6.897: Advanced Data Structures Spring 2003

Lecture 17 — April 28, 2003

Prof. Erik Demaine *Scribe: Jonathan Brunsman, Glenn Eguchi*

1 Overview

This lecture will cover sorting in the cache-oblivious world. Sorting seems like an unusual topic for a data structures course, but as we will see, the results of our discussion of cache-oblivious sorting will lead to our development of cache-oblivious priority queues. We first review external-memory sorting before moving on to cache-oblivious sorting, priority queues, and Funnel Heaps.

1.1 Notation

This lecture uses capital letters in our analysis. We choose this notation because some papers use the notation $n = \frac{N}{B}$ and $m = \frac{M}{B}$, where N is the number of elements, M is the size of the cache, and B is the block size. This notation is confusing so we will avoid it.

2 External-Memory Sorting

First, recall from last class that we obtained a bound of $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ in memory transfers for sorting in the comparison model. We obtained this bound by using an $\frac{M}{B}$-way merge sort. In this lecture we will refer to that bound as the *sorting bound*, because there is a matching lower bound for sorting in the comparison model [2]. The logarithmic base $\frac{M}{B}$ and the multiplicative constant $\frac{N}{B}$ were significant because they were departures from our idea of classic merge sort, which was bounded by $O(n \lg n)$.

Another observation to keep in mind throughout the discussion of sorting is that if we want a data structure capable of sorting N elements and require that its performance match the sorting bound, we will need its performance to be $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ per element.

3 Cache-Oblivious Sorting

As we begin our discussion of cache-oblivious sorting, note that the algorithms and data structures we have already seen have not depended on the parameter M, the cache size. Our sorting algorithm, however, will depend on M, thereby increasing the complexity of our analysis.

The cache-oblivious sorting algorithm presented here will be a version of funnel sort [8, 5], which is similar to merge sort. There are two principle versions of this sort: an aggressive funnel sort, developed first [8], and a lazy funnel sort, developed recently [5]. We give the latter, simpler description.

3.1 Tall-Cache Assumption

In our previous discussions of cache-oblivious data structures, we required that the quotient $\frac{M}{B}$ must be at least 1 so that we can store something useful in the cache. This assumption is rather weak and in this lecture we shall replace it with a stronger assumption, called the *tall-cache assumption*, which will be useful in cache-oblivious sorting. In fact, Brodal and Fagerberg [6] proved that a tall-cache assumption is necessary to achieve the sorting bound via a cache-oblivious algorithm. This assumption generally states that the cache is taller than it is wide. More precisely, the tall-cache assumption requires that $M = \Omega(B^{1+\gamma})$ for some constant $\gamma > 0$.[1] In practice, a common cache size is $M = \Omega(B^2)$ (i.e., $\gamma = 1$), and we will make this stronger assumption for simplicity in the algorithms and data structures below.

3.2 K-Funnel

The basic approach of Funnel Sort is similar to that of a merge sort, and the overall goal will be to develop a fast merger process. If we can merge quickly in the cache-oblivious world, then we can sort quickly as well.

A *funnel* merges several sorted lists into one sorted list in an output buffer. Similarly, a K-funnel merges K sorted lists into one sorted list. Each of these lists be large so that the total size is at least K^3. The K-funnel sorts these K lists using $O(\frac{K^3}{B}\log_{\frac{M}{B}}\frac{K^3}{B} + K)$ memory transfers. Note that the additive term K is required because we must look at each list at least once. Note also that, while the merging process in a typical merge sort requires linear time, funnel merging will require the logarithmic term. This difference stems from counting memory transfers, and will become apparent at the end of our analysis of Funnel Sort.

The K-funnel is a static data structure, defined recursively in terms of $\sqrt{K}$-funnels; see Figure 1. The K-funnel has an output buffer of size K^3 and K input sources of possibly varying size but total size at least K^3. Within the K-funnel are two levels of $\sqrt{K}$-funnels separated by middle buffers of size $K^{3/2}$. There are $\sqrt{K}$ middle buffers (each storing the output of one lower $\sqrt{K}$-funnel), so the total buffer space separating the $\sqrt{K}$-funnels is K^2.

