However the list extracted may contain very small chunk (group of consecutive sorted data) because the procedure extracts only consecutive numbers which are already sorted. Finding sorted data in a random list is very sparse and hence the chunks formed are of small size. Since the sublists are of small size more number of merge calls are required. Hence there is no significant decrease in execution time in case of Heuristic merge sort [12].

In this paper, we present a proposal to form larger pool of sorted data for merging thereby reducing the number of merge function calls. Our proposed technique uses the design patterns to create a memory pool of data to sort data more efficiently. The order of Settlement of elements is recorded by a design pattern and then merging of one phase of elements with another is done. The design pattern reduces computational complexity of swapping operation and memory usage. We have also combined the various approaches which were used in heuristic merge sort [12], and bottom up merge sort [11], to form the merge sort procedure a more efficient one. We have reduced the necessity of dividing the list or the 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.

II. Motivation

A pool in computer science is a set of initialized resources that are kept ready to use rather than allocated and destroyed on demand. A client of the pool requests an object from the pool and performs various operations on the returned object. When the client has finished with an object (or resource), it returns it to the pool, rather than destroying it. Pooling of resources can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled resource is obtained in predictable time when creation of the new objects (especially over network) may take variable time.

Example of Pooling-

The idea of object (or resource) pooling is similar to the operation of your local library. When you want to read a book, you know that it’s cheaper to borrow a copy from the library rather than purchase your own copy. Likewise, it is cheaper (in relation to memory and speed) for a process to borrow an object rather than create its own copy. In other words, the books in the library represent objects and the library patrons represent the processes. When a process needs an object, it checks out a copy from an object pool rather than instantiate a new one. The process then returns the object to the pool when it is no longer needed.
However, these benefits are mostly true for objects which are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps.

Since the merging process is a high cost operation, we have tried to reduce the number of merge function calls. We noticed that instead of extracting smaller sub list again and again from the given list, if we can create a larger pool of sorted data then larger pool of sorted data is provided as input to the merge procedure which may decrease the execution time significantly.

### III. Advanced Pattern Based Merge Sort

Our proposed algorithm targets the pattern of input data which consists of successive sorted numbers either in ascending or descending order. The proposed algorithm scans the numbers in two directions that is from forward direction as well as from backward direction. By alternating the scanning direction we ensure that we always get the maximum number of numbers. Our algorithm uses the modulus operator for alternating the scanning direction. The numbers which are extracted are then stored in a larger memory pool. Finally an efficient merging procedure is applied for merging the numbers present in the pooled set of numbers.

```plaintext
Algorithm merge_sort_pooling(a)
1. Input a
   /* a is the given set of numbers to be sorted*/
2. Input n,lenc,key,middle
   /* n represents the number of elements that are sorted till now.
   lenc represents the total number of numbers to be sorted. Key
   represents the element which is currently selected. middle
   represents the number of elements in the current sorted list. */
3. Input c
   /*c is the completely sorted set of numbers*/
4. while(n<lenc)
5.   if(n%2==0)
6.      key=a[(lenc-1)]; middle=0;
7.   for(i=(lenc-1);i>=0;i--)
8.      if((a[i]==-1)||(key>a[i]))
9.         continue;
10.     endif
11.    c[n]=a[i];
12.    key=a[i];
13.    a[i]=-1; n++; middle++;
14.  end for
15. else
16.    key=a[0]; middle=0;
17.    for(i=0;i<lenc;i++)
18.      if((a[i]==-1)||(key>a[i]))
19.         continue;
20.    endif
21.    c[n]=a[i]; key=a[i];
22.    a[i]=-1; n++; middle++;
23.  end for
24. merge_pool(c,b,0,(n-middle),n-1);
25. end while
```

Initially a given set of numbers (array a) is given as input to the algorithm and the algorithm returns another set of numbers (array c) as output. The algorithm continues execution until the entire set of numbers is sorted. In our algorithm we have used scanning alternately from both ends. The end from which scanning is to be done is decided by using modulus operator function. We have used a simple even odd logic for deciding the end from which scanning is done. By adding a new counter variable and incrementing it each time when scanning from one end is completed and then applying the modulus operator on that counter. If the modulus condition is satisfied then the scanning is done from the last element otherwise from first elements. Our algorithm uses the first element in the given list as the key of a new list called the sub-list b. Then the remaining elements of the original list a is scanned, and every time an element is found which is greater than the last element of the sub-list b, it is appended to the sub-list b, and removed from the original list a. After extracting a sorted sub-list b from the original list, putting that list aside, we then extract another sub-list in the same manner. Then the two sub-lists are merged. The process of extracting a sub-list and merging with the previous sub-list is continued until the original list is exhausted. If the data is already sorted it gives theoretical as well as computational time complexity to be O(n).

