Chapter 4

Cache-Oblivious Sorting Algorithms

As Frigo *et al*. presented the novel model of cache-oblivious computing they also presented a host of optimal cache-oblivious algorithms and data structures [FLPR99]. Aside from Fast Fourier Transformation, Matrix Multiplication, and Matrix Transposition they presented two optimal sorting algorithms. Both are in essence cache-oblivious variants of the two sorting algorithms presented in the previous chapter. What follow is a thorough presentation of cache-oblivious merge sort, dubbed funnelsort.

4.1 Funnelsort

The cache-awareness of multiway merge sort is twofold; initial runs are of size M, and in merges runs with an M/B-merger.

In multiway mergesort, we could simply implement the merger using a binary heap, or an equivalent, as a priority queue; but without knowing M or B, we do not know what size it should be, nor do we know if it can even fit in cache. If a binary heap, for instance, cannot fit in cache, its I/O performance decreases drastically. Hence, we need a merger structure that, no matter what M is, can merge efficiently. The key to this is, as with binary searching, recursion, albeit in a slightly different form. However, simply adapting the layout of a binary heap turns out to be insufficient; indeed constructing an I/O optimal heap is non-trivial. Fortunately, we do not need to go so far.

4.1.1 Merging with Funnels

The cache-oblivious equivalent of a multiway merger is a *funnel*. A k-funnel is a static data structure capable of merging k sorted input streams. It does so by merging k^d elements at a time, for some $d > 1$. It is arranged as a tree, with two notable features: it is stored recursively according to the van Emde Boas layout, with the top and bottom trees them selves being (sub-) funnels; secondly, along each edge, it keeps a buffer. Intuitively, we need such buffers to amortize the cost of using a (sub-) funnel, in case it is so big it cannot be operated with a constant number of memory transfers.

Despite it being a rather young data structure, the funnel has already undergone a couple of changes. After its introduction in 1999 along with funnelsort, Brodal and Fagerberg [BF02a] presented a conceptually simpler version, dubbed lazy funnelsort, and in this thesis, we will introduce further simplifications. It seems natural to describe the original funnel and then introduce the modifications, to establish an intuition of why the structure is optimal. The analysis will follow the final modifications.

The Eager Funnel

The original k-funnel[1] merged k^3 elements at a time. It consists of $\sqrt{k}$ $\sqrt{k}$-funnels L_0, L_1, …, $L_{\sqrt{k}-1}$, at the bottom, each connected to a $\sqrt{k}$-funnel R, at the top, with edges, that contain buffers, B_0, B_1, … $B_{\sqrt{k}-1}$. We will discuss the case of non-square orders below. The buffers doubles as both the output of the bottom funnels as well as input for the top funnel. Figure 4-1, shows a 16-funnel consisting of five 4-funnels and four buffers. At the base of the recursion, at the nodes of the tree, we have constant sized binary or ternary mergers. Note, that a k-funnel in it self does not have input or output buffers, though it may sometimes be convenient to conceptually include e.g. an output buffer in a funnel. The funnel is stored at contiguous locations in memory, according the van Emde Boas layout: first, the top tree is laid out recursively, immediately thereafter comes B_0 followed by a recursive layout of L_0 followed by B_1 and so on.

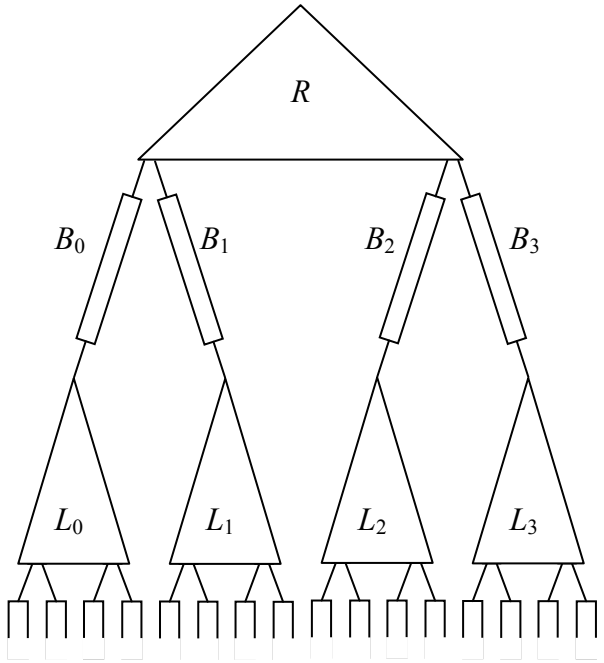

Figure 4-1. A 16-funnel with its 16 input streams.

To operate a funnel, we use a recursive procedure invoke, illustrated in Algorithm 4-1. It reads a total of k^3 elements form the funnel's sorted input streams, and output k^3 elements in sorted order. For the amortization argument to work, there must initially be at least k^2 elements in each input buffer, but as the merging progresses, there will be fewer and fewer elements in the input. If at some point a (sub-) funnel cannot fulfill the obligation to output k^3 elements, due to insufficient input, it outputs what it can, is marked exhausted, and will not be invoked again. This is done to avoid futile descends into bottom funnels, that may not produce enough elements. There will be at most one invocation, per funnel, that outputs less k^3 than elements, so it will not influence the asymptotic analysis.

[1] ... was originally called a k-merger, but since a merger, at least in my mind, is a broader class of data structures, I will use the term k-funnel for this kind of merger.

