Lecture Notes: External Memory Model and Sorting

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

1 The Computation Model

You are perhaps familiar with the complexities of many algorithms in internal memory. For example, it is well-known that N numbers can be sorted with $O(N \log N)$ time in the *RAM model*. What this statement says exactly is that, there is an algorithm able to solve the sorting problem by performing $O(N \log N)$ "basic operations". In particular, each basic operation either performs some "standard" CPU work (e.g., $+$, $-$, $*$, $/$, comparison, taking the AND/OR/XOR of two words, etc.) or accesses a memory location.

Many applications in practice need to deal with datasets that are too large to fit in memory. While it is true that the memory capacity of a computer has been increasing rapidly, dataset sizes have exploded at an even greater pace, such that it is increasingly unrealistic to hope that someday we could run all the applications entirely in memory. In reality, data still need to be stored in an external device, typically, a hard disk. An algorithm in such environments would need to perform many disk I/Os to move data between the memory and the disk. Since an I/O is rather expensive (at the order of 1-10 milliseconds), the overall execution cost may be far dominated by the I/O overhead.

This phenomenon has triggered extensive research in the past three decades on algorithms in the *external memory* (EM) model, which was proposed in 1988 [1], and has been very successful in capturing the characteristics of I/O-bound algorithms. A computer of this model is equipped with a memory of M words, and a disk of an unbounded size. The disk has been formatted into disjoint *blocks*, each of which has the length of B words. An I/O either brings a block of data from the disk to the memory, or conversely writes B words in the memory to a disk block. The *space complexity* of a data structure or an algorithm is measured as the number of disk blocks occupied, while the *time complexity* is measured as the number of I/Os performed. CPU calculation can be done only on the data that currently reside in the memory, but any such calculation is charged with no cost. Accessing any data in the memory is also for free.

The value of M is assumed to be at least $2B$, i.e., the memory can be as small as just 2 blocks. However, it is often acceptable to assume $M \geq B^2$, which is known as the *tall cache assumption*. By fitting in some typical values of B in practice, you can convince yourself that a memory with B^2 words is available in almost any reasonable computer nowadays.

For a dataset of N elements[1], the minimum number of blocks required to store all the elements is $\Omega(N/B)$. Therefore, *linear cost* should be understood as $O(N/B)$, as opposed to $O(N)$.

2 External Sort

Let us start with the sorting problem. We are given a set S of N elements in $\mathbb{R}$. At the beginning, these elements are stored in a file of $O(N/B)$ blocks. Eventually we should output a file of $O(N/B)$ blocks where the elements have been properly sorted. Adapting a memory-resident algorithm to sort an external file can easily result in a cost of $O(N \log_2 N)$ I/Os.

[1]Unless otherwise stated, each element is described by a word.

Let us warm up by refreshing our memory about the *external sort* algorithm [1] at the undergraduate level, which solves the problem in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. For simplicity, we will assume $M \geq 3B$.

First, we divide S arbitrarily into $\lceil N/M \rceil$ partitions, each of which has exactly M elements, except possibly the last partition, which may have less elements. Each partition therefore can be stored in at most M/B blocks. We load each partition into memory, sort the elements there (in memory), and write the sorted sequence (at most M/B blocks) back to the disk. We call the above step the *initialization phase*.

Set $m = M/B$. Given an integer $i \geq 0$, we define a *level-i sequence* as a sequence of "partitions" such that all the following conditions hold:

- Each partition is a set of elements in S. All the partitions are mutually disjoint; and their union is S.
- Each partition has $M \cdot (m-1)^i$ elements, except perhaps the last partition which may have less elements.
- The elements in each partition are sorted, and stored in $O(m(m-1)^i)$ blocks.

Note that a level-i sequence has $\lceil N/(M(m-1)^i) \rceil$ partitions. Furthermore, we have obtained the level-0 sequence after the initialization phase.

