Algorithms for External Memory Sorting

Milind Gokhale

mgokhale@indiana.edu

December 10, 2014

1. Abstract

Sorting has been one of the fundamental operations for processing data in any database for a very long time. With the increasing amount of data, the problem of sorting data within a stipulated time has become one of the major factors for almost all applications. We study two papers on algorithms for external memory (EM) sorting and describe a couple of algorithms with good I/O complexity.

2. Introduction

With the increasing complexity of applications and the humongous data associated with the applications, many required operations like scanning, sorting, searching, and query output can take longer time. Even with the computing power of advanced processors, the internal memory is limited and the frequent movement of data has magnified the external factor of data movement to be of much higher importance along with time and space complexity in algorithms.

The problem of how to sort efficiently has strong practical and theoretical merit and has motivated many studies in the analysis of algorithms and computational complexity. Studies [1] confirm that sorting continues to account for roughly one-fourth of all computer cycles. Much of those resources are consumed by external sorts, in which the file is too large to fit in internal memory and must reside in secondary storage. The bottleneck in external sorting is the time for input/output (I/O) operations between internal memory and secondary storage. So external sorting is required when the data being sorted does not fit into the internal memory of the computing device and instead must reside in the slower external memory. We study two papers which talk about sorting in external memory (EM) and present optimal algorithms for EM sorting.

Paper 1: *The Input/output Complexity of Sorting and Related Problems*, authored by Alok Aggarwal and Jeffrey Scott Vitter and published in the journal Communications of the ACM Volume 31, issue number 9 in 1988.
Area: Algorithms and Data Structures

Paper 2: *Algorithms and Data Structures for External Memory* written by Jeffrey Scott Vitter and published in the journal Foundations and Trends in Theoretical Computer Science Volume 2, issue number 4 Chapter 5 in 2006.
Area: Algorithms and Data Structures, External Memory Algorithms.

In section 3 we describe a couple of EM sorting algorithms viz. Distribution Sort and Merge Sort algorithms, in section 4 we compare the way the chosen papers focus on the different aspects of the algorithms with a common goal to reduce the I/O complexity.

3. Techniques/Algorithms

3.1. External Sorting

Even today, Sorting accounts for a significant percentage of computer operations and so is an important paradigm in the design of efficient External Memory algorithms. Both the papers present variants of merge-sort and distribution-sort algorithms as the optimal algorithms for external sorting. They also characterize the I/O complexity of the sorting methods in a theorem as below

The average-case and worst-case number of I/Os required for sorting N = nB data items using D disks is

$$Sort\ (N) = Theta\ (\frac{n}{D} \log_m n)$$

The algorithms are based upon the generic sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file. Distribution sort and merge sort using the randomized cycling and simple randomized merge sort are the methods of choice for external sorting.

3.2. Parameters:

- N - # records to sort
- M - # records that can fit into internal memory
- B - # records that can be transferred in a single block
- n – N/B - # blocks of records to sort
- m – M/B - # blocks of records that can fit into internal memory
- D - # independent disk drives
- P - # of CPUs

3.3. Distribution Sort

In this algorithm the elements to be sorted are partitioned into subsets also called as buckets. The important property of the partitioning is that all the items in one bucket precede all the items in the next bucket. The sorting is completed by recursively sorting the individual buckets and concatenating them to form a single fully sorted list.

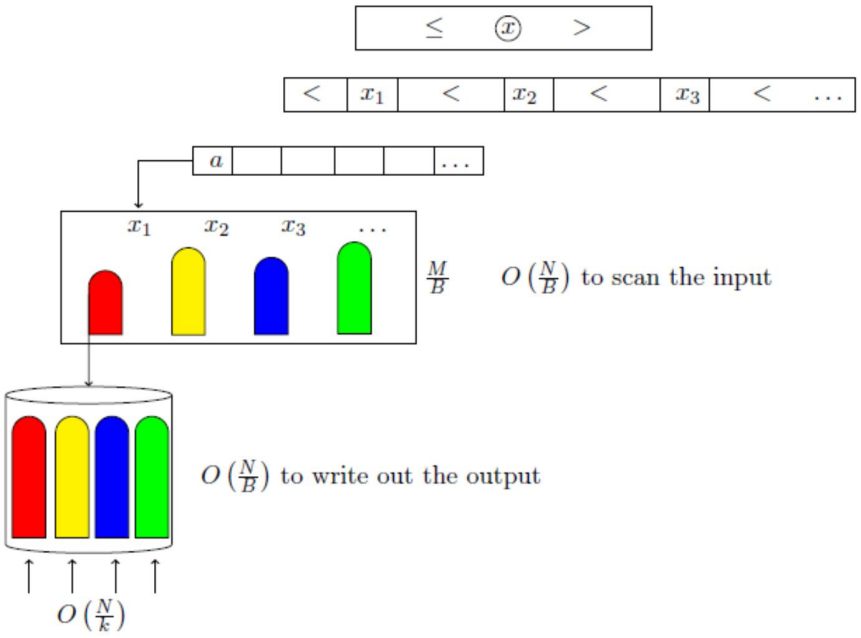

Figure 1: The sorting process [3]