Algorithm 4-1. invoke(k-Funnel F)

```
if k is 2 or 3 then
    merge "manually"
else
    repeat k^3/2 times do
        for each buffer B_i, 0 ≤ i < k do
            if B_i less than half full and L_i is not exhausted then
                invoke(L_i)
        invoke(R)
    od
fi
if less than k^3 elements was output then
    mark F exhausted
```

The recursion of invoke follows the layout; when a funnel is invoked, it invokes the top funnel $k^{3/2}$ times, each time contributing $k^{3/2}$ elements to the output, for a total of k^3 elements. Before each invocation, however, it checks whether there are enough elements in the buffers (the input of the top funnel) and for each buffer with less than $k^{3/2}$ elements, it invokes the bottom funnel with that particular buffer as its output. For this to work, the buffers are implemented as FIFO queues and have a capacity of at least $2k^{3/2}$. In Figure 4-1, each of the four buffers has a capacity of 128 elements. This will insure that if there are at most $k^{3/2}$ elements there is room for an additional $k^{3/2}$, while at the same time guarantee that there will be enough elements to output $k^{3/2}$ elements, even in the extreme case the elements to be output all come from the same input stream.

It can be proven (see Section 4.1.2 below) that a k-funnel takes up $\mathcal{O}(k^3)$ contiguous memory locations. So, intuitively, the reason the funnel works is that we may have to "pay" the memory transfers to get it going (about $\mathcal{O}(k^2/B)$), but we get $\Omega(k^3)$ elements out of it (paying a total price of $\Omega(k^3/B)$). Additionally, if the funnel is too big to fit in cache, regardless of its size, there will be some level of the van Emde Boas recursion, at which the subfunnel will fit in cache. For that level, the previous argument applies, and as it turns out, there are not too many levels.

The above description may seem as a simple way of merging; however, there a couple of inherent practical problems. Most severe is perhaps the subtle dependencies between the buffer sizes and the order of both the top and bottom funnels. Given that we in practice cannot split a k-funnel into $\sqrt{k}$ -funnels, simply because k may not be square, we run into rounding issues: The standard way of avoiding rounding problems would be to choose k as the smallest order that ensures only square funnels throughout the funnel. This order however is one that gives the tree a power-of-two height, that is, it needs to be of order power-of-power-of-two, which would make the funnel too big in the worst case. So a k-funnel is in general comprised of j $\left\lceil\sqrt{k}\right\rceil$-funnels and $\left(\left\lceil\sqrt{k}\right\rceil - j\right)\left\lfloor\sqrt{k}\right\rfloor$-funnels (perhaps one less) aside from the top funnel, for some $j \leq \left\lceil\sqrt{k}\right\rceil$. Now the buffers may be too small either to hold the output of a bottom funnels or to ensure enough elements for the top funnel. A solution to this problem is, after the order of the top and bottom funnels have been determined, the i'th buffer should be of size $2m_i^{3/2}$, where m_i is at the maximum of the order of the i'th bottom funnel and the top funnel. But now a single invoke of a bottom funnel will not insure, that the buffer is at least half filled, so the algorithm need to sometimes do more than a single invoke.

In addition, the apparent simplicity of the merging phase may be deceiving; we have explicitly made certain prior to the invocation of any funnel, that there will be enough elements in the input to produce the output and may merge k^3 elements, without checking the state of the buffers. This is false. The issue of exhaustion cannot be controlled, because it depends on the input, so any subfunnel may at any time become exhausted and its output buffer only be partially filled. The funnel that uses this buffer as input has to consider this, in effect making it paranoid, always checking to see if there indeed are more elements.

Aside from these concerns, there is the practical issue of doing the actual layout. While doing a van Emde Boas layout of a binary tree with fixed-sized and -typed nodes may be trivial, doing it with variable sized buffers and mixed data types is not. Suffice it to say, that on some hardware architectures, certain data types cannot be placed at arbitrary memory locations. This is known as alignment. For example, a double precision floating-point number may only be addressed, if it is placed at addresses divisible by eight, while an integer can be at any address divisible by four. If we were to merge doubles with a funnel, we would have to lay out buffers capable of containing doubles intermixed with nodes containing pointers and Booleans. Say a node consists of six pointers of four bytes each and a byte for the exhausted flag, for a total of 27 bytes. Now we cannot simply place a buffer, a node, and then another buffer contiguously. Simply using four bytes for the exhausted flag does not help. To make matters worse, to our knowledge, no language supports a simple way of determining alignment requirements of data types.

The Lazy Funnel

Many of these issues were eliminated with the introduction of the *lazy funnel* [BF02a]. The modification lies primarily in the operation of the funnel. The lazy funnel generates the output of bottom funnels lazily, in the sense that they are invoked as needed, when there are no more elements in the buffers, as opposed to explicitly checking the state of the buffers and then perhaps invoking a bottom funnel. This means, in turn, that there is no longer operationally a need to follow the layout in the recursion, and thus no need for a notion of top and bottom funnels; all base funnels (the binary and ternary mergers) are invoked the same way, through a procedure called `lazy_fill`, illustrated, for a binary node, in Algorithm 4-2. Note that if one of the children is exhausted, the corresponding input may not contain any elements so the head of that input may be undefined and should be ignored or taken to be the largest possible element.

Algorithm 4-2. lazy_fill(Node v)

```
while v's output buffer is not full do
    if left input buffer empty and left child of v not exhausted then
        lazy_fill(left child of v)
    if right input buffer empty and right child of v not exhausted then
        lazy_fill(right child of v)
    if head of left input < head of right input then
        move head of left input to output
    else
        move head of right input to output
od
if left input buffer empty and left child of v exhausted and
        right input buffer empty and right child of v exhausted then
    mark as v exhausted
```

This allows for two structural simplifications: the lazy funnel is now simply a balanced binary tree, no mix of binary and ternary mergers. Secondly, there are no subtle dependencies between the size of the buffers and the funnels using them as input or output. The storing of the funnel is still done according to the van Emde Boas layout as are the capacity of the buffers still a function of the order of the subfunnel that has it as its output. The shape of the function, however, now only has an analytical significance, so we are free to choose from a larger class of functions. The analysis was originally carried out with k-funnels generating output of size k^d, with $d \geq 2$.

These simplifications have since paved the way for the use of the funnel data structure as a key element in many cache-oblivious algorithms and data structures, such as the distribution sweeping class of geometrical algorithms [BF02a] and the funnel heap [BF02b]. The many uses of the funnel underline the need for good solid implementation.

