Optimal Time and Space Sorting Algorithm on Hierarchical Memory with Block Transfer

Mirza Abdulla, AMA International University

***Abstract*— Sorting is one of most studied problems due to the importance of rearranging data to provide efficient retrieval of required information. The most widely used mechanism for the study of the performance of algorithms is the Rand Access Machine (RAM) model. The model is simple and provides a good tool for the analysis of algorithms. However, computers a have a number of levels of memory hierarchy that necessitates incorporating the idea of hierarchical memory. In this paper, we use the Hierarchical memory with block transfer model which is a RAM but access to a block of locations ending at location x takes x^{α}, for constant $0 < \alpha < 1$, for the last element in the block plus one unit of cost per adjacent items. We provide an asymptotically optimal space and time sorting technique for this problem.**

***Index Term*— Memory Hierarchy, merge sort, time complexity**

I. Introduction

Merge sort is a simple yet efficient sorting technique that can be implemented recursively or bottom up. In the recursive implementation, we sort the first half of the data and the second half of the data separately and then they sorted sequences are merged. On the other hand, in bottom up implementation of merge sort every two odd, even index elements in the sequence form a list and the lists of two elements each are sorted. After the first pass we end up with lists of 4 elements each. Again these lists are sorted and so on until we end up with a single sorted list.

Such an activity is quite efficient if the data can fit in its entirety in main memory. However, with today's heavy use of data, most lists are much larger than can be placed in their entirety in main memory all at once. Rather, data are brought into main memory in blocks that can fit in main memory and are rearranged. Such a task necessarily requires repeated transfer of data from secondary memory to primary memory. Data transfer from higher levels to lower levels of memory is time consuming and is of no direct benefit to the sorting algorithm performance. Indeed, to account for this time consumption we can charge a cost on data that need to be transferred to lower memory levels. However, this charge should also take into account the fact that once the first data item is accessed on slower memory, bringing other "adjacent" data is relatively cheaper and can be used to offset the high cost of bringing the first data item to faster memory. Therefore, we can base our charge on the access of access to a block of data to be brought to faster memory the cost of accessing the block and the length of the block. Indeed this what [2] used as to model a RAM with memory hierarchy.

Our task will be to find efficient implementation of sorting on such a machine. [2] provided an O($n$lg$n$ llg$n$) algorithm that sort n data items in O(n) space, as well as another techniques which sorts n data items in O($n$lg$n$) time, but uses O($n$llg$n$) space. Thus they provided a sorting technique that performs asymptotically optimal in space but not in time, while the second is optimal in time but not space. In this paper we will provide an asymptotically optimal time and space algorithm for this problem.

II. Discussion

A. Model of Computation.

The RAM model is the most widely used model for measuring computational complexity of algorithms. It is a simple model where the program and data are stored in "registers", and each "elementary" operation executes in a constant time. The running time of an algorithm on an instance of the input is the number of operations executed in terms of the size of the instance. Similarly the space used is the amount of extra space in terms of the size of the input. However, since the assumption is that all data is stored in registers makes it unrealistic for large data sets which for the most of data don't reside in registers or primary memory. Indeed, in modern computers most of the data resides on secondary memory and is brought in blocks of suitable size to offset the cost of accessing the secondary memory. To our knowledge, the first to highlight the importance of incorporating the access cost to higher memory was that of [3] which studied the effect of having 2 levels of memory on sorting and relating, [1] which studied the effect of having multiples of levels of memory, and [2] which studied the effect of having multiple levels of memory that can be accessed at an increasing cost but with block transfer allowed. The model of [2] is defined below.

DEFINITION [2]. The Hierarchical Memory Machine (HMM) is a Random Access Machine (RAM) operating under the following conditions:

I. The access cost to any location, *i*, costs *f(i)*, where *f* is a non-decreasing function.

II. The cost of moving a block of length *l* in locations [x-*l*, *x*] to locations [*y*-*l*, *y*] costs *f*(*x*)+*f*(*y*)+*l*.

The problem of sorting was in the center of the works of [2], however, they report two solutions; one that provides an optimal time performance but non optimal space usage, and the other uses space optimally but renders a non optimal time performance. In this paper, we shall adopt the model given by [2] and show that their techniques for solving the sorting problem on such a model can be used to provide an optimal solution in both time and space for the sorting problem under the access cost of $f(n) = n^{\alpha}$, $0 < \alpha < 1$.