In general, given a level-i sequence R, we can produce a level-$(i+1)$ sequence in linear I/Os (i.e., $O(N/B)$ I/Os). To do so, divide the partitions of R arbitrarily into groups, each of which has $m-1$ partitions except possibly the last group. For each group, merge the partitions therein into one sequence, with all the elements in the sorted order. This requires reading and writing these elements only once (think: how?). That sequence corresponds to a partition in the level-$(i+1)$ sequence.

Starting from a level-0 sequence, we iteratively compute sequence of higher levels until the number of partitions becomes 1. As the number of partitions decreases by a factor of $\Omega(m)$, except possibly the last iteration. The total number of levels is $O(\log_m(N/M)) = O(\log_{M/B}(N/B))$. The total cost is therefore $O((N/B)\log_{M/B}(N/B))$.

Remark. We leave as an exercise for you to think about what to do if $M = 2B$. But here is a hint: you can still use the above algorithm directly but on a smaller block size.

3 Distribution Sort

External sort takes a bottom-up approach (i.e., iteratively merging partitions into larger ones). Next, we will discuss another algorithm called *distribution sort* that goes top-down. To illustrate the idea, imagine we partition the input S into f subsets $S_1, ..., S_f$ of roughly the same size, such that any element in S_i should be smaller than all elements in S_j, for any $i < j$. Then, we can work on each subset recursively, until the subset is small enough to fit in memory, at which point we simply sort the subset in memory. The core of the algorithm is to obtain $S_1, ..., S_f$ from S efficiently. Intuitively, if f is not too large, this ought to be simpler than sorting S, because we do not care about the ordering of the elements in each S_i.

3.1 *k*-Selection

We will first look at a relevant problem called *k-selection*, where we are given a set S of N elements in $\mathbb{R}$, and want to report the k-th smallest one ($1 \leq k \leq N$). We will discuss the problem in internal memory (specifically, the RAM model) because the EM extension is straightforward. The problem is trivial to solve in $O(N \log N)$ time by sorting. It turns out that there is a clever algorithm [2] to do so in $O(N)$ time, as explained next.

Without loss of generality, we will assume that N is a multiple of 10 (if not, then pad at most 9 dummy elements larger than all the existing ones). Divide S arbitrarily into groups of size 5: $G_1, ..., G_{N/5}$. For each group, collect the median of each group into a set G, which therefore, has size $N/5$. Call each element in G the *representative* of its origin group. So far the cost is $O(N)$.

Next, we find the median g of G. Note that this is another k-selection (with $k = |G|/2$) problem, but on a smaller set, and therefore can be solved recursively in the same way. Now assume that we have already obtained g. In linear time, we divide S into S_1 and S_2 containing all the elements at most and larger than g, respectively. If $|S_1| \geq k$, we recurse by finding the k-th smallest element in S_1, and otherwise, by finding the $(k - |S_1|)$-th smallest in S_2.

Lemma 1. $|S_1| \leq 7N/10$ *and* $|S_2| \leq 7N/10$.

Proof. We will bound only $|S_2|$ because the case with $|S_1|$ is similar. In each group, 2 elements are smaller than the representative. As before g there are $N/10 - 1$ group representatives, we know at least $3(N/10 - 1) + 3 = 3N/10$ elements are at most g, where the term $+3$ counts the elements in the group of g. Hence, $|S_2| \leq 7N/10$. □

We now analyze the running time of the above algorithm. Denote by $F(N)$ the time of solving an input of size N. It is easy to observe that, for N greater than a sufficiently large constant c,

$$F(N) \leq F(N/5) + F(7N/10) + O(N) \tag{1}$$

where $F(N/5)$ is the cost of finding g, and $F(7N/10)$ the cost of solving the problem recursively on either S_1 or S_2. For $N \leq c$, we can solve the problem by simply sorting all elements in $O(c \log c) = O(1)$ time. Solving the recurrence gives $F(N) = O(N)$.

The above algorithm can be easily modified to terminate in $O(N/B)$ I/Os in external memory.