### IV. Improving Merge Procedure for Patterns

The complexity of mergersort algorithm is O(nlog(n)). However it includes only the number of comparisons among the elements being sorted. In this paper we focus on some other factors that are not ignorable. For a single processor based system the key points, which have drawn attention of most researchers, is to improve the performance of the merge sort algorithm which mainly includes reducing the number of comparisons and cutting down the size of required auxiliary array. Earlier top down merge sort proposed a method to cut the auxiliary array down to half while natural merge-sort deals with reducing some condition checking in loops. However, to the best of our knowledge, little attention is given to reduce the number of recursive function calls, which certainly adds some overhead on the performance of merge sort. In this paper we first studied the earlier existing merging approaches which were used in top down merge sort and natural merge sort and then proposed a new improved merging procedure. The new proposed merge sort reduces the number of merge function calls and hence performs better.

#### A. Reducing Auxiliary Memory Size

We noticed that it is not necessary to copy the second half of array a to the auxiliary array b. Since, 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. Therefore, it cuts the required auxiliary array size as well as the copy operations to half of that needed in the basic approach.

#### B. Reducing the Loop Condition Checking

We also noticed that in top down merge sort [8] the second while-loop of the merging algorithm checks whether any of the two lists are ended. However, both the list will never end at the same time. Hence, checking only the list that will end earlier is sufficient. This cuts down almost half of the CPU time which is spent in checking the while loop condition. The improved algorithm, along with the improvement done earlier now looks like this
Algorithm natural_merge_sort(a, low, mid, high)
1. Input a
   /*a is the given set of numbers */
2. Input low, mid, high
   /*low represents the first element position in the given list which is to be merged. Mid represents the middle element at which the split is to be performed. High represents the last element in the given list.*/
3. if (a[mid] > a[high])
4.   i=0; j=low;
5. while (j<=mid)
6.     b[i]=a[j]; i=i+1; j = j+1;
7. end while
8. i=0; k=low;
9. while (j<=high)
10. // no need to check whether the left list is finished
11.    if (b[i]<=a[j])
12.       a[k]=b[i]; k = k+1; i = i+1;
13.    else
14.       a[k] = a[j]; k = k+1; j = j+1;
15. end while
16. else
17.    while (k < j)
18.      a[k]=b[i]; k = k+1; i = i+1;
19. end while
20. i=0; j=mid; k=0;
21. while (j<=high)
22.    if (b[mid-1]>a[high])
23.       while(j<=high)
24.         if (b[k]<a[j])
25.            a[i]=b[k]; i++; k++;
26.         else
27.            a[i]=a[j]; i++; j++;
28.       end while
29.     else
30.       while(k<=mid-1)
31.         if (b[k]<a[j])
32.            a[i]=b[k]; i++; k++;
33.         else
34.            a[i]=a[j]; i++; j++;
35.       end while
36.    end if
37. while(j<high)
38.      a[i]=a[j]; j++; i++;
39. end while

Algorithm merge_pool(a, b, low, mid, high) {
1. Input a
2. /*a is the given set of numbers */
3. Input b
4. /*b is the given set of sub list which is to be merged with original list*/
5. Input low, mid, high
6. /*low represents the first element position in the given list a which is to be merged. Mid represents the middle element at which the split is to be performed. High represents the last element in the given list a.*/
7. if(mid==0)
8. //Storing Initial elements into array
9. return;
10. end if
11. if(a[mid]<a[mid])
12. return;
13. end if
14. int i,j,k=0;
15. for(i=0;i<mid;i++)
16.     b[i]=a[i];
17. end for
18. i=0; j=mid; k=0;
19. while (j<=high)
20.      if(b[k]<a[j])
21.        a[i]=b[k];  i++;  k++;
22.      else
23.        a[i]=a[j];  i++;  j++;
24.      end while
25. while(k<=mid-1)
26.     if (b[k]<a[j])
27.        a[i]=b[k];  i++;  k++;
28.     else
29.        a[i]=a[j];  i++;  j++;
30.     end while
31.     while(k<mid-1)
32.     if (b[k]<a[j])
33.        a[i]=b[k];  i++;  k++;
34.     else
35.        a[i]=a[j];  i++;  j++;
36.     end while
37. end if
38. while(j<high)
39.     a[i]=a[j];  j++;  i++;
40. end while