B. Merge Sort Algorithm

Top down solution.

The merge sort technique in its simplest form works on dividing the input into two halves that are sorted recursively and the two halves are then merged to form a final sorted list. The merging of the two halves is basically performed by:

- Choose the smaller of the first item in the two lists
- Remove it from its pile, thereby exposing another element to be the smallest of that list.
- Place the chosen element onto the output array.
- Repeatedly perform basic steps until one of the lists is empty, in which case we just take the remaining input items in the non empty list and place onto the output array.

Indeed, the pseudo code for the merge operation can be written as:

MERGE (A, p, q, r)
  $l_1 \leftarrow q - p + 1$; //Length of the left list
  $l_2 \leftarrow r - q$;//Length of the right list
//Copy the data to arrays L[1 . . l_1 + 1] and R[1 . . l_2 + 1]
  FOR $i \leftarrow 1$ **TO** l_1
    L[i] ← A[$p + i - 1$]
  FOR $j \leftarrow 1$ **TO** l_2
    R[j] ← A[$q + j$]
  L[$l_1 + 1$] ← ∞;
  R[$l_2 + 1$] ← ∞;
  $i \leftarrow 1$;
  $j \leftarrow 1$;
  FOR $k \leftarrow p$ **TO** r
    IF L[i] ≤ R[j]
      THEN A[k] ← L[i]
        $i \leftarrow i + 1$
      ELSE A[k] ← R[j]
        $j \leftarrow j + 1$

Each of these basic steps in the merging process can be performed in a constant time, resulting in at least one of the halves becoming one item less and the final array is one item more. Since there are a total of n elements to be merged, it follows that the merging process would be performed in at most $O(n)$ time.

It follows therefore, that the total time cost of sorting n elements can be obtained from the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$T(1) = 1$$

Which can be solved as follows:

$$T(n) - 2T\left(\frac{n}{2}\right) = n$$
$$2T(n/2) - 4T\left(\frac{n}{4}\right) = n$$
$$4T(n/4) - 8T\left(\frac{n}{8}\right) = n$$
….
….
$$\frac{n}{2}T(2) - nT(1) = n$$

And summing all of these equations we get:

$$T(n) - n \leq nlgn$$

Which implies that $T(n) = \theta(nlgn)$. Moreover, the space usage is $S(n) = \theta(n)$.

Bottom Up Merge Sort

The top down technique is recursive and the recursion stack can have a depth of size $O(\lg n)$. To avoid recursion, we can perform merge sort bottom up. The steps are as follows:

- The ***bottom-up*** **mergesort** algorithm first merges pairs of adjacent arrays of 1 elements
- *Then* merges pairs of adjacent arrays of *2* elements
- And *next* merges pairs of adjacent arrays of *4* elements
- And so on.... Until the whole array is merged

The pseudo code is as follows:

```
sort(a[n]){
    for ( int w = 1; w < n; w = 2*w ) {
   // Combine pairs of array a of width "w"
       for ( int i = 0; i < n; i = i + 2*w ) {
              left = i;
              middle = i + w;
              right  = i + 2*w;
              merge( a, left, middle, right );
             }
         }
      }
```

On each iteration of the inner for loop, the merge function, and as mentioned earlier, would require time of the order of the size of the lists to be merged. Since the merge operation is performed on different items on each iteration of the inner loop, it follows that cost of merge on each element in the array involved in the merge operation is $O(1)$. It follows, therefore, that the overall cost of the inner for loop on each iteration of the outer for loop is at most $O(n)$ time.

On each iteration of the outer loop the size of the variable ***w*** is doubled. The outer loop terminates when ***w*** is no longer less than the number of elements in the array. The number of iterations is, thus, at most, ***k*** where$2^k \geq n$. It follows, therefore, that the outer loop is performed at most $O(log_2 n)$ times, from which it follows that the overall cost of the mergesort algorithm is $O(n\ log_2 n)$time.