3.2 The f-Splitter Problem

We now turn to another relevant problem called the *f-splitter problem*. Here, the input is a set S of N elements in $\mathbb{R}$. We want to find f *splitters* $p_1, p_2, ..., p_f \in S$ in ascending order such that there are $O(N/f)$ elements in the range $(p_{i-1}, p_i]$, for each $i \in [1, f+1]$, defining dummy splitters $p_0 = -\infty$ and $p_{f+1} = \infty$. We will describe an algorithm proposed in [1] that solves this problem in linear I/Os for $f = \lceil \sqrt{M/B} \rceil$.

Divide S arbitrarily into $\lceil N/M \rceil$ groups $G_1, ..., G_{\lceil N/M \rceil}$, each of which has M elements except possibly the last one. Then, load each G_i into memory, sort it, go through the sorted list in ascending order, and collect one out of every f elements into a set G (i.e., G includes the f-th element of G_i, the $2f$-th, and so on). Those elements are the *representatives* of G_i. In total, G has $O(\frac{M}{f}\frac{N}{M}) = O(N/f)$ elements. Up to now, we have performed only linear I/Os.

Next, for each integer $i \in [1, f]$, pick the $i\lceil \frac{N}{f^2} \rceil$-th smallest element in G as p_i. To do so, we apply the k-selection algorithm f times, but since G has only $O(N/f)$ elements, the total cost of all the k-selection is $O(\frac{N}{fB} \cdot f) = O(N/B)$.

We claim that $p_1, ..., p_f$ are indeed what we need:

Lemma 2. *For each* $i \in [1, f+1]$*, the size of* $S \cap (p_{i-1}, p_i]$ *is at most* $10N/f$.

Proof. Let $S_i = S \cap (p_{i-1}, p_i]$. We will analyze the number x_j of elements in S_i that are contributed by group G_j ($1 \leq j \leq \lceil N/M \rceil$). We achieve this purpose by examining the number y_j of representatives from G_j that are in the range $(p_{i-1}, p_i]$.

By the way that group representatives are selected, we have:

$$x_j \leq y_j \cdot f + f \tag{2}$$

On the other hand, by the way that splitters are picked, we have:

$$\sum_j y_j = \left\lceil \frac{N}{f^2} \right\rceil. \tag{3}$$

Combining (2) and (3) gives

$$\sum_j x_j \leq \left\lceil \frac{N}{f^2} \right\rceil \cdot f + f \cdot \left\lceil \frac{N}{M} \right\rceil \leq 2\frac{N}{f} + 8f \cdot \frac{N}{f^2} \leq 10N/f.$$

□

Remark. The constant 10 is due to the looseness of our analysis, and can be made considerably smaller with a more careful analysis.

3.3 Distribution Sort

We are now ready to clarify the details of distribution sort. Recall that the input is a set S of N elements in $\mathbb{R}$. If S fits in memory, then we solve the problem in $O(N/B)$ I/Os by doing the sorting in memory. We will assume, without loss of generality, that M/B is at least a sufficiently large constant.

Now let us focus on the case where $|S| > M$. Set $f = \sqrt{M/B}$. Use the algorithm of Section 3.2 to find splitters $p_1, ..., p_f$. Define, for each $i \in [1, f]$, $S_i = S \cap (p_{i-1}, p_i]$, defining dummy splitters $p_0 = -\infty$ and $p_{f+1} = \infty$. $S_1, S_2, ..., S_f$ can be easily obtained in $O(N/B)$ I/Os. Now, sort each S_i recursively for each $i \in [1, f]$, and then, concatenate $S_1, ..., S_f$ into the final sorted order.

Denote by $F(N)$ the cost of distribution sort on a set with cardinality N. It holds that, when $N > M$

$$F(N) \leq \sum_{i=1}^{f} F(|S_i|) + O(N/B)$$

where each $|S_i| \leq 10N/f$ and $\sum_{i=1}^{f} |S_i| = N$. Also, $F(N) = O(N/B)$ when $N \leq M$. Solving the recurrence gives

$$F(N) = O((N/B) \log_f (N/M)) = O((N/B) \log_{M/B} (N/B))$$

noticing that there are at most $O(\log_{f/10}(N/M)) = O(\log_f (N/M))$ levels in the recursion when M/B (and hence f) is greater than a large enough constant.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.