3.3.1. The Sorting Process

The elements stored in external memory are read and partitioned into S buckets. To partition the elements S-1 partitioning elements are found first. The input is partitioned into S buckets of roughly equal size such that any element in bucket i is greater than or equal to the (i-1)-th partitioning element and lesser than or equal to the i-th partitioning element [2]. This process is repeated till the bucket size is reduced to M.

Partitioning Invariant: $(i-1)th\ partioning\ element < i < ith\ partitioning\ element$

3.3.2. Finding the partitioning elements

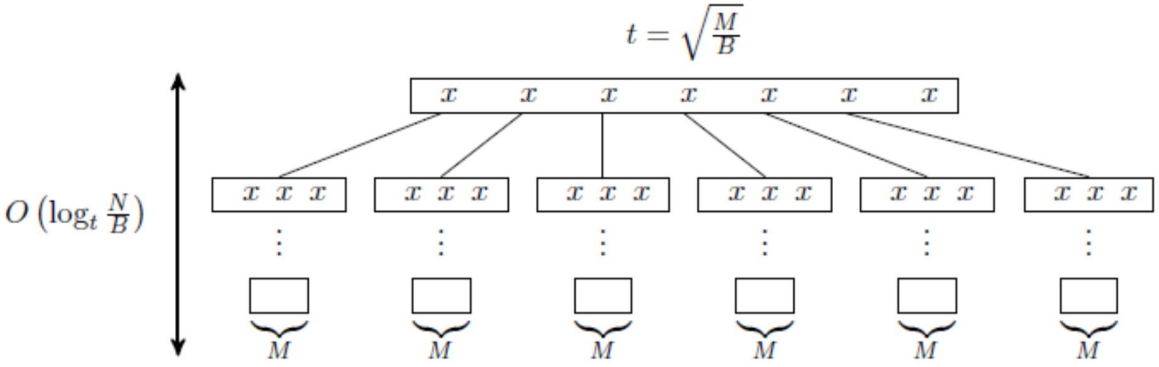

Figure 2: Recursively Distribute Items into S buckets until each bucket contains at most M elements. [2]

While choosing S-1 partitioning elements into equal size buckets, the bucket size decreases from one recursion level to next by a relative factor of Θ(S). So there are $O(\log_S n)$ levels of recursion. During each level of recursion, data is scanned into internal memory and partitioned into S buckets. When the buffer is full, another buffer is used to store the next

set of incoming items for the bucket. So the maximum number of buckets is Θ(M/B) = Θ(m), and the thus the number of levels of recursion is $\Theta(\log_m n)$.

3.3.3. Analysis of Distribution Sort

The I/O complexity of the above algorithm can be described by the below recursion expression.

$$Q(N) = S\,Q\left(\frac{N}{S}\right) + O\left(\frac{N}{B}\right) \qquad if\ N > M$$

$$= O\left(\frac{N}{B}\right) \qquad if\ N \leq M$$

Therefore,

$$Q(N) = O\left(\frac{N}{B}\left(1 + \log_S \frac{N}{M}\right)\right)$$

If S = $\Theta\left(\frac{M}{B}\right)$,

Then,

$$Q(N) = O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}} \frac{N}{B}\right)\right)$$

$$= Sort(N)$$

Thus the number of partitioning elements S is $\Theta\left(\min\left\{m, \frac{n}{m}\right\}\right)$.

However since there is no algorithm to find the m partitioning elements using O(n) I/O operations, hence we find $\sqrt{m}$ partitioning elements using O(n) I/O operations.

Now, $O\left(n\ \log_{\sqrt{m}} n\right) = O(n\ \log_m n)$

The number of I/O operations to implement the distribution sort is therefore,

$$O\left(n\ \log_{\sqrt{m}} n\right) \approx O(n\ \log_m n) = O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$$

3.3.4. Algorithm to identify partitioning elements

We see an algorithm to identify partitioning elements or pivots deterministically using O(n) I/O complexity. We apply the selection algorithm to find the ith element in sorted order.