Sorting data under non-uniform access cost was studied by [2] for a variety of access costs. They gave optimal time and space solutions when the access cost is log$n$, n/log$n$, and n. However, for the case when the access cost to location n is $n^{\alpha}, 0 < \alpha < 1$, they gave two algorithms. One the algorithms is asymptotically optimal in time but requires $O(nloglogn)$ space, which we shall refer to as **SORT1**

algorithm. The other one runs in $O(nlgn\ loglogn)$time but uses $\theta(n)$ space, and we shall refer to it as **SORT2** algorithm. More formally:

Theorem 1. [2]. Given *n* elements in the memory of HMM machine with block transfer running under access cost of $f(n) = n^{\alpha}$, for constant $0 < \alpha < 1$:

1. Reading can be performed in $\theta(nloglogn)$ time.
2. Merging two lists of size $O(n)$ each can be performed in $\theta(nloglogn)$ time.
3. Sorting of can be performed in $O(n\ lgn\ loglogn)$time and $\theta(n)$ space.
4. Sorting can be performed in $\theta(nlogn)$ time and $O(nloglogn)$space.

III. The Optimal Time and Space Algorithm

A. Algorithm

In the following we will show that with simple modifications it would have been possible for [2] to provide a solution that is optimal in both space and time, by using the SORT1 algorithm coupled by bottom up merge sort. The pseudo code is as follows, assuming that the input array is initially placed in locations 1 to *n* in the memory hierarchy.

Sort3(a, n){

Step	statement
1	**Move the array a from locations 1..*n* to locations 5*n*+1..6*n*;**
2	***w* = *n*/loglog*n*;**
3	**for (*i* = 5*n*+1; *i* < *n*; *i* = *i*+*w*) {** **a) Move elements a[*i*.. *i*-1+*w*]) of the array to locations 1..*w*;** **b) Apply SORT1 to sort these elements;** **c) Move the sorted elements back to locations *i*.. *i*-1+*w*;** **}**
4	**for (*i* = *w*; *i* < *n*; *i* = 2**i*) {** **// Combine pairs of array a** **// initially of width "*w*"**

5	**for (*k* = 5n+1;*k*<*n*; *k* = *k*+2**i*) {**
	a) *left* = *k*; // mark the beginning of the first list
	b) *middle* = *k* + *i*; // mark the end of the first list
	c) *right* = *k* + 2i*; // mark the end of the second list**
	d) merge(a, *left*, *middle*, *right*);
	}

6	**Move data from locations 5*n*+1..6*n* to locations 1..*n*.** **}**

B. Analysis

The algorithm first moves all that data to start from location 5n+1. Such a move costs at most $O(n)$ time, since the cost is dominated by the size of data moved. The purpose of the move is to allow enough extra space for any sorting or merging of sub lists.

Step 3 brings sub-lists of size *n*/loglog*n* each to faster memory for sorting using SORT1 algorithm. The time to perform the sort is) $O\left(\left(\frac{n}{loglogn}\right) log\left(\frac{n}{loglogn}\right)\right) = O(\frac{nlogn}{loglogn})$. The sorted list is then placed back to its original space, which again can be performed in *n*/loglog*n* time. Thus each iteration of step 3 can be performed in at most $O\left(\frac{nlogn}{loglogn} + nloglogn + nloglogn\right) = O(\frac{nlogn}{loglogn})$ time, and at most $O(n)$space.

Step 3 would be iterated upon until all the data is exhausted. Thus, the loop will iterate for $\frac{n}{\frac{n}{loglogn}} = loglogn$ times. It follows, therefore, that step3 would require at most$O\left(\frac{nlogn}{loglogn}\ loglogn\right) = O(nlogn)$ time, and $O(n)$ space.

The rest of the steps perform merge sort on the sorted sub-lists of step 3, by taking each adjacent sub-lists and merging them to obtain a new sorted list of twice the size of the original lists. The actual merging of sub lists is performed in step 5. The merging of sub lists according to Theorem 1.b costs $O(loglogn)$ per item involved in the merge. Thus, the overall cost of step 5 during each iteration of the for loop of step 4 is at most $O(nloglogn)$ time, and of course $O(n)$ space.

The for loop of step 4 will iterate on variable ***i*** starting from a value of $\frac{n}{loglogn} = 2^{logn-loglogn}$ and doubles before the start of a new iteration until it reaches the value $n = 2^{logn}$. The number of iterations is at most $logloglogn$ iterations.

It follows, therefore, that the bottom up merge sort process in steps 4 and 5, can be performed in at most $O(n\ loglogn\ logloglogn)$ time and $O(n)$ space.