3.3 Lazy Merge

The lazy merge process fills the middle buffers at the subsequent level within the K-funnel. The algorithm for lazy merge is simple: if the two lower-level buffers both have data, copy that data and do a binary merge on the two buffers. This procedure is very compact since we are only considering three specific buffers in memory: the two input buffers and the resulting, merged buffer.

When one child buffer is empty, however, we must recurse to fill that buffer. As a base case, we define the input to the leaves as the input lists to sort. If that input is empty then we shall indicate it through the use an EOF or other such marker.

The difficult part of the lazy merge process is the analysis. The recursive layout will be similar to that used in van Emde Boas structures but now we must include the middle buffers between

[1] We use the letter γ because it is the third letter of the alphabet, standing for "cache".

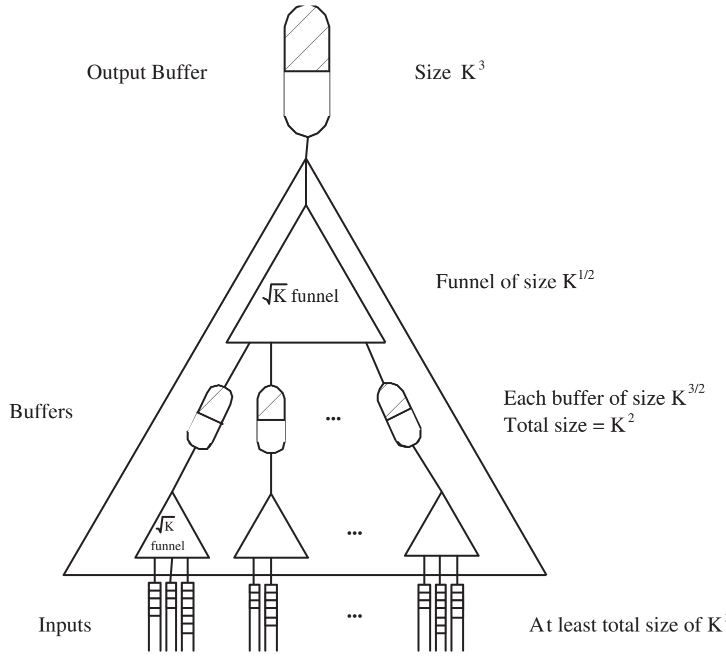

Figure 1: A K-funnel

funnels in our analysis. This requirement is not problematic as long as we maintain the funnels in consecutive memory.

3.4 Analysis of Lazy Merge

We begin our analysis of lazy merge by examining its space requirement. First, let's consider when the funnels do and do not fit inside the cache by examining the space required by the funnels:

$$S(K) = (\sqrt{K} + 1)S(\sqrt{K}) + K^2$$

The three terms of the recurrence are easily derived. First, the number of recursions is $\sqrt{K} + 1$. Second, each of these recursions costs $S\sqrt{K}$. Finally, the size of the buffer is K^2 (not K^3 since we do not count the final output buffer).

We solve this recurrence using recursion trees. Since there are $O(K)$ leaves, each requiring constant space, the root cost of K^2 clearly dominates and the recurrence simplifies to $O(K^2)$. More precisely, we shall say that $S(K) = cK^2$ for some constant $c \geq 1$.

The analysis of the time requirement for lazy merge proceeds similarly to that of the van Emde Boas structures. We recurse to a specific level, J, chosen such that the size of a J-funnel is $< \frac{M}{4}$ and the size of a J^2-funnel is $\geq \frac{M}{4}$. In other words, J is the largest level that has funnels of size smaller than $\frac{M}{4}$.

$$\text{Size}(J\text{-funnel}) = cJ^2 < \frac{M}{4} \text{ but } \text{Size}(J^2\text{-funnel}) = cJ^4 \geq \frac{M}{4}$$

Next, we examine how one J-level behaves. The merging process requires us to access a J-funnel, so let's fix its size at $\sqrt{K}$. The size of a J-funnel is K, and the space required for the input buffers is K^2. These input buffers are quadratically bigger than the funnel size! How do we store these large buffers?