Two-Phase Funnel

The practical issues that remain in the lazy funnel are the special attention to the exhaustion flag and the complex layout. The *two-phase funnel* resolves these issues. As for the latter, it is simply not needed; the two-phase funnel does not care about the layout. As with the lazy funnel, the van Emde Boas recursion is still used for determining the capacity of the buffers, so the layout of the original funnel structure is not completely gone. With the two-phase funnel, the use of controlled layout is optional. Whether it influences performance in practice will be investigated in Chapter 6.

As for the former, we first look at scheduling of the nodes. It would seem that the merging proceeds until either the output is full or *exactly one* of the node's inputs is empty, in which case lazy_fill is called on the corresponding child. There are exactly two cases, where this is not true. One is in the initial funnel, where all buffers are empty. Invoking lazy_fill on the root will not merge until one input is empty, simply because both are empty. The other case is when the last time lazy_fill was called on it, the node below an input, was exhausted and could thus not produce any elements. This is important, since it gives an indication of when a merger may be exhausted; we will thus maintain the invariant that *as we start filling, buffers that are empty, are the output of exhausted mergers, and should thus be ignored*. This is not true, however, in the initial funnel, but that can be fixed by introducing a special warm-up phase. This phase is the

part of the merging that takes place before the first call to lazy_fill starts outputting elements. It consists essentially of head-recursive calls to lazy_fill, as Algorithm 4-3 illustrates.

Algorithm 4-3. warmup(Node v)

```
if v not leaf then
    warmup(left child of v)
    warmup(right child of v)
fi
fill(v)
```

If, in case all input streams are empty, the invocation of fill simply does nothing and returns, the invariant holds; warmup guarantees that fill is called on nodes, only after fill has been called on all the children. Leaf nodes are exhausted if and only if no elements are in the input. If so, fill will do nothing and return, thus leaving an empty input buffer for the parent. This in turn will be taken to mean that the leaf is exhausted. Inductively, if all children of an internal merger have returned with empty buffers, they are all exhausted, and since no elements are in the buffers, the internal merger is exhausted. The same argument applies in the second phase, when elements are actually output from the root of the funnel. Note that the scheduling of nodes are the same as in the lazy funnel, thus this is a mere algorithmic simplification and not an operational one. The modified version of lazy_fill is illustrated in Algorithm 4-4. Note, also, the potential performance gained through the stronger invariant. While Algorithm 4-2 evaluates four conditional branches per element merged, Algorithm 4-4 only evaluates three. In addition, it explicitly needs to check the exhaustion flag.

Algorithm 4-4. fill(Node v)

```
if both inputs are non-empty then
    while v's output buffer is not full do
        if head of left input < head of right input then
            move head of left input to output
            if left input buffer empty then
                fill(left child of v)
        else
            move head of right input to output
            if right input buffer empty then
                fill(right child of v)
        fi
    od
else if exactly one input buffer empty then
    move as many elements as possible from the other input to the output
    if input buffer got empty then
        fill(corresponding child of v)
else
    return
fi
```

Generalization of base nodes from binary to multiway is now also straightforward; the invariant holds as long as fill simply returns, if no stream contain elements and it only considers non-empty streams for merging. Using z-way base mergers corresponds

to stopping the van Emde Boas recursion before we reach trees of height one; stopping at e.g. height two would yield a funnel with four-way base mergers. In the analysis that follows, we will consider the use of z-way base mergers, as well as a slightly larger class of buffer size functions, namely αk^d with α and d being constants and k the order of the funnel below the buffer.

4.1.2 Funnel Analysis

In this section, we analyze the complexity in the cache-oblivious model of filling the output buffer of a funnel, using a two-phase funnel. First, the total size of the funnel needs to be bounded; this is important for determining when a funnel fits in cache. For this, we follow the van Emde Boas recursion. Recall that the van Emde Boas recursion is actually a horizontal split of the tree at depth half the height h, the order of a funnel being z^h, so the size of the output becomes αz^{hd}.

Lemma 4-1. Assuming $d \geq 2$ and $k = z^h$ for some $h > 0$, a k-funnel spans at most $S(k) \leq \gamma k^{\frac{d+1}{2}} B^{-1} + 4k$ blocks, with $\gamma = 2.00033 \frac{\alpha z^{d+1}}{1 - z^{-2}}$.

Proof. First, consider the number of blocks needed to hold the buffers. A buffer is an array of $\beta(h) = \alpha z^{hd}$ elements and occupies as such no more than $\beta(h)/B+2$ blocks, the extra two being in the case of the ends reaching into parts of other blocks. For now, we assume that buffers take up $\beta(h)/B$ blocks, and will compensate later. A funnel of height h has by convention of this thesis a top tree of height $\lfloor h/2 \rfloor$[1]. Aside from the buffers in the top and bottom funnels, such a funnel has at most $N(h/2) = z^{h/2}$ buffers. Each of these has a capacity determined by the bottom funnel, namely $\beta(\lceil h/2 \rceil) \leq \beta((h+1)/2)$, since the bottom funnel has height $\lceil h/2 \rceil$. Correspondingly, there are at most $N((3h+1)/4)$ buffers in the bottom funnel and $N(h/4)$ in the top funnel. The recursion continues in this way, down to funnels of height one, so the total number of blocks used by buffers of a funnel of height h is

$$\begin{aligned} s(h) &\leq N(h/2)\,\beta\left(\tfrac{h+1}{2}\right)B^{-1} + \left(N\left(\tfrac{3h+1}{4}\right) + N(h/4)\right)\beta\left(\tfrac{h+3}{4}\right)B^{-1} \\ &\quad + \left(N\left(\tfrac{7h+5}{8}\right) + N\left(\tfrac{5h+3}{8}\right) + N\left(\tfrac{3h+1}{8}\right) + N(h/8)\right)\beta\left(\tfrac{h+7}{8}\right)B^{-1} + \cdots \qquad (4.1) \\ &\leq \sum_{i=1}^{\log h}\left[\beta\left(2^{-i}h+1\right)B^{-1}\sum_{j=1}^{2^{i-1}}\left(N\left(2^{-i}h(2j-1)+2j\right)\right)\right] \end{aligned}$$