B. Final Modified Merging Algorithm
We have modified the merge procedure even further by adding two more additional condition checks which reduces the overall execution time as well as the time spent in copying the elements. Let the two sub lists be s1 and s2 which are to be merged, then if the last element of s1 is lesser than the first element of s2 then we can directly append s2 to s1 without any more condition checking. We have introduced a new condition check. If the last element of s1 is greater than the last element of s2 then definitely s2 will end earlier as compared to s1. So there is no need to check whether s1 has already ended or not. By introducing these two additional checks merging time can be reduced more effectively.

V. Experimental Results and Analysis
For performance measurement, we have executed our proposed advanced pattern based merge sort algorithm and the other existing merge sort algorithms namely bottom up merge sort[11] algorithm and heuristic merge sort[12] algorithm on a PC having Intel Core 2 Duo I CPU 1.8 GHz and 3.00 GB of RAM. For testing our proposed algorithm we have implemented our algorithm in JAVA and used timer function in order to calculate the running or execution time required for sorting of numbers. The time unit used is nano seconds. We have run the simulated algorithms on same randomly generated Data sets of different sizes on the two merge sort techniques. We have generated eight different databases each containing ten different data sets of same size and ran the
algorithms on them. After running the algorithm on different datasets of a database we have taken the average execution time needed to sort the sets. The average execution time required to sort the records is given in Table 1.

Table 1: Average Execution Time of Various Merge Sort Algorithms

<table>
<thead>
<tr>
<th>Database Size</th>
<th>Bottom-up Merge Sort (time in nano seconds)</th>
<th>Heuristic Pattern Sort (time in nano seconds)</th>
<th>Pattern Based Merge Sort with Pooling (time in nano seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>52860</td>
<td>24633.8</td>
<td>16679.1</td>
</tr>
<tr>
<td>100</td>
<td>140618</td>
<td>630681.3</td>
<td>170846.1</td>
</tr>
<tr>
<td>1000</td>
<td>2321744</td>
<td>6423269</td>
<td>261366.2</td>
</tr>
<tr>
<td>10000</td>
<td>117372283</td>
<td>383049384</td>
<td>29798303</td>
</tr>
<tr>
<td>30000</td>
<td>981015994</td>
<td>2678122785</td>
<td>130561310</td>
</tr>
<tr>
<td>50000</td>
<td>2650592882</td>
<td>7100290014</td>
<td>275109618</td>
</tr>
<tr>
<td>70000</td>
<td>5342830896</td>
<td>13690772148</td>
<td>464356275</td>
</tr>
<tr>
<td>100000</td>
<td>10539018286</td>
<td>27848741279</td>
<td>794770350</td>
</tr>
</tbody>
</table>

Finally, it is observed that our algorithm performs better as compared to other existing mergesort techniques. Moreover, it was also noticed that when larger datasets are given as input, the algorithm performs better as compared to other existing algorithms. This also advocates for the proposed method.

VI. Conclusion

In this paper, we have presented a new advanced pattern based merge-sort technique which targets larger pattern of successive sorted numbers. It greatly enhances the performance of merging by replacing small list element merging with larger chunk merging. We have also studied some of the approaches which were made earlier in order to optimize the merging procedure and found some flaws which may reduce the actual execution time. The technique of Memory Pooling greatly enhances the speed with which merging is done and thereby reduces the execution time practically.

Though the computational complexity increases during the first phase because of increase in the number of comparisons while extracting the sub lists, but it surely runs faster during the second phase when merging of sub lists is done and this compensates for increase in the complexity during the first phase. Since larger numbers of sub lists are already sorted the number of comparisons required during merging as well as the number of merges function call decreases. Hence our algorithm performs better and the experimental results also support this claim.

References

Manas Ranjan Kabat received his M.Tech in Machine Intelligence from Bengal Engineering & Science University. He is also a Ph.D Holder. He is having 11 years of teaching experience. He is pursuing research from last 8 years. He is currently working as a Senior Lecturer in Veer Surendra Sai University of Technology, Burla. His research areas include Algorithms, Computer Networking, Real Time Systems and Internet and Web Technology. He is the author of various papers which are published in various journals, 5 papers in international journals, 15 research papers and attended 10 International Conferences.