Finally, Step 6 is just a move operation to place the sorted data sequence back to its original locations at the start of the sort algorithm. This operation can be performed in O(n) time and space.

Thus we have:

Theorem 2. ***n*** data items on HMM with block transfer running under access cost of $f(n) = n^{\alpha}$, for constant $0 < \alpha < 1$, can be sorted in $\theta(nlogn)$time and $\theta(n)$space.

IV. Conclusion

In this paper we showed that when we apply memory hierarchy constraints on the Random Access Machine model through the HMM with block transfer model similar to that of [2], we can still get optimal time and space sorting algorithm, matching those on the RAM model. This is a good indicator that the sorting problem possesses a good degree of spatial and temporal locality to offset the cost of data transfer between various levels of the memory hierarchy, even for access cost as high as $\boldsymbol{x^{\alpha}}$, for constant $\mathbf{0} < \alpha < 1$ to access memory ***x*** when block transfer is allowed.

REFERENCES

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, A Model for Hierarchiecal Memory, Proceedings of the 19th Annual ACM Symposium on Theory of Computing, ACM Press (1987), 305–314.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, Hierarchiecal Memory with Block Transfer, Proceedings of the 28th Annual IEEE Symposium on Foundations of Computing, IEEE Computer Society Press (1987), 204–216.

[3] Aggarwal and J. S. Vitter."The Input/Output complexity of sorting and related problems". Communications of the ACM, 31(9):1116-1127, 1988.

[4] J. S. Vitter, External Memory Algorithms and Data Structures: Dealing with MASSIVE Data, ACM Computing Surveys 33,2 (2001), 209–271.

[5] Aho, A. V.; Hopcroft, J. E.; and Ullmann, J. D. "Data Structures and Algorithms". Reading, MA: Addison-Wesley, pp. 369-374, 1987.

[6] Folk, Michael J.; Zoellick, Bill (1992), File Structures (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4

[7] Hong Jia-Wei, H. T. Kung, I/O complexity: "The red-blue pebble game", Proceedings of the thirteenth annual ACM symposium on Theory of computing, 1981

[8] M. Mirza, "Data Structures and Algorithms for Hierarchical Memory Machines", Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, 1990.

[9] M. H. Nodine and J. S. Vitter. "Paradigms for optimal sorting with multiple disks.", Proc. of the 26th Hawaii Int. Conf. on Systems Sciences, 1993.

[10] J. D. Ullman and M. Yannakakis. "The input/output complexity of transitive closure", Annals of Mathematics and Artificial Intellegence, 1991.

[11] J. S. Vitter. "Efficient memory access in large-scale computation" (invited paper), Symposium on Theoretical Aspects of Computer Science, LNCS 480, 1991.

[12] J. S. Vitter and E. A. M. Shriver. "Algorithms for parallel memory, I: Two-level memories", Algorithmica, 1994.

[13] I. Flores, "Analysis of Internal Computer Sorting", ACM, vol. 7, no. 4, **(1960)**, pp. 389- 409.

[14] G. Franceschini and V. Geffert, "An In-Place Sorting with O(n log n) Comparisons and O(n) Moves", Proceedings of 44th Annual IEEE Symposium on Foundations of Computer Science, **(2003)**, pp. 242-250.

[15] D. Knuth, "The Art of Computer programming Sorting and Searching", 2nd edition, Addison-Wesley, vol. 3, **(1998)**.

[16] A. D. Mishra and D. Garg, "Selection of the best sorting algorithm", International Journal of Intelligent Information Processing, vol. 2, no. 2, **(2008)** July-December, pp. 363-368.

[17] E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms, Galgotia Publications.

[18] T. H. Coreman, C. E. Leierson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, MIT Press.

[19] J. Alnihoud and R. Mansi, "An Enhancement of Major Sorting Algorithms", International Arab Journal of Information Technology, vol. 7, no. 1, **(2010)**, pp. 55-62.

[20] E. Kapur, P. Kumar and S. Gupta, "Proposal of a two way sorting algorithm and performance comparison with existing algorithms", International Journal of Computer Science, Engineering and Applications (IJCSEA), vol. 2, no. 3, **(2012)**, pp. 61-78.

[21] S. Lipschutz, "Theory and Problems of Data Structure", McGraw Hill Book Company.