Using the bound

$$\sum_{j=0}^{n-1} x_i^{\,j} = \frac{x_i^{\,n}-1}{x_i-1} = \frac{x_i^{\,n}-1}{x_i\left(1-1/x_i\right)} \leq \frac{1}{1-\max_i\left(x_i\right)^{-1}}\, x_i^{\,n-1}, \qquad (4.2)$$

we get the bound of the inner sum

[1] [BDFC00] uses the convention that search trees of height h have bottom trees of height $\lceil h/2 \rceil$. However, this rounding could lead to very large buffers just below the root, when the tree has height 2^j+1 for some integer j.

$$\begin{aligned}\sum_{j=1}^{2^{i-1}} N\left(2^{-i}h(2j-1)+1\right) &= \sum_{j=1}^{2^{i-1}} z^{2^{-i}(h(2j-1)+2j)} \\ &= z^{-2^{-i}h}\sum_{j=1}^{2^{i-1}} z^{2^{1-i}(h+1)j} \\ &\le z^{-2^{-i}h}\left(\frac{1}{1-z^{-2}}z^{2^{1-i}(h+1)2^{i-1}}\right) \\ &\le \frac{z}{1-z^{-2}}z^{\left(1-2^{-i}\right)h}\end{aligned} \tag{4.3}$$

and the outer sum becomes

$$\begin{aligned}s(h) &\le \sum_{i=1}^{\log h}\frac{z}{1-z^{-2}}z^{\left(1-2^{-i}\right)h}\beta\left(2^{-i}h+1\right)B^{-1} \\ &= \frac{z}{1-z^{-2}}\sum_{i=1}^{\log h} z^{\left(1-2^{-i}\right)h}\left(\alpha z^{\left(2^{-i}h+1\right)d}B^{-1}\right) \\ &= \frac{z}{1-z^{-2}}\frac{\alpha}{B}\sum_{i=1}^{\log h} z^{\left(1-2^{-i}\right)h+\left(2^{-i}h+1\right)d} \\ &= \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(z^{h\frac{d-1}{2}}+\sum_{i=2}^{\log h} z^{h\left(2^{-i}(d-1)+1\right)}\right) \\ &\le \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(z^{h\frac{d+1}{2}}+z^{h\frac{d+3}{4}}\log h\right) \\ &= \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(1+z^{h\frac{1-d}{4}}\log h\right)z^{h\frac{d+1}{2}}\end{aligned} \tag{4.4}$$

With $d,z \ge 2$, the following bound has been estimated numerically:

$$z^{h\frac{1-d}{4}}\log h < 1.00033\,.$$

The term is actually bounded by a finite constant, for all $d > 1$. Realistically, as the d approaches 1, the term becomes bounded by $\log h$, which for $\alpha \ge 1$, $z \ge 2$ and the size of the output $\alpha z^{hd} < 2^{64}$, is bounded by 6.

Each buffer may extend onto two additional blocks and each node may span at most two blocks (assuming a block is big enough to hold at least one). Since there are no more than z^h buffers or nodes, nodes and buffer-ends can contribute with no more than $4z^h$ blocks, which completes the proof. □

Note, that for constant α and z, $S(k)$ is $\mathcal{O}(k^{(d+1)/2}/B+4k)$. The fact, that there is no need to layout the funnel in contiguous memory locations not only makes life easer on the implementer, it also provides for greater flexibility which is important when using the funnel in dynamic data structures, such as the funnel heap [BF02b].

Now we are ready to prove the I/O bound of fill.

Theorem 4-1. Assuming $M \geq (cB)^{(d+1)/(d-1)}$, for some $c > 0$, and the input streams contain a total of αk^d elements a k-funnel performs $\mathcal{O}(k^d \log_M(k^d)/B + k)$ memory transfers during an invocation of fill on the root, for fixed α and z.

Proof. For this proof, we also conceptually follow the van Emde Boas recursion, but this time only until we reach a subfunnel of order j, such that $\gamma j^{(d+1)/2} \leq \varepsilon M$, with ε being the solution to

$$\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} = \frac{c}{5}(1-\varepsilon). \tag{4.5}$$

For finite $c > 0$ and $d > 1$ we have $0 < \varepsilon < 1$. Figure 4-2 shows the recursion in case the k-funnel is too big to fit in cache.

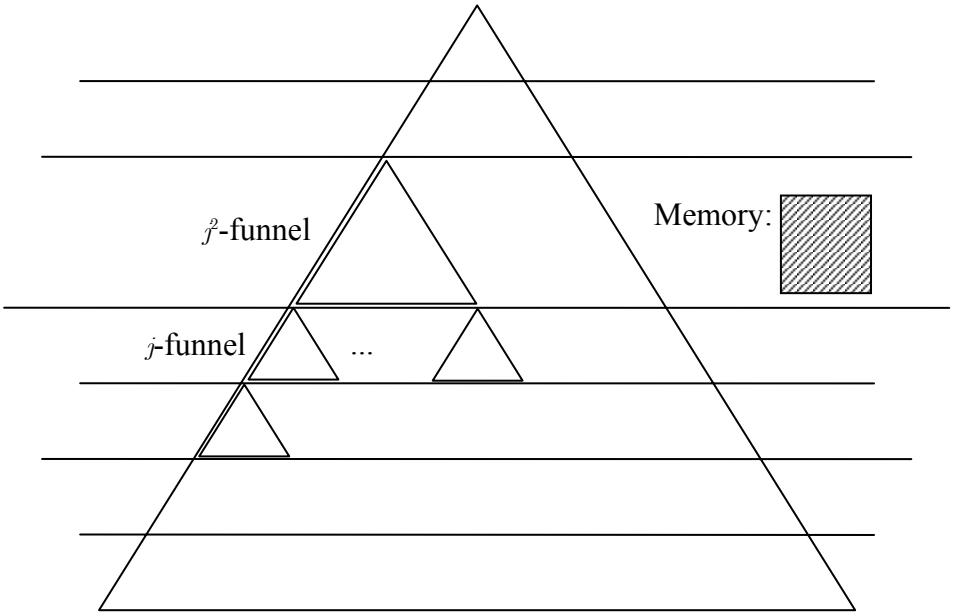

Figure 4-2. A k-funnel at the level of detail, where it consists of j-funnels. The shaded box indicates the relative size of memory.

