External-Memory Sorting
(lecture notes)

Simonas Šaltenis

1 External Memory Model

When data do not fit in main memory (RAM), external (or secondary) memory is used. Magnetic disks are the most commonly used type of external memory. When compared to RAM, disks have these properties (see chapter 18 of [1] for a more thorough discussion):

1. Usually disks can store much more data than RAM.

2. Access to data on disk drives is much slower than access to RAM (by orders of magnitude).

3. Because of the mechanics of disk drives, it takes a lot of time to access a random byte on a disk, but it is relatively fast to transfer that byte and subsequent bytes from the disk to RAM. This means that it is beneficial to access data on the disk in large portions called *blocks* or *pages*. The block sizes from 2Kb to 16Kb are common for modern disk drives.

Because access to disk drives is much slower than access to RAM, analysis of external-memory algorithms and data structures usually focuses on the number of disk accesses (I/O operations), not the CPU cost.

When data is stored on the disk, algorithms that are efficient in main memory may not be efficient when the running time of the algorithm is expressed as the number of I/O operations. External memory algorithms are designed to minimize the number of I/O operations.

In the following we assume that we have a data file of N data elements (records). Each disk block can store B data elements. We also assume that the main memory available to the algorithm can store M data elements ($M < N$). Then the number of disk pages in the data file is $n = N/B$ and the number of disk pages that fit in the available memory is $m = M/B$. Notice that we treat both B and M (or m) as important problem parameters in our asymptotic performance bounds. They are not considered "just constants" which we can ignore.

2 Main-Memory Merge Sort

Before considering algorithms for external-memory sorting, we look at the merge-sort algorithm for main-memory sorting. Some ideas from this algorithm are useful when considering external-memory sorting (see Section 2 of [1] for more details).

The main component of the MERGE-SORT algorithm is the MERGE procedure, which takes two sorted arrays as input and merges them into one sorted array.

The MERGE procedure repeatedly compares the smallest remaining element from one input array with the smallest remaining element from the other input array. It then moves the smallest of the two elements to the output array and repeats, until one of the input arrays becomes empty. At the end, all the remaining elements from the non-empty array are moved to the output array. Figure 1 illustrates the merging of two sorted arrays, each storing four integers. At each step of the algorithm, the smallest elements in the two arrays that are compared are shown in bold.

Operation	Array 1	Array 2	Output
compare $1 < 3$	**1**, 4, 6, 7	**3**, 5, 8, 9	
compare $3 < 4$	**4**, 6, 7	**3**, 5, 8, 9	1
compare $4 < 5$	**4**, 6, 7	**5**, 8, 9	1, 3
compare $5 < 6$	**6**, 7	**5**, 8, 9	1, 3, 4
compare $6 < 8$	**6**, 7	**8**, 9	1, 3, 4, 5
compare $7 < 8$	**7**	**8**, 9	1, 3, 4, 5, 6
copy array 2 to output		8, 9	1, 3, 4, 5, 6, 7
end			1, 3, 4, 5, 6, 7, 8, 9

Figure 1: Merging two sorted arrays into one

Once we have the MERGE procedure, the MERGE-SORT algorithm is very simple. Here, it is assumed that MERGE($A, A1, A2$) merges arrays $A1$ and $A2$ into array A.

```
MERGE-SORT($A$)
1  if $A$ contains more than one element
2      Copy the first half of $A$ into array $A1$, and the second half of $A$ into array $A2$
3      MERGE-SORT($A1$)
4      MERGE-SORT($A2$)
5      MERGE($A, A1, A2$)
```

As explained in [1], the running time of this algorithm is $\Theta(n \log_2 n)$, which is asymptotically optimal for the comparison-based main memory sorting.

3 External-Memory Merge Sort

If we "unwind" the recursion, the MERGE-SORT algorithm works by first merging pairs of "sub-arrays" of 1 element into sorted sub-arrays of 2 elements, then merging pairs of sorted sub-arrays of 2 elements into sorted sub-arrays of 4 elements, and so on. Let us call a sorted sub-array of elements a *run*. Then, in each step the number of runs decreases twofold and the algorithm stops when only one run remains.

