6.854 Advanced Algorithms

Lecture 21: 11/24/2003 Lecturer: Erik Demaine

Scribes: David R. Cheng

External Memory Algorithms

21.1 Preliminaries

Now consider a change in the model that takes into account the memory hierarchy of a modern computer. Previously, the algorithms we study assume the system has CPU and Memory, every memory access having equal cost. In practice, we typically have a CPU, several levels of caches, main memory, and disk. At each memory level away from the CPU, the capacity available and the cost of an access both increase. So it's cheap to access a small number of things, and costs more to access everything.

We will study algorithms in the context of the External Memory Model [AV88]. We have two levels in the memory hierarchy: cache and disk. The CPU can access the cache quickly, while there is a large overhead cost (seek time) to access the disk. Once the disk overhead is paid, the CPU can move data quickly. To capture this with the model, we say the disk is partitioned into *blocks* (or pages) of size B, and each read or write is of an entire block. Also, the cache has size M and holds $\frac{M}{B}$ blocks, while the disk holds an infinite number of blocks. The model is illustrated in Figure 21.1.

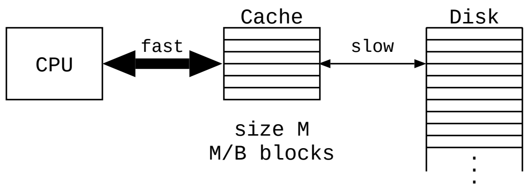


Figure 21.1: External Memory Model

The objective is to consider the number of reads and writes to disk, called *memory transfers*, that an algorithm makes. Note that any algorithm with time $T(N)$ makes at most $T(N)$ memory transfers. We want to reduce this cost.

21.2 Basic Algorithms

Note the common number of memory transfers of $O(\lceil \frac{N}{B} \rceil)$ in these algorithms. This can be thought of as "linear."

21.2.1 Scanning

We have an array of N elements stored consecutively on disk and want to compute the sum. By reading one block at a time, we pay $\lceil \frac{N}{B} \rceil$ memory transfers. This algorithm is optimal, because we have to pay for each block the array occupies. Note that only one block of the cache is used.

21.2.2 Reversing an Array

Reversing the order of elements in an array is like performing two scans: reverse the order of the blocks, and reverse all the elements within each block. This can be done by reading in opposing blocks, reversing their elements in cache, then writing them out with swapped positions. The number of memory transfers is again $O(\lceil \frac{N}{B} \rceil)$, and the cache needs to hold at least two blocks.

21.2.3 Merging Two Sorted Arrays

Given two input arrays already sorted, we want to output a new sorted array from their merge. The algorithm initially reads the first block from both inputs, merges them in cache to create an output block, and writes it out. Whenever an input block has been entirely consumed, it reads the next block from the input. The number of memory transfers is $O(\lceil \frac{N}{B} \rceil)$, and at most 3 blocks of cache are used at all times.

21.3 Search Tree

We want to maintain the items of a tree in external memory, supporting the operations Insert, Delete, and Search.

One idea is to use a heap, because its structure is easily represented by an array. To see how this is not a great choice, consider the similar task of performing binary search. At each step, an entire block is read for a single element; not until the end when the interval has size $\leq B$ can we take advantage of reading blocks. The number of memory transfers is $O(\lg N - \lg B)$.

21.3.1 Using a B-tree

A better idea is to use a B-tree. Recall that every node of a B-tree has at most B items (in sorted order), and at most $B + 1 = \Theta(B)$ children. Choose the node size B to coincide with the block size B; thus, we can read one node at a time. The total number of memory transfers per operation is $O(\text{height}) = O(\log_{B+1} N) = O\left(\frac{\lg N}{\lg(B+1)}\right)$. Note that this algorithm saves a multiplicative factor of $\lg B$, rather than the additive factor of the heap.

21.3.2 Lower Bound

The problem of finding an element in the B-tree is at least as difficult as finding the position of some item x among N sorted numbers. We can find a lower bound on the number of memory transfers to solve this second problem in the comparison model. The argument is information theoretic:

$$\#\text{ memory transfers} \geq \frac{\min \#\text{ bits to be determined}}{\max \#\text{ bits each block can give}}$$

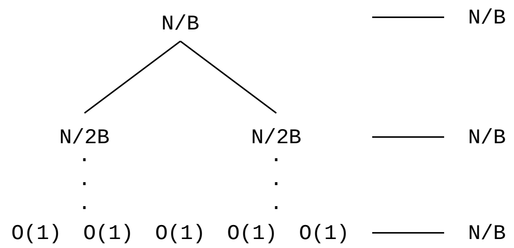
Note there are $\Theta(N)$ answers (positions), so we want to determine $\Theta(\lg N)$ bits of information. Now given a block, we can compare x to each element in the block to learn $O(B)$ ranks that x does or does not have. Thus, each block reveals at most $\Theta(\lg(B+1))$ bits. The total number of block reads is then at least $\Theta\left(\frac{\lg N}{\lg(B+1)}\right)$. The B-tree is optimal.