A j-funnel has two important properties. First, the blocks spanned by a j-funnel, as well as a block from each of its input streams, all fit in cache. Second, if the entire k-funnel does not fit in cache, a j-funnel outputs at least a (positive) factor of B more elements, than there are input streams. The first property states that $S(j)+j \leq M/B$, which is proved using definition of j

$$\gamma j^{\frac{d+1}{2}} \leq \varepsilon M \Leftrightarrow 5j \leq 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M^{\frac{2}{d+1}} \tag{4.6}$$

and Lemma 4-1:

$$\begin{aligned} S(j) + j &= \gamma j^{\frac{d+1}{2}} B^{-1} + 5j \\ &\leq \varepsilon M/B + 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M^{\frac{2}{d+1}} \\ &= \varepsilon M/B + 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M \cdot M^{\frac{1-d}{1+d}} \\ &\leq \varepsilon M/B + \frac{5}{c}\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M/B \\ &\leq M/B \end{aligned} \tag{4.7}$$

The second property states that $\alpha j^d > ajB$, for some a. Essentially, this is a guarantee that the funnel will be able to pay the price of touching the input streams with the elements output. By definition of j, the subfunnel on the previous recursion level could not fit in cache. That funnel is of order at most zj^2 (the factor z being in case the j-funnel is the top funnel of a funnel of odd height) so we have

$$\begin{aligned} &\gamma\left(zj^2\right)^{\frac{d+1}{2}} > \varepsilon M \Leftrightarrow \\ &\gamma z^{\frac{d+1}{2}} j^{d-1} > \varepsilon^{\frac{d-1}{d+1}} M^{\frac{d-1}{d+1}} \Leftrightarrow \\ &z^{\frac{d+1}{2}} j^{d-1} > \frac{c}{\gamma} \varepsilon^{\frac{d-1}{d+1}} B \Leftrightarrow \\ &\alpha j^d > \frac{\alpha c}{\gamma} z^{-\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} jB \end{aligned} \tag{4.8}$$

Consider now an invocation of fill on the root of a j-funnel. Assuming enough elements in the input, this invocation will output at least αj^d elements. First, however, we must load the j-funnel and a block from each stream. This will cost at most $\gamma j^{(d+1)/2}/B + 5j$ memory transfers. Now the funnel can stay in cache and output the αj^d elements in a streaming fashion. When a block from an input stream has been used, the optimal replacement will read the next part of the stream read into that block; hence, the input will also be read in a streaming fashion. By the second property, the number of memory transfers per element output becomes at most

$$\begin{aligned} \frac{\gamma j^{\frac{d+1}{2}} B^{-1} + 5j}{\alpha j^d} + 2B^{-1} &\leq \left(\frac{\alpha}{\gamma j^{\frac{d-1}{2}}} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right) B^{-1} \\ &\leq \left(\frac{\alpha}{\gamma} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right) B^{-1} \end{aligned} \tag{4.9}$$

While there are a total of αk^d elements in the input of the k-funnel, and it therefore will not run empty during the merge, the input of all but the bottom-most j-funnels may run empty while it is in cache and actively merging. This may evict the funnel from memory, cause the j-funnel below the empty buffer to be loaded, and in turn reload the first funnel. However, at least αj^d elements have been merged into the buffer, and by the argument above, each of these elements may be charged the $(5\gamma/(\alpha c)z^{(d+1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers to pay what it costs to reload the funnel.

In total each element is charged $2(5\gamma/(\alpha c)z^{(d-1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers, per j-funnel it passes through. There are no more than $1+\log_j k$ j-funnels on the path from the input of the k-funnel to the root and since $\gamma z^{(d+1)/2} j^{d+1} > \varepsilon M$ implies $j > (M\varepsilon/\gamma)^{1/(d+1)}/z^{1/2}$, that number is bounded by

$$\begin{aligned}
1+\log_j k &= 1+\frac{\log k}{\log j} \\
&< 1+\frac{\log k}{\log\left(\frac{\varepsilon^{\frac{1}{d+1}}}{\gamma} z^{-1/2} M^{\frac{1}{d+1}}\right)} \\
&= 1+\frac{(d+1)\log k}{\log\left(\frac{\varepsilon}{\gamma z^{\frac{d+1}{2}}} M\right)} \\
&= 1+\log_{\frac{\varepsilon M}{\gamma z^{\frac{d+1}{2}}}} k^{d+1}
\end{aligned} \tag{4.10}$$

The total number of memory transfers caused by all elements going all the way through the k-funnel is then bounded by

$$\frac{\alpha k^d}{B} + u\,\frac{\alpha k^d}{B}\log_{\frac{\varepsilon M}{\gamma z^{\frac{d+1}{2}}}} k^{d+1} \tag{4.11}$$

with

$$u = 2\left(\frac{\alpha}{\gamma} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right). \tag{4.12}$$

When finally a j-funnel becomes exhausted, it may not have output αj^d elements and the elements them selves will not be able to pay for the memory transfers. However, the j-funnel is permanently marked exhausted and will thus not be invoked again. So we will only come up short once per j-funnel, and can thus charge the missing payment to the output buffer it self. Charging each position in the buffer $(5\gamma/(\alpha c)z^{(d-1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers, will account for might be missing, when the funnel below it became exhausted. However, since there are at most $\gamma k^{(d+1)/2}$ buffer positions in total, this is a low-order term. □

As for the work complexity of merging elements with a funnel, the following theorem holds:

Theorem 4-2. A k-funnel performs $\mathcal{O}(\alpha k^d z \log_z k)$ operations during an invocation of fill on the root.