The answer is to store just one block from each buffer, rather than store the entire buffer. If there are J buffers, we only need store just JB:

$$JB \leq \sqrt{\tfrac{M}{4c}}\sqrt{M} = \tfrac{M}{\sqrt{4c}} \leq \tfrac{M}{2}$$

Now we must relate the parameters B and M, which we can do through the use of the tall-cache assumption:

$$M = \Omega(B^2) \text{ and } cJ^4 \geq \tfrac{M}{4} \geq \Omega(B^2)$$
$$\geq J^2 = \Omega(B)$$

Another approach is to observe that the product JB is the geometric mean of J^2B^2:

$$JB = \sqrt{J^2B^2} \leq \sqrt{\tfrac{M}{4}} * \sqrt{M} = \sqrt{\tfrac{M^2}{4}} = \tfrac{M}{2}$$

We could also have used the golden ratio $\frac{1+\sqrt{5}}{2}$ in place of 4, to get slightly better constants, but for simplicity we'll stick with 4.

Now that we have an expression for the size of one J-funnel in memory, what happens when we are forced to load a J-funnel? Reading an entire J-funnel + 1 block per input stream costs $O(\frac{J^2}{B} + J) = O(\frac{J^3}{B})$. This result follows since J^2 is at most M and M is at least B^2. More specifically:

$$J^4 = \Omega(B^2) \leq J^2 = \Omega(B)$$

Multiplying both sides by $\frac{J}{B}$:

$$\tfrac{J^3}{B} = \Omega(J)$$

The recursion process could lead to cache swaps, possibly causing everything stored within the cache to be overwritten. Thus, when an input buffer empties and is recursively filled, the previous J-funnel will be erased from the cache. Recall that the buffers are the cube of the size of the funnel above them, allowing us to charge the cost, $\frac{J^3}{B}$, to reload that funnel to the J^3 elements that will be filled by the recursive call. Essentially, the amortized cost of loading the J-funnel back into memory is just $\frac{1}{B}$ per element. Thus, we have linear time, $\frac{N}{B}$, merge.

This result seems too fast! In fact, it is. Our argument is basically correct but we forgot to count the number of times each element is charged the $\frac{1}{B}$ loading cost. How many times does each element get charged at level J? Just once: each element visits each level once as it moves up through the funnels, and no element ever moves down a level. The height is $\log_J K$, so each element gets charged $\log_J K + O(1)$ times.

Taking the loading cost into account yields:

$$\leq \frac{\lg K}{\lg J} + O(1) \leq \frac{\lg K}{\frac{1}{4}\lg M} + O(1) = O(\frac{\lg K}{\lg M})$$

On the second step we can simplify the $\frac{M}{4}$ term to just M by adding the O(1) term. Thus, the amortized cost per element is:

$$\frac{1}{B}O(\frac{\lg K}{\lg M}) = O(\frac{1}{B\log_M K})$$

This result seems more reasonable, but it is still not our goal of $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{K}{B})$. We claim that neither the $\frac{M}{B}$ nor the $\frac{K}{B}$ terms really matter and justify this claim through use of the tall-cache assumption. This is easy to show if $K \geq B^2$ but more difficult to show if $K \leq B^2$.

With the use of the tall-cache assumption, we have arrived at a fast merge procedure. Now let's move on to the sort.

4 Funnel Sort

We can use K-funnels to construct a cache-oblivious sort, *Funnel Sort.* Funnel Sort runs as follows:

1. Conceptually split the elements into $N^{1/3}$ segments of length $N^{2/3}$ each.
2. Call Funnel Sort recursively on each segment.
3. Merge the sorted segments into the output stream using an $N^{1/3}$-funnel.

The base case occurs if $N < O(B^2)$, where, by the tall-cache assumption, we can move the entire list into the cache and sort in $O(B)$ time.