We can use the same procedure for external-memory sorting, but instead of starting with trivial runs of 1 element, we start with runs of size M (the available main memory). Let us assume that X points to a file that stores the input data which has to be sorted and Y points to an empty file. The *external-memory merge-sort* algorithm has these two main phases:

- *Phase 1 (produce initial runs)*: Repeat the following process until the end of file X is reached:

    1. Read the next M elements from file X into main memory.
    2. Sort them in main memory using any main-memory sorting algorithm.

3. Write the sorted elements at the end of file Y.

   At the end of this phase, we have, in file Y, $\lceil N/M \rceil$ runs of M elements (the last run may be shorter).

- *Phase 2 (merge runs)*: Repeat the following while there is more than one run in file Y:

  1. Make file X empty.
  2. Repeat the following until the end of file Y is reached: call TWOWAY-MERGE to merge the next two runs from file Y into one run, which is written at the end of file X. If only one run remains in Y, just copy it at the end of file X.
  3. Exchange pointers X and Y: make new X point to file Y and new Y to file X.

At the end of this algorithm the sorted sequence of elements is in file Y.

Phase 2 of the algorithm works in essentially the same way as the the main-memory merge sort, except that main-memory MERGE algorithm can not be used to merge two runs of M or more elements stored in external memory. Instead we use a TWOWAY-MERGE(X, Y, l, q, r) algorithm, which merges the first run consisting of pages l through $q-1$ from file Y with the second run consisting of pages q through $r-1$ from file Y. The merged output is appended to file X.

The TWOWAY-MERGE algorithm maintains three main-memory arrays of size B (i.e., storing one disk page each). The first two arrays *Bf1* and *Bf2* are buffers for disk pages read from the first and the second runs. The third array is an output buffer. The algorithm works exactly as the main-memory MERGE, merging buffers *Bf1* and *Bf2* into buffer *Bfo*. If the end of *Bf1* or *Bf2* is reached, the next page from the corresponding run is read into main memory. Also, as soon as *Bfo* becomes full it is flushed to the end of file X. Figure 2 visualizes the merging of main memory buffer pages.

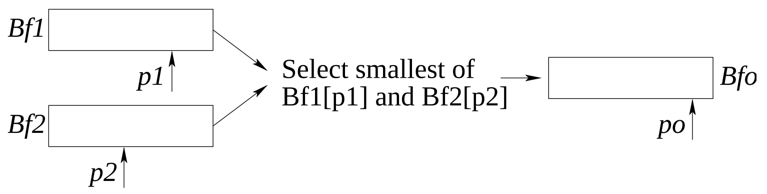

Figure 2: Main-memory organization for two-way merging

In the pseudocode of TWOWAY-MERGE, which is on the next page, we assume that *DiskRead*(*Bf*, Y, *k*) reads the k-th page of file Y into main-memory array *Bf*. Similarly, *DiskWrite*(*Bfo*, X) appends the contents of main-memory array *Bfo* to the end of file X.

The described external-memory merge-sort algorithm can sort a file of any size. Let us analyze its running time, i.e., count the number of I/O operations it performs.

Phase 1 of the algorithm just reads all the pages from file X and writes the same amount of pages to file Y. Thus $2n = \Theta(n)$ I/O operations are performed (remember that n is the number of disk pages in the initial file).