21.4 Sorting (Comparison Model)

The problem is to sort an array of N elements on disk. We can easily get number of memory transfers $O(N \log_{B+1} N)$ by inserting all elements into a B-tree, then traversing.

21.4.1 Merge Sort

We can do a little better by the merge sort. The recurrence on memory transfers is $T(N) = 2T(N/2) + O(N/B)$. The last term is the cost of merging two sorted arrays, determined in Section 21.2.3. The base case is $T(B) = O(1)$, because we can sort B elements in cache without hitting disk. To solve this recurrence, we can look at the recursion tree:

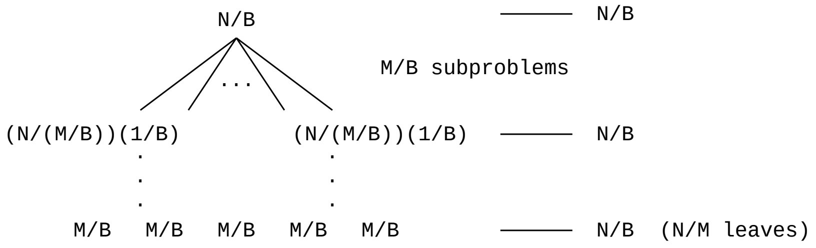


The cost at each level is $O(\frac{N}{B})$, and the number of levels is $\lg \frac{N}{B}$ because there are $\frac{N}{B}$ children. The total number of memory transfers is therefore $O\left(\frac{N}{B} \lg \frac{N}{B}\right)$.

21.4.2 $\frac{M}{B}$-Merge Sort

A better idea is to merge more than two lists at once, making use of the cache capacity of M: we can merge $\frac{M}{B} - 1 \approx \frac{M}{B}$ blocks without hitting disk. Assume the cache size is actually $M + O(1)$ so the approximation holds and we have some room to work with $\frac{M}{B}$ blocks present.

The algorithm splits the input into $\frac{M}{B}$ pieces and recursively solves each piece. The merge is begins by taking the first block from each piece. The $\frac{M}{B}$-way merge proceeds and writes out new blocks in sorted order, reading subsequent blocks from the pieces as necessary. The number of memory transfers for this merge is $O\left(1+\frac{N}{B}\right)$. This gives us the following recurrence: $T(N) = \frac{M}{B}T\left(\frac{N}{M/B}\right) + O\left(1+\frac{N}{B}\right)$. The base case is $T(M) = O\left(\frac{M}{B}\right)$. Again, we solve by looking at the recursion tree:



Again, the total cost of each level is $\frac{N}{B}$, for total cost

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

21.4.3 Lower Bound

Again, we can make an information theoretic argument for the lower bound of sorting. We want to determine the sorted order, among $N!$ possible orders; this is $O(N\lg N)$ bits. If we make the simplifying assumption that a block is sorted when we read it, then we need $O\left(N\lg\frac{N}{B}\right)$ bits. Now we want to determine the bits learned from reading a block. Recall that we have M elements already in the cache, and then read a block of B elements. After comparing all the elements of the block with those in memory, we have learned how B elements are interleaved in $M+B$ total elements. There are $\binom{M+B}{B}$ possible interleavings, so the information learned is:

$$\begin{aligned}
\lg\binom{M+B}{B} = \lg\frac{(M+B)!}{B!M!} &= \lg(M+B)! - \lg(B!) - \lg(M!) \\
&\approx (M+B)\lg(M+B) - B\lg B - M\lg M \\
&\approx (M+B)\lg M - B\lg B - M\lg M \\
&= B\lg M - B\lg B \\
&= B\lg\frac{M}{B}
\end{aligned}$$

So the number of block reads is at least $\frac{N\lg N/B}{B\lg M/B} = \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$. The $\frac{M}{B}$-merge sort is optimal.

21.5 Buffer Trees [Arg95]

It is slightly annoying that sorting is faster than building search trees. We want a data structure that does $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers per operation, so we can get achieve the lower bound for sorting with it. Note this is $o(1)$ memory transfers per operation. Buffer trees are a data structure that support the operations Insert, Delete, Batched Search, Batched Range-Search, and Delete-Min all with amortized $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers. The batched operations don't provide the answer right away; rather, as other operations are made, the buffer tree returns pieces of the answer. Eventually, the entire answer will come out. At any time, we can ask for all results with the Flush operation, but we must pay for this.

The buffer tree is similar to the B-tree of Section 21.3.1. Each internal node has size $\Theta\left(\frac{M}{B}\right)$ while each leaf has size $\Theta(B)$. Corresponding with each internal node is a buffer of size M. The buffer of the root node is kept in cache at all times. Additionally, all the elements of the buffer tree are either in leaves or buffers; the internal nodes hold copies. Figure 21.2 illustrates the data structure.