Proof. During a fill on a given node, elements are moved from one level in the merger to a higher level. At each step in fill, the smallest element among the heads of the z buffers must be found and moved to the output. This costs at most z-1 comparisons and one move and the result is that the element is at a higher level. The merger has height $\mathcal{O}(\log_z k)$, so the total number of moves and comparisons are $\mathcal{O}(z \log_z k)$ per element merged. Using binary heaps to merge the z inputs of a node, we get the optimal bound of $\mathcal{O}(\log k)$.

Since we move from level to level and visits each node several times during an invocation of fill on the root of a k-funnel, we should also consider the number of *tree operations*, that is, the number of times we move from one node to another. For that, we see that moving from a node to its parent implies having filled the buffer on the edge connecting them or that the node is the root of a merger has become exhausted. In the first case, moving from a node to its parent, crossing the middle of a k'-funnel, implies having merged $\Omega\left(\beta\left(\sqrt{k'}\right)\right) = \Omega\left(\alpha k'^{d/2}\right)$ elements. We need to move a total of αk^d element past the middle, so we cross it $\mathcal{O}((\alpha k^d)/(\alpha k'^{d/2})) = \mathcal{O}(k^d/k'^{d/2})$ times. The total number of times we go from a node to its parent is thus bounded by the recurrence

$$T\left(k'\right) = \begin{cases} \mathcal{O}\left(\frac{k^d}{z^d}\right), k' \leq z \\ \left(\sqrt{k'}+1\right) T\left(\sqrt{k'}\right) + \mathcal{O}\left(\frac{k^d}{k'^{d/2}}\right), otherwise \end{cases} \tag{4.13}$$

which is dominated by the base case. There are at most $\mathcal{O}(½\log_z k)$ levels with $k' \leq z$ and since we begin and end at the root, we go from parent to child as many times as we go from child to parent and so the total number of tree operations is bounded by $\mathcal{O}((k/z)^d \log_z k)$. If the node is the root of an exhausted merger, we may bring no elements to the parent. However, this can only happen once per node for a total of k times. □

4.1.1 The Sorting Algorithm

Now that we are able to merge multiple sorted streams cache-obliviously and efficiently, we can use that in a mergesort algorithm. The question is now, what order funnel should be used for the merging? We may simply use $k = N$, in which case the sorting algorithm *is* the merging. However doing so we are only feeding the merger one element per stream, and that would violate the requirement that a total of at least αk^d elements is in the input, since d is a constant greater than one. Furthermore, the merger would take up a super-linear amount of space. The next obvious choice would then be $k = (N/\alpha)^{1/d}$ corresponding to the funnel outputting exactly N elements, in which case there are enough elements in the input to satisfy the funnel requirements and the funnel will require sub-linear space as well. The algorithm proceeds as outlined in Algorithm 4-5.

Algorithm 4-5. funnelsort(Array A)

```
if |A| ≤ αz^d then
    sort A "manually"
else
    split A into roughly equal-sized arrays S_i, 0 ≤ i < (|A|/α)^(1/d)
    for 0 ≤ i < (|A|/α)^(1/d)
        funnelsort(S_i)
    construct a (|A|/α)^(1/d)-funnel F
    attach each S_i to F
    warmup(F.root)
    fill(F.root)
fi
```

It is understood that both funnelsort and fill require some way of providing the output. This could be in the form of a pointer to where the first (smallest) element should be placed. The output of fill will be the same as the output of funnelsort, but discussions on exactly where the output of funnelsort will go, is deferred to the next chapter.

The base case threshold is set to αz^d, simply because it needs to be set somewhere. This is a nice place, since it guaranties, that $k > z$, when a k-funnel is used, which may eliminate the need to handle some special cases in the funnel. Exactly how the manual sorting is done, whether with quicksort or bubblesort or something third, is asymptotically insignificant, since it done on a constant-sized array.

Theorem 4-3. Assuming $M^{(d-1)/(d+1)} \geq cB$, for some $c > 0$, Algorithm 4-5 performs $\mathcal{O}(N \log N)$ operations and incurs $\mathcal{O}(dN/B \log_M(N))$ memory transfers to sort an array of size N, for fixed α and z.

Proof. The base case is when all elements fit in cache twice, once for the input and once for the output, along with the funnel needed to merge. In that case, funnelsort incur at most the memory transfers needed to fetch the input into cache. If the input, output and funnel does not fit cache, the second property of the funnel, states that αk^d is $\Omega(kB)$. Further more, with $k = (n/\alpha)^{1/d}$, Theorem 4-1 states that merging n elements incur $\mathcal{O}(\log_M(n)/B)$ memory transfers per element. This gives us the following recurrence for the number of memory transfers incurred by Algorithm 4-5:

$$T(n) = \begin{cases} \mathcal{O}\left(n/B\right), 2n + S\left(\left\lceil n/\alpha^{1/d} \right\rceil\right) \leq M \\ \left\lceil n/\alpha^{1/d} \right\rceil T\left(\left\lceil \alpha^{1/d} n^{\frac{d-1}{d}} \right\rceil\right) + \mathcal{O}\left(n/B \log_M n\right), otherwise \end{cases} \tag{4.14}$$

The recurrence expands to the sum

$$
\begin{aligned}
T(N) &= \mathcal{O}\left(\sum_{i=0}^{\log\log_M N} N^{\left(\frac{d-1}{d}\right)^i} \left(\log_M N^{\left(\frac{d-1}{d}\right)^i}\right) B^{-1}\right) \\
&= \mathcal{O}\left(N/B \log_M N \sum_{i=0}^{\log\log_M N} \left(\frac{d-1}{d}\right)^i\right) \\
&= \mathcal{O}\left(d\, N/B \log_M N\right)
\end{aligned}
\tag{4.15}
$$

For the work bound, we know from Theorem 4-2, that the work done in the funnel is bounded by the number of moves. Since funnelsort, when unrolled, is essentially one big funnel, all elements are merged through $\mathcal{O}(z\log_z N)$ base mergers for a total of $\mathcal{O}(N\log N)$ comparisons and moves, for fixed z. □

Using (3.10) and Theorem 4-2, we conclude that funnelsort is an optimal cache-oblivious sorting algorithm.