1. First, the array of given data items is split into pieces of size 5 each.
2. Then we find the median of each piece of 5 elements each.

3. We recursively select the median of N/5 selected elements.
4. Distribute the elements into two lists using computed median.
5. Then we recursively select in one of the two lists.

Step 2 and 4 are performed in O (N/B) I/Os. While step 5 recurses on at most $\frac{7}{10}N$ elements. Thus algorithm to identify partitioning elements can be given by the below recursion:

$$T(N) = O\left(\frac{N}{B}\right) + T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) = O\left(\frac{N}{B}\right) I/Os$$

3.4. Merge Sort

Merge-sort algorithm for external memory uses the sort and merge strategy to sort the huge data file on external memory. It sorts chunks that fit in main memory and then merges the sorted chunks into a single larger file. Thus it can be divided into 2 phases – Run formation Phase (Sorting chunks) and Merging Phase.

3.4.1. Run formation

n blocks of data are scanned, one memory load at a time. Each memory load consisting of m blocks is sorted into a single run and is given as output to a series of stripes on the disk. Thus there are N/M or n/m runs each sorted in stripes on the disk.

3.4.2. Merging Phase

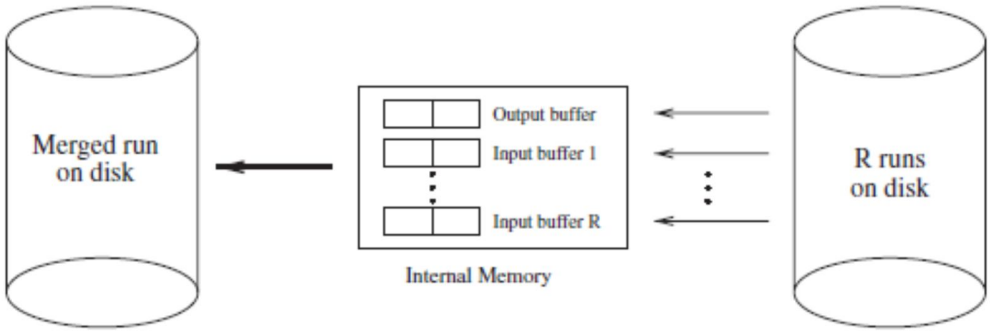

Figure 3: Merge Phase [2]

After initial runs are formed, the merging phase begins where groups of R runs are merged. For each merge, the R runs are scanned and merged in an online manner as they stream through the internal memory. With double buffering for overlap in I/O and computation, maximum R = Θ(m) runs can be merged at a time and hence the number of passes to merge the R runs is $O(\log_m n)$ [2].

3.4.3. Analysis of External Merge-Sort algorithm

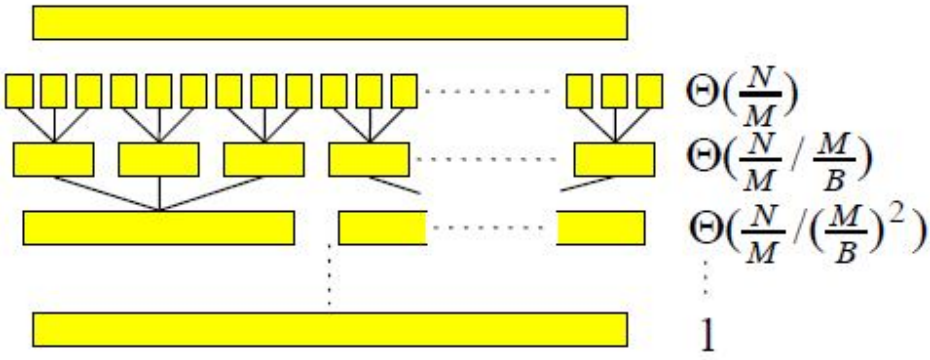

Figure 4: I/O complexity of External merge-sort algorithm [**4**]

The run formation phase which involves creation of N/M or n/m memory sized sorted lists takes place in $O\left(\frac{N}{M}\right)$ I/O operations.