4.1 Analysis of Funnel Sort

Lemma 1. *Funnel-Sort requires* $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ *memory transfers.*

Proof. Splitting the elements into segments is purely conceptual and requires no transfers. Let $T(N)$ equal the number of transfers that Funnel Sort requires. Since each segment is length $N^{2/3}$, sorting the segments requires $N^{1/3}T(N^{2/3})$ transfers. Merging the segments with an $N^{1/3}$-funnel requires $O(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}+N^{1/3}))$ transfers. We can now set up the following recursion for the number of transfers:

$$T(N) = N^{1/3}T(N^{2/3}) + O(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B} + N^{1/3}))$$

The base case occurs when $N = O(B^2)$, in which case we can copy the entire list into the cache and sort it in $O(B)$ time:

$$T(O(B^2)) = O(B)$$

There are a total of $\frac{N}{B^2}$ leaves in the recursion tree with a cost of $O(B)$ each. The total leaf cost is therefore:

$$\frac{N}{B^2}O(B) = O(\frac{N}{B})$$

Recall that the root cost is $O(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B} + N^{1/3}))$. Using the fact that for non-leaves $N = \Omega(B^2)$, we can simplify this expression:

$$\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} = \Omega(\sqrt{N}\log\sqrt{N}) = \Omega(N^{1/3})$$

Thus, the root cost is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$. The root dominates the recurrence, and therefore the total memory transfers required by Funnel Sort is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

5 Priority Queues

As described, we will also use the K-funnels to build a cache-oblivious priority queue. First, let's examine work that has been done in the external-memory model.

5.1 External Memory

Efficient priority queues have been developed for the external memory model. For example, buffer trees can be used to implement priority queues[3]. A buffer tree can be view as a modified B-tree with nodes grouped into chunks and a buffer of updates associated with each chunk. When the update buffer for a chunk fills, the calls are pushed down to the chunks of lower levels. Thus, updates are made to the tree in batches. Using buffer trees, we can perform Insert, Delete, and *Delayed-Search* in $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ amortized memory transfers. Delayed search answers a query but perhaps at a later time (for example, multiple searches may be batched).

5.2 Cache-Oblivious

In the cache-oblivious model, Arge, Bender, et al. accomplished Insert, Delete and Delete-Min in $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ amortized memory transfers [4]. Their method, however, required sorting as a subroutine. Ideally, we would like a data structure which implemented priority queues more directly.

5.3 Funnel Heap

Fortunately, Brodal and Fagerberg also successfully created a cache-oblivious priority queue that did not use sorting as a subroutine [7]. Their data structure, *Funnel Heaps*, executes Insert and Delete-Min (but not Delete) in $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ amortized memory transfers.

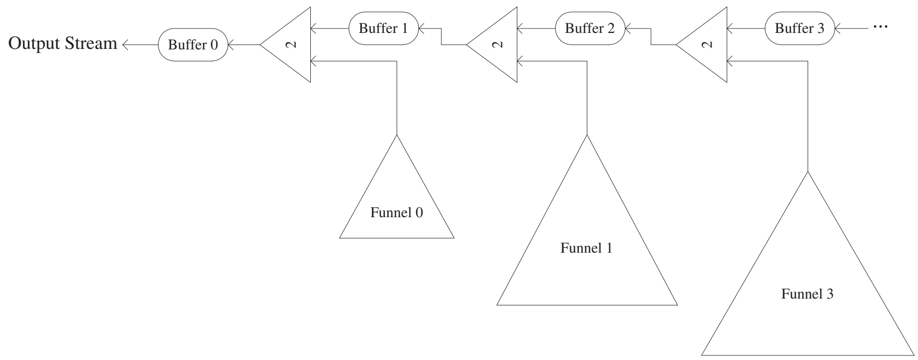

Figure 2: A Funnel Heap

A Funnel Heap is composed of a series of interconnected funnels, buffers, and 2-funnels. The funnels and buffers and arranged in the following manner:

At the output of the funnel heap is buffer 0. At the input of buffer 0 is a 2-funnel which merges the output of funnel 0 and buffer 1. At the inputs of funnel 0 are some of our elements. At the input of buffer 1 is a 2-funnel which merges the outputs of funnel 1 and buffer 2. Funnel 1 also contains some of our elements and is larger than funnel 0. At the input of buffer 2 is a 2-funnel which merges the output of funnel 2 and buffer 3.