As with multiway mergesort, we can see that as long as $\alpha^{1/d}N^{d-1/d} < M$, only the top level of recursion goes outside the cache, incurring at most $4N/B+4$ memory transfers, including those that write back blocks. With M half a gigabyte, $\alpha = 1$, and $d = 2$, this will be the case for all inputs taking up less than 2^{58} bytes, thus we do not expect funnelsort to perform any different from multiway mergesort on this particular level of the memory hierarchy.

4.2 LOWSCOSA

When it comes to the practical application of algorithms minded for massive data sets, it is not only the asymptotic I/O complexity, but also the amount of memory actually needed to perform the task, that is of importance. The space required by the algorithm, can be divided into the space required to store the actual input and workspace. Clearly, a sorting algorithm needs linear space to store the input, and indeed most external memory algorithms require $\mathcal{O}(N)$ space in total. Many, however, also require linear workspace.

Multiway mergesort, unfortunately, is one such an algorithm. While it has good, indeed asymptotically optimal I/O performance, it requires space the size of the array being sorted to store the element in sorted order. These elements are mere duplicates of the elements in the input, so keeping the input around seems wasteful, yet it is necessary for the algorithm to work. Being able to work with data, using very little extra space would be a benefit; compared to otherwise equal sorting algorithms, mergesort might incur up to *twice* the memory transfers of an in-place variant, when $M < N$, simply because we need not incur memory transfers writing the output to previously untouched memory. When $M/2 < N \leq M$, it is even worse; sorting using an algorithm like funnelsort would incur N-$M/2$ memory transfers to write the output, not counting the ones needed to bring the elements into cache, while in-place algorithm would incur no memory transfers at all. In this section, we present a novel extension to k-mergers that enable them to be used in a sorting algorithm that does not require linear working space, in a way that does not affect its overall I/O complexity.

4.2.1 Refilling

The idea for a cache-oblivious sub-linear workspace merge-based sorting algorithm is quite simple; the merger is extended with a refilling feature. The motive is to recycle the space occupied by elements that have already been or are in the process of being merged, while maintain temporal and special locality.

The technique can be applied to heap-based mergers as well as funnels. It consists of requiring the merger to invoke a predefined function when elements have been read in from the input. The function is given an indication of from what section of what input stream they were read. Furthermore, the merger must invoke it before the elements read in is written to the output. The function knows that the elements from that part of the input now is in the hands of the merger, and will eventually be output, so it is free to reuse the space they take up. A key to making this work is that the merger is obligated to invoke this function after having read in no more than a constant number of elements. This way, the I/O complexity of the merger is unaffected; that section of the input is in cache because it has recently been read by the merger. In case of the funnel, one would invoke the refiller after having called fill on a leaf. At this point, at most αz^d elements have been read in from a given stream, so, assuming $2\alpha z^{d+1} < M$, that part of the stream is still in cache, and it can be recycled for free, or at least the price of fetching the elements used. The refilling functionality may in general be useful many scenarios, where space is sparse.

4.2.2 Sorting

The basic steps for the low-order working space cache-oblivious sorting algorithm (LOWSCOSA) is first to provide for a “working area” inside the input array. When that is done, the merger will then sort recursively and then merge using the working area. What we will do is partition the array into two equal sized parts, the first containing elements larger than all elements in the second, then recursively sort the second part. The first part then becomes the output of the merging of the second, but with the merger refilling what it reads in with elements from the first part. The process is illustrated in Figure 4-3. The refiller maintains a pointer to elements in the first part of the array, initially pointing to the first, going forward from there. When invoked, with a section of size n, it copies n elements beginning from the pointer to that section.

