Lecture Notes: Permutation in External Memory

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

In this lecture, we will study the *permutation problem*. The input is an array A_1 of size n where the i-th cell ($1 \leq i \leq n$) is a pair (e_i, p_i). We refer to e_i as an *element*, which fits in one word but its contents are of no interest to us. The second field p_i is an integer from 1 to n such that the values of $p_1, p_2, ..., p_n$ are distinct. The goal is to output another array A_2 of n elements where e_i is placed at the p_i-th position (note that e_i was in the i-th cell of A_1).

In RAM, we can easily solve the problem optimally in $O(n)$ time: for each $i \in [1, n]$, set $A_2[p_i] = e_i$. The problem is far more interesting in EM, where A_1 is given to us in $O(n/B)$ blocks, and we need to output A_2 also in $O(n/B)$ blocks. There are two obvious solutions. The first one is to sort the elements $e_1, ..., e_n$ by their designated positions $p_1, ..., p_n$; the cost is $O((n/B) \log_{M/B}(n/B))$ I/Os. The second solution is to simply *ignore* blocking, and apply the RAM algorithm directly; the cost is $O(n)$, because writing to each position of A_2 may cost us an I/O.

A natural question arises—since it is possible to achieve linear cost (i.e., $O(n)$) in RAM, is it possible to achieve linear cost (i.e., $O(n/B)$) in EM? This is one of the biggest open problems in EM. What is known is that this is impossible for an important class of algorithms that obey the so-called *indivisibility assumption* (both the naive solutions mentioned earlier are in this class). Below, we will clarify the assumption, and prove that any algorithm in this class must incur $\Omega(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{B}\})$ I/Os in the worst case. In other words, the best thing we can do is to simply choose between the two naive solutions!

1 The Indivisibility Model

Remember that any analysis aiming to upper bound the running time of an algorithm must be carried out under a computation model (e.g., EM). Likewise, every lower bound argument also comes with its *own* computation model, such that the argument applies to all and only the algorithms under that model.

We will now define a lower-bound model called the *indivisibility model*. Same as in EM, our machine is equipped with finite memory and an infinite disk formatted into disjoint blocks. Both the memory and the disk, however, store nothing but elements. Specifically, at any moment, the memory always stores M elements (counting an "empty slot" also as a special element), while each disk block always stores B elements. An algorithm is allowed only two operations:

- *Read I/O*: Load a block of elements into memory, replacing B elements there.
- *Write I/O*: Collect B elements in the memory, and write them to a disk block, erasing the elements that the block originally contained.

The *cost* of an algorithm is the number of I/Os performed.

You may be wondering: why haven't we defined any operation for computing? The answer is: *we don't care!* The above model is concerned only with the movement of elements, but not with *how* the algorithm learns to do the movement. To illustrate, let us consider the following problem. Suppose that we are given a set S of n elements stored in n/B blocks. Now we want to load a particular element $e \in S$ into memory, but do not know which is the block containing it. How

many I/Os do we need to perform in the worst case? In the indivisibility model, the answer is one—simply go to the block of e, and read it—and we do not have to worry about how to find the address of that block! In EM, clearly we need n/B I/Os for the same purpose, but how does the definition of the EM model prevent the "magic algorithm" of only one I/O? The answer lies in the EM requirement that, before an algorithm reads/writes a block, it should have already obtained the block's address through its execution so far. Hence, the "magic algorithm" is not really a valid algorithm in the EM model.

You may still be wondering: are algorithms in the indivisibility model more powerful? This intuition is valid (but not completely correct, as we will see), and can be formalized as follows. Suppose that an algorithm in the EM model always treats each element as an *atom*—in other words, the algorithm is not allowed to, for example, (i) break an element into pieces and store them in different blocks; or (ii) compress an element. We can convert this algorithm to work in the indivisibility model with (at most) the same cost (think: how?). The other way around, however, is not true. Namely, an algorithm in the indivisibility model does not necessarily have a counterpart in the EM model with the same cost—we have seen an example earlier.

The above asymmetry indicates that the indivisibility model is not a good model for studying real algorithms, but it is a good model for studying lower bounds. Specifically, if we can show that any algorithm in this model must perform at least x I/Os solving a problem, then any EM algorithm must also perform at least x I/Os.

Why not study lower bounds in the EM model directly? The answer is that some EM algorithms may potentially be even more powerful than those in the indivisibility model. If an EM algorithm does *not* treat elements as atoms, then it has no counterpart in the indivisibility model; and therefore, a lower bound in the indivisibility model does not apply to such an EM algorithm. Now it should be clear that neither of the indivisibility model and the EM model subsumes the other. A lower bound in the indivisibility model applies only to a class of algorithms in EM—this class is often referred to as the *indivisibility class*.