In one iteration of the main loop of phase 2, disk pages of all runs in file Y are read once and the same amount of pages is written to file X. Again $2n = \Theta(n)$ I/O operations are performed. How many loop iterations are there in phase 2? Each iteration reduces the number of runs twofold. We start with $\lceil N/M \rceil = \lceil n/m \rceil$ runs and finish with one run. Thus, there are $\log_2(n/m)$ loop iterations each doing $\Theta(n)$ I/O operations.

```
TWOWAY-MERGE($X, Y, l, q, r$)
r1 $\leftarrow l$    ▷ pointer to a page in run 1
r2 $\leftarrow q$    ▷ pointer to a page in run 2
DiskRead(Bf1, $Y$, r1)    ▷ buffer page for run 1
DiskRead(Bf2, $Y$, r2)    ▷ buffer page for run 2
p1 $\leftarrow 1$    ▷ pointer to an element in Bf1
p2 $\leftarrow 1$    ▷ pointer to an element in Bf2
po $\leftarrow 1$    ▷ pointer to an element in Bfo
while r1 $< q$ and r2 $< r$
    if Bf1[p1] $<$ Bf2[p2]
        Bfo[po] $\leftarrow$ Bf1[p1]
        p1 $\leftarrow$ p1 $+ 1$
        if p1 $> B$    ▷ need a new page from run 1
            r1 $\leftarrow$ r1 $+ 1$
            if r1 $< q$    ▷ if not the end of run 1
                DiskRead(Bf1, $Y$, r1)
                p1 $\leftarrow 1$
    else    ▷ Bf1[p1] $\geq$ Bf2[p2]
        Bfo[po] $\leftarrow$ Bf2[p2]
        p2 $\leftarrow$ p2 $+ 1$
        if p2 $> B$    ▷ need a new page from run 2
            r2 $\leftarrow$ r2 $+ 1$
            if r2 $< r$    ▷ if not the end of run 2
                DiskRead(Bf2, $Y$, r2)
                p2 $\leftarrow 1$
    po $\leftarrow$ po $+ 1$
    if po $> B$    ▷ output buffer page Bfo is full
        DiskWrite(Bfo, $X$)
        po $\leftarrow 1$
if r1 $= q$
    Copy elements Bf2[p2], $\ldots$, Bf2[$B$] to Bfo, write Bfo to disk (file $X$), and copy
    the remaining pages of run 2 (from r2 $+ 1$ to $r - 1$) to the end of $X$.
else    ▷ r2 $= r$
    Do the same for Bf1 and run 1.
```

Summing up the costs of phase 1 and phase 2 we get the total running time of $\Theta(n \log_2(n/m))$. It turns out, this is not the best we can do.

How can we make the algorithm more efficient? The important observation is that, while we use all the available main-memory in phase 1, we use only 3 out of m available pages of main-memory in phase 2. It is easy to see that allocating larger buffers *Bf1*, *Bf2* and *Bfo* does not change the number of performed I/O operations (although it reduces the number of random I/O operations—see the last section). Instead, as the next section explains, we need to merge not two but more runs at the same time.

4 Two-Phase, Multiway Merge Sort

The *two-phase, multiway merge-sort* algorithm is similar to the external-memory merge-sort algorithm presented in the previous section. Phase 1 is the same, but, in phase 2, the main loop is performed only once merging all $\lceil N/M \rceil$ runs into one run in one go. To achieve this, multiway merging is performed instead of using the TWOWAY-MERGE algorithm.

The idea of multiway merging is the same as for the two-way merging, but instead of having 2 input buffers (*Bf1* and *Bf2*) of B elements, we have $\lceil N/M \rceil$ input buffers, each B elements long. Each buffer corresponds to one unfinished (or active) run. Initially, all runs are active. Each buffer has a pointer to the first unchosen element in that buffer (analogous to *p1* and *p2* in TWOWAY-MERGE).

The multiway merging is performed by repeating these steps:

1. Find the smallest element among the unchosen elements of all the input buffers. Linear search is sufficient, but if the CPU cost is also important, minimum priority queue can be used to store pointers to all the unchosen elements in input buffers. In such a case, finding the smallest element is logarithmic in the number of the active runs.

2. Move the smallest element to the first available position of the output buffer.

3. If the output buffer is full, write it to the disk and reinitialize the buffer to hold the next output page.

4. If the buffer, from which the smallest element was just taken is now exhausted of elements, read the next page from the corresponding run. If no pages remain in that run, consider the run finished (no longer active).

When only one active run remains the algorithm finishes up as shown in lines 30 and 32 of TWOWAY-MERGE—it just copies all the remaining elements to the end of file X.

Figure 3 visualizes multiway merging.

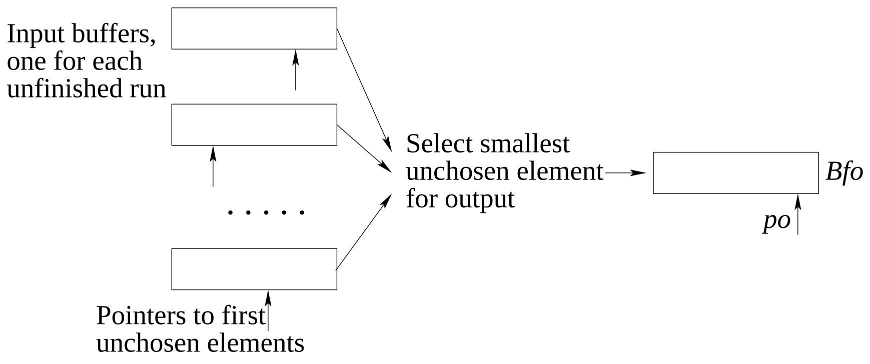

Figure 3: Main-memory organization for multiway merging

It is easy to see that phase 2 of the two-phase, multiway merge-sort algorithm performs only $\Theta(n)$ I/O operations and this is also the running time of the whole algorithm. In spite of this, the algorithm has a limitation—it can not sort very large files.