The structure continues such that the input of buffer i is a 2-funnel which merges funnel i and buffer $i+1$. The funnels are of doubly increasing size where the size of funnel $i = 2^{(4/3)^i}$.

The invariant holds that the elements in buffer i are less than or equal to the elements in funnel i and the elements in buffer $i+1$. Therefore, buffer 0 always stores the minimum elements.

5.4 Delete-Min

Assuming the invariant holds, the Delete-Min operation is straightforward: if buffer 0 is not empty, extract the first element from buffer 0. Otherwise (if buffer 0 is empty), fill in buffer 0 using the 2-funnel connected to its input.

5.5 Insert

Maintaining the invariant throughout the insert procedure is a bit more involved. The key is to allow funnels to have missing input streams. When we first create the data structure, the input streams to all of the funnels are missing. Suppose after several inserts, some funnels have non-empty input streams. Lets call funnel i the first funnel with a missing input stream. To insert an element into our data structure:

1. First move the elements from buffers 0 through i and the buffers in the root-to-empty-input-stream path of funnel i in order into an auxiliary buffer. Empty the buffers as we do so. Because of our invariant, the resulting list in our auxiliary buffer is sorted.

2. Insert into a second auxiliary buffer the elements contained in funnels 0 through i in sorted order. This can be accomplished by repeatedly calling Delete-Min.

3. Merge the two auxiliary buffers.

4. Fill buffers 0 through i with elements from our new merged list. Fill the buffers as full as they were prior to the insert.

5. Fill the root to leaf path to funnel i.

6. Place the remaining elements into the empty input stream for funnel i.

The majority of the elements will be placed into the empty input stream. The input stream of funnel i is large enough to store the elements because of the doubly increasing size of our funnels.

Notice that following Insert, funnels 0 through $i-1$ are empty. Thus it will require many inserts before an input stream of funnel i is the first empty stream. Therefore, the amortized cost of a single insert is:

$$O(\sum_{j=0}^{\infty} \frac{1}{B} \log_{\frac{M}{B}} N^{(3/4)^j}) = O(\frac{1}{B} \log_{\frac{M}{B}} N).$$

6 Open Problems

Throughout our discussion of sorting, we have relied upon the comparison model. The comparison model is a fairly weak assumption. Essentially, it describes what we can do with two elements. If we have M elements in the cache, the best we can do is derive some ordering of those M elements.

What if we were given a permutation of the elements, and simply need to execute it, instead of also needing to find the sorted permutation? The permutation bound, $\Theta(\min(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}, N)$ is a lower bound on the amount of time it would take us to arrange our N elements in the given order [2]. Thus, even if we are given the correct order, we cannot do much better than sorting.

Can we do better outside the comparison model? This model says in particular that we cannot divide the elements: we cannot share bits, splice elements, separate different bits, or perform other dividing operations. Can we do better if the elements are divisible, say, they are integers and can be split into their constituent bits. This divisible model was used successfully for other problems; see [1].

Lastly, in our discussion of external-memory priority queues, we mentioned a priority queue can perform *Delayed-Search* in $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ amortized memory transfers. Can cache-oblivious data structures also take advantage of delayed search?

References

[1] Micha Adler. New coding techniques for improved bandwidth utilization. In *Proc. 37th IEEE Symposium on Foundations of Computer Science*, 1996.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31.9:1116–1127, 1998.

[3] Lars Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract).

[4] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 268–276. ACM Press, 2002.

[5] Gerth Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*, 2380:426–438, 2002.

[6] Gerth Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003.

[7] Gerth Stolting Brodal and Rolf Fagerberg. Funnel heap - a cache oblivious priority queue. *ISAAC*, pages 219–228, 2002.

[8] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *FOCS*, pages 285–298, 1999.