During the merging phase the, R runs are merged together repeatedly. As seen in the I/O complexity diagram (figure 4), it forms a recursion tree with N/M elements at leaves and height of the tree log N/M. Since the problem is divided into M/B parts every time, the I/O complexity of the merging phase becomes

$$O\left(\log_{\frac{M}{B}} \frac{N}{M}\right) phases\ using\ O\left(\frac{N}{B}\right) I/Os\ \ each\ time\ giving$$

$$\boldsymbol{O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right) I/Os}$$

4. Comparison

First paper focuses on establishing tighter upper and lower bounds for the number of I/Os between internal and external memory required for five sorting related problems of sorting, Fast Fourier Transform (FFT), permutation networks, permuting and matrix transposition. On the other hand second paper focuses on the many variants of distribution sort and merge sort algorithms and improvisations taking into consideration the parallel disk model (PDM) with multiple disks. It also analyses the various effects of partial or complete striping on the external sorting algorithms.

4.1. Improvements in External Sorting Algorithms

4.1.1. Load Balancing across multiple disks in Distribution Sort:

In order to meet the I/O bound by the given theorem the buckets at each recursion level must be formed by O(n/D) I/Os. For a single disk, D = 1, Each input I/O step and each output I/O step during the bucket formation stage in distribution sort must involve on the average Θ(D) blocks. So the challenge in distribution sort is to output the blocks of the buckets to the

disks in an online manner and achieve a global load balance by the end of the partitioning, so that the bucket can be input efficiently during the next level of the recursion.

Using Partial Striping can also be effective by reducing the amount of information stored in the internal memory. [2] In this the disks are grouped into clusters of size C and data is output in logical blocks of size CB. Choosing $C = \sqrt{D}$ will although not improve the sorting time by more than a constant factor, however it will give better I/O complexity than Full Striping.

4.1.2. Randomized Cycling Distribution Sort

Using the standard striping method, blocks that belong to a given stripe belong to multiple buckets and the buckets will not be striped across the disks. Thus in the output phase, each bucket must keep a track last block output to each disk so that the blocks for the bucket can be linked together which is an overhead. Hence a better approach is to stripe the contents of each bucket across the disks so the input operation can be done in a striped manner and during the output step multiple buckets will be transmitted to the multiple stripes [2].

So the basic loop of distribution sort algorithm is same as before, to stream the data items through the internal memory and partition them into S buckets, and the blocks for each individual bucket will reside on disks in stripes. Thus each block has predefined disk where it must be output. If we choose the normal round-robin method of ordering the disks for striping, then the blocks of different buckets can collide. To solve this problem, randomized cycling is used instead of round-robin. For each of the S buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1,2,\ldots,D\}$. The S random permutations are also chosen independently from the S-1 buckets meaning each bucket has its own random permutation ordering chosen independently from those of the other S-1 buckets [2]. The resulting sorting algorithm called the randomized cycling distribution sort (RCD), achieves optimal sorting bound as in the theorem 1 with extremely small constant factors.

4.1.3. Simple randomized merge sort

Striping technique can also be used in merge sort to stripe runs across multiple disks. However disk striping uses too much of internal memory to cache blocks not yet merged and thus effective order of the merge is reduced to $R = \Theta(m/D)$ which gives a non-optimal result. The Simple Randomized merge sort uses much less space in internal memory for caching blocks and thus allows run R to be much larger. Each run is striped across the disks, but with a random starting point. During the merging process, the next block needed from each disk is input into internal memory, and if the memory is full, then least needed blocks are flushed back to disk. Simple randomized merge sort is not optimal for some parameter values, but it outperforms disk striping.

5. Conclusion

Although both EM sorting algorithms, distribution sort and merge sort have almost similar I/O complexity, there are differences in the way they work. Distribution sort work by partitioning the unsorted values into smaller "buckets" that can be sorted in main memory,

while the Merge sort algorithm works on the principle of merging sorted sub-lists. The former paper by J. S. Vitter in 1988 claims that the standard merge sorting algorithm is an optimal external sorting method while the later paper in 2006 points out that the Distribution sort algorithm has an advantage over the merge approach in a way that they typically make better use of lower levels of cache in the memory hierarchy of real systems, however, merge approaches can take advantage of the replacement selection to start off with larger run sizes.

References

1. Aggarwal, Alok, and Jeffrey S Vitter. "The Input/output Complexity of Sorting and Related Problems." Communications of the ACM, Vol 31, no. 9, pp 1116-127, 1988.
2. J. S. Vitter, "Algorithms and Data Structures for External Memory", Foundation and Trends® in Theoretical Computer Science, vol 2, no 4, pp 305–474, 2006.
3. Sitchinava, Nodari. "EM Model." Lecture, Algorithms for Memory Hierarchies, October 17, 2012.
4. Arge, Lars. "I/O-Algorithms." Lecture, Spring 2009, Aarhus, January, 2009.