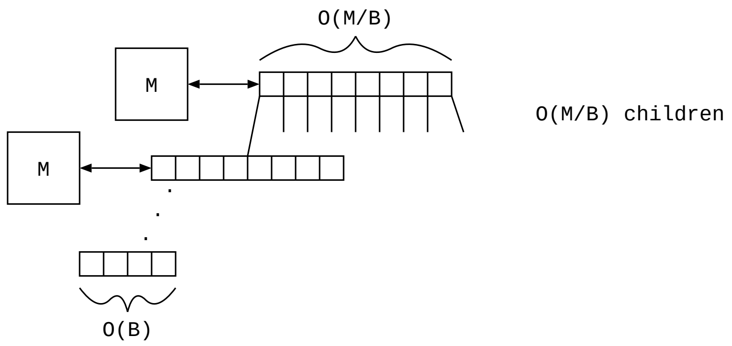


Figure 21.2: Buffer Tree

21.5.1 Insert

1. Append element to the root's buffer
2. If the buffer overflows, we want to push all the elements to the children
   (a) Sort the buffer
   (b) Distribute items to buffers of the appropriate children
      i. Do this by visiting the children in order, with a linear scan
      ii. Have $O\left(\frac{M}{B}\right)$ children, and touching each child takes one transfer
      iii. Have M elements, occupying $O\left(\frac{M}{B}\right)$ blocks (if contiguous)
      iv. Total cost is $O\left(\frac{M}{B}\right)$
   (c) If pushing the elements to a child causes an overflow,
      i. Child's new buffer has size $\leq 2M$

   ii. Sort each half, then merge
   iii. Distribute among children, checking for overflow here as well
   iv. This can go on for a while if all the buffers are full

(d) If a leaf overflows,
   i. Split the leaf repeatedly as a B-tree would, possibly causing splits up the tree
   ii. Do this only after all the other overflows are handled. The buffers on the path from the leaf to the root will be empty, so the performance is the same as for a B-tree.

For the analysis, note that a single overflow costs $O\left(\frac{M}{B}\right)$ memory transfers. We charge that to the M elements moved down to a lower level. An element can only go down, so the number of charges to any element is $O(\text{height}) = O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$. This gives us amortized cost per insertion of $O\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$.

21.5.2 Delete-Min

The techniques used to make this operation efficient are reminiscent of the van Emde Boas data structure. In a typical B-tree, the minimum element is the leftmost leaf. However, we may not have flushed the buffer tree, so the minimum element can be any buffer on the leftmost path. To perform Delete-Min efficiently, we keep an extra buffer in the cache of the smallest $\frac{M}{2}$ elements. Call this the min buffer. For convenience in this section, assume the root buffer also has size $\frac{M}{2}$.

The algorithm to delete the minimum element typically removes it from the min buffer, for cost 0. When the min buffer becomes empty, the algorithm needs to fill it back up. This is done by flushing all buffers on the leftmost path, then extracting $\Theta(M)$ elements from the siblings of the leftmost leaf. Furthermore, the min buffer needs to be maintained when insertions are performed. We do this by first checking if the new element is smaller than the largest element in the min buffer, and swapping if so.

The only cost is the flush, when the min buffer becomes empty. Each buffer we flush requires $O\left(\frac{M}{B}\right)$ memory transfers, and there are $O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$ buffers on the leftmost path. The cost of a flush is only incurred after $O(M)$ calls to Delete-Min, so the amortized cost is $O\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$.

21.5.3 Flush

This operation empties all buffers. There are two cases: the buffer is either full or not full. We can perform sort by inserting all elements, flushing all the buffers, then reading the leaves. Therefore, the cost of emptying all the full buffers is $O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$. Now we consider the non-full buffers. There are $\frac{N}{B}$ leaves, so the number of internal nodes one level up from the bottom is $O\frac{N/B}{M/B}$. This means there are only $O\left(\frac{N/B}{M/B}\right)$ total buffers. We empty these buffers in breadth-first order, for cost $O\left(\frac{N/B}{M/B}\right) \times O\left(\frac{M}{B}\right) = O\left(\frac{N}{B}\right)$. So the cost of a flush is dominated by emptying the full buffers.

21.5.4 Batched Range-Search

We can do this with $O\left(\frac{1}{B}\log_{\frac{M}{B}} + \frac{K}{B}\right)$ memory transfers, K being the size of the output. The algorithm adds the search request, represented by the lower endpoint, to the root's buffer. While sorting or merging buffers, the algorithm scans for matches of buffered queries and output elements as they are found. When a buffer flushes and the elements are distributed among the children, the queries are replicated to the appropriate children. Upon reaching a leaf, the query outputs the matching elements and finishes.

References

[Arg95] Lars Arge. The buffer tree: A new technique for optimal i/o-algorithms. *Lecture Notes in Computer Science*, 955, 1995.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.