If phase 1 of the algorithm produces more than $m-1$ runs ($N/M > m-1$), all runs can not be merged in one go in phase 2, because each run requires a one-page input buffer in main-memory and one page of main-memory is reserved for the output buffer. How large should the file be for this to happen?

Consider a computer with 256Mb of available main-memory and the disk page size of 8Kb. If the sorted elements are four-byte integers, $M = 2^{26}$ and $B = 2^{11}$. Then,

$$\frac{N}{M} > m - 1 \quad \Rightarrow \quad N > M\left(\frac{M}{B} - 1\right) = \frac{M^2}{B} - M \approx 2^{41}.$$

The file of 2^{41} four-byte integers occupies 8 terabytes. Thus, two-phase, multiway merge sort will work for most of the practical file sizes.

5 Multiway Merge Sort of Very Large Files

Sometimes there may be a need to sort extremely large files or there is only a small amount of available main memory. As described in the previous section, two-phase, multiway merge sort may not work in such situations.

A natural way to extend the two-phase, multiway merge sort for files of any size is to do not one but many iterations in phase 2 of the algorithm. That is, we employ the external-memory merge-sort algorithm from Section 3, but instead of using TWOWAY-MERGE, we use the multiway merging (as described in the previous section) to merge $m - 1$ runs from file Y into one run in file X. Then, in each iteration of the main loop of phase 2, we reduce the number of runs by a factor of $m - 1$.

What is the running time of this algorithm, which we call simply *multiway merge sort*. Phase 1 and each iteration of the main loop of phase 2 takes $\Theta(n)$ I/O operations. After phase 1, we start up with $\lceil N/M \rceil = \lceil n/m \rceil$ runs, each iteration of the main loop of phase 2 reduces the number of runs by a factor of $m - 1$, and we stop when we have just one run. Thus, there are $\log_{m-1}(n/m)$ iterations of the main loop of phase 2. Therefore, the total running time of the algorithm is $\Theta(n \log_{m-1}(n/m)) = \Theta(n \log_m n - n \log_m m) = \Theta(n \log_m n - n) = \Theta(n \log_m n)$.

Remember that the cost of the external-memory merge-sort algorithm from Section 3 is $\Theta(n \log_2(n/m))$. Thus, multiway merge sort is faster by a factor of $\Theta\left(\left(1 - \frac{1}{\log_m n}\right) \log_2 m\right)$. Actually, $\Theta(n \log_m n)$ is a lower bound for the problem of external-memory sorting. That is, multiway merge sort is an asymptotically optimal algorithm.

6 Sequential vs. Random I/O

In all the analysis of the previous sections, we did not distinguish between the sequential and random access of disk pages. If sequential access is much faster than random, then it may be wise to minimize the number of random I/O operations, even if the total number of I/O operations increases.

Let us look at the multiway merge-sort algorithm and let us assume that files X and Y consist of a sequence of sequential disk pages. While almost all the I/O operations in phase 1 access sequential disk pages, most of the I/O operations in phase 2 are random. To increase the number of sequential disk accesses, we can slightly modify the multiway merging procedure by increasing the sizes of the input buffers and the output buffer from one disk page to a larger number of disk pages. Whenever an input buffer is exhausted during multiway merging, instead of doing one random disk-read operation, we do one random and a number of sequential disk-read operations. If the number of runs after phase 1 is much smaller than $m - 1$, we can do this even without increasing the total number of I/O operations. (Exercise: think why this is true. How much can we icrease the size of the input buffers without increasing the total number of I/O operations?).

References

[1] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* 2nd edition, MIT Press (2001)

[2] D. E. Knuth. *The Art of Computer Programming: Volume 3 (Sorting and Searching).* 2nd edition, Addison-Wesley (1998)

[3] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The complete Book.* International edition, Prentice Hall (2002).