2 A Permutation Lower Bound

Next, we will prove that any algorithm in the indivisibility model must perform $\Omega(\min\{n, \frac{n}{B}\log_{M/B}\frac{n}{B}\})$ I/Os to solve the permutation problem in the worse case. This argument is due to Aggarwal and Vitter [1].

We consider $n \geq 2B$. Fix an arbitrary algorithm; and let H be the number of I/Os it performs in the worst case. We will consider $H \leq n^2$ (otherwise, H is already $\Omega(\min\{n, \frac{n}{B}\log_{M/B}\frac{n}{B}\})$. Also, we will consider that the algorithm never reads an empty block (i.e., a block where all elements are empty slots)—if it does, then we can ignore the I/O and simply remove the B elements in memory that the algorithm replaces with the I/O; doing so can only reduce the algorithm's cost.

We say that a block is *used* if it is either a block used to store the input, or a block that has ever been written by the algorithm. We regard each used block as a multi-set of size B, consisting of the elements in the block (namely, we ignore the elements' ordering in the block). Similarly, we will also regard the memory as a multi-set of size M.

We define the *state* of an algorithm as a sequence Σ of multi-sets. The first multi-set of Σ is the memory. Then, the rest of Σ corresponds to the sequence of all the used blocks, in ascending order of their addresses. The algorithm's *initial state* is clear: the memory contains M empty elements, and the rest of Σ is the sequence of blocks containing our input. Note that, at any moment, the number of used blocks is at most H.

The crux of our argument is to analyze how many new states can be generated from any state Σ

by performing one I/O. Let us first assume that the I/O is a read. After the I/O, only the memory has changed in Σ (the contents of all the used blocks remain the same). Observe that there are at most $H \cdot \binom{M}{B}$ ways that the memory can change. First, there are H used blocks the algorithm can choose from. Second, no matter which block is read, the algorithm has $\binom{M}{B}$ ways to select B elements in memory to replace.

The situation of a writing I/O is only slightly more complex. After the I/O, the memory will remain the same, but two types of changes can occur in the sequence of used blocks:

- An existing used block may have been replaced. There are at most $H \cdot \binom{M}{B}$ new states created this way (at most H used blocks to write to, and $\binom{M}{B}$ ways to form up a block from memory).
- A new used block may have been created. In this case, Σ had at most $H-1$ used blocks before the I/O (otherwise, the algorithm will have to perform $H+1$ I/Os eventually, contradicting that H is a worst case bound). The new used block can be inserted into H positions in relation to the existing used blocks. Hence, there are also at most $H \cdot \binom{M}{B}$ new states created this way.

In summary, a single I/O can generate at most $3H \cdot \binom{M}{B}$ new states from Σ, no matter what Σ is. It thus follows that by performing H I/Os, the algorithm can create at most $(3H \cdot \binom{M}{B})^H \leq (3n^2 \cdot \binom{M}{B})^H$ states from the initial state.

How many states must the algorithm be able to create to correctly solve the permutation problem? At least, $n!/(B!)^{n/B}$—there are $n!$ possible permutations, and for each permutation, we do not care about the element ordering in each block. Hence, we have (all the logs have base 2 by default):

$$\begin{aligned} \left(3n^2 \cdot \binom{M}{B}\right)^H &\geq \frac{n!}{(B!)^{n/B}} \\ \Rightarrow H \log \left(3n^2 \cdot \binom{M}{B}\right) &\geq \log \frac{n!}{(B!)^{n/B}} \end{aligned}$$

In general, for any integer $x > 0$, it holds that $x \log \frac{x}{e} \leq \log(x!) \leq \log(2.6\sqrt{x}) + x \log \frac{x}{e}$. Also, for any integers $x, y \geq 0$, it holds that $\log \binom{x}{y} \leq y \log \frac{ex}{y}$. Using these inequalities, we have:

$$\begin{aligned} H\left(\log(3n^2) + B \log \frac{eM}{B}\right) &\geq n \log \frac{n}{e} - \frac{n}{B}\left(\log(2.6\sqrt{B}) + B \log \frac{B}{e}\right) \\ \Rightarrow H\left(\log(3n^2) + B \log \frac{eM}{B}\right) &\geq n \log \frac{n}{B} - \frac{n}{B} \log(2.6\sqrt{B}) \end{aligned}$$

When n and B are sufficiently large, we have:

$$\begin{aligned} 3H\left(\log n + B \log(M/B)\right) &\geq \frac{n}{2} \log(n/B) \\ \Rightarrow H &\geq \frac{\frac{n}{6} \log(n/B)}{\log n + B \log(M/B)} \end{aligned}$$

When $\log n \geq B \log(M/B)$ or equivalently, $n \geq (M/B)^B \geq 2^B$ (recall that $M \geq 2B$), $H = \Omega(n)$. Otherwise, $H = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

References

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