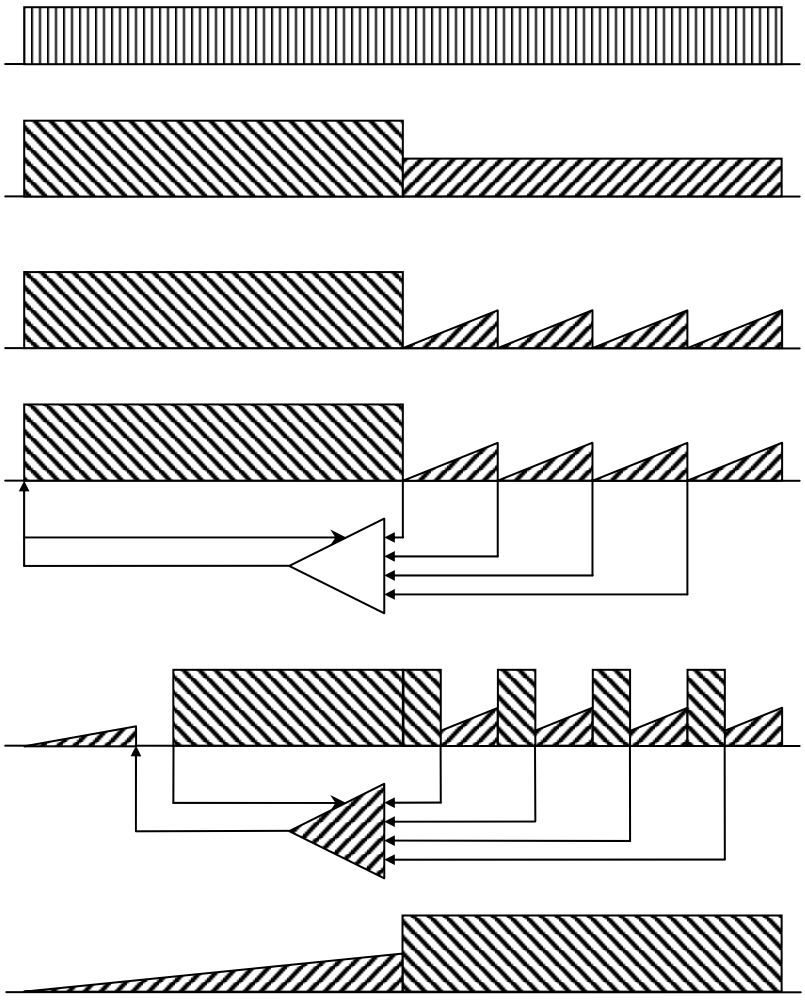

Figure 4-3. The process of multiway merging with low-order working space. Horizontal sections indicate unordered elements. First step partitions the array into large (diagonal-down patterned) and small (diagonal-up patterned) elements. Arrowheads indicate flow direction of elements; the arrow going into the side of the merger is the refiller. The gap between the output and the refiller shows that elements are read in through the refiller before elements are output from the merger.

Since the refiller is invoked before elements are output, the refiller pointer will always stays ahead of where the output is written, and within the first part of the array. The algorithm has now sorted the first half of the array. It then recurses on the second half, as is illustrated in Algorithm 4-6, where a funnel is used as merger. The merger and the parameters can be changed to that of a cache-aware mergesort.

Algorithm 4-6. LOWSCOSA(Array A)

```
if 2|A| ≤ αz^d then
    sort A "manually"
else
    partition A into a half of large elemens A_l and a half of small A_s
    split A_l into roughly equal-sized arrays S_i, 0 ≤ i < (N/(2α))^(1/d)
    for 0 ≤ i < (|A|/(2α))^(1/d)
        funnelsort(S_i)
    construct a (|A|/(2α))^(1/d)-funnel F
    attach each S_i to F
    set pointer in F.refiller to A_l
    warmup(F.root) // A_s now contain large elements
    LOWSCOSA(A_s)
fi
```

We will discuss the practicality of the partitioning in Section 5.5.2. For now, in the analysis, we use of the following lemma:

Lemma 4-2. Finding the median of an array incurs $\mathcal{O}(1+N/B)$ memory transfers, provided $M \geq 3B$.

Proof. The proof can be found in [Dem02]. Since the algorithm relies on constant time operations and scanning, the result is rather intuitive. □

With the median at hand, we can make a perfect partitioning efficiently, thus avoiding any complications and give a worst-case bound.

Theorem 4-4. Assuming $M^{(d-1)/(d+1)} \geq cB$, for some $c > 0$, and $M > 2\alpha z^{d+1}$, Algorithm 4-6 performs $\mathcal{O}(N\log N)$ operations and incurs $\mathcal{O}(dN/B\log_M(N))$ memory transfers to sort an array of size N.

Proof. Elements now pass through the merger in two ways; either through the refiller or through input streams being merged. After the partitioning the large elements pass through the refiller, incurring $\mathcal{O}(B^{-1})$ memory transfers each. The small elements gets sorted recursively incurring $\mathcal{O}(d\log_M((N/\alpha)^{1/d}/2)/B) = \mathcal{O}(d\log_M(N)/B)$ memory transfers each, pass through the merger, incurring $\mathcal{O}(\log_M(N)/B)$, but is not present at the next level of recursion. The base case is when all elements and the merger used to merge the recursively sorted streams fits in cache. From then on, no memory transfers are incurred. Using Theorem 4-1, Theorem 4-3, and Lemma 4-2, we get the recurrence

$$T(n) = \begin{cases} \mathcal{O}(1), n + S\left(\left\lceil n/\alpha^{1/d} \right\rceil\right) \leq M \\ \mathcal{O}\left(n/B\right) + \mathcal{O}\left(d\,\frac{n}{B}\log_M n\right) + T\left(\left\lceil n/2 \right\rceil\right), otherwise \end{cases} \tag{4.16}$$

which expands to

$$\begin{aligned} T(N) &= \mathcal{O}\left(\sum_{i=0}^{\log N} 2^{-i} N \left(1 + d \log_M \left(2^{-i} N\right)\right) B^{-1}\right) \\ &= \mathcal{O}\left(B^{-1} d \log_M N \sum_{i=0}^{\log N} 2^{-i} N\right) \\ &= \mathcal{O}\left(d \, {}^{N}\!/_{B} \log_M N\right) \end{aligned} \tag{4.17}$$

The work bound is given by a similar recurrence:

$$\begin{aligned} T(n) &= \begin{cases} \mathcal{O}(1), 2n \leq \alpha z^d \\ \mathcal{O}(n) + \mathcal{O}\left({}^{n}\!/_{2} \log n\right) + T\left(\left\lceil {}^{n}\!/_{2} \right\rceil\right), otherwise \end{cases} \\ &= \mathcal{O}\left(\sum_{i=0}^{\log N} 2^{-i} n \left(1 + \log\left(2^{-i} n\right)\right)\right) \\ &= \mathcal{O}(n \log n) \end{aligned} \tag{4.18}$$

□

So the LOWSCOSA is also optimal in the cache-oblivious model, but what is unique to it is that the following theorem holds:

Theorem 4-5. Algorithm 4-6 requires no more than $\mathcal{O}(N^{(d+1)/(2d)}+N^{(d-1)/d})$ working space.

Proof. Working space is required to store the funnel and the output of the sort of . Since the algorithm is tail-recursive, it requires no stack to store the state of the recursive calls. At each level, we use funnelsort to sort subarrays of size $\mathcal{O}(N^{(d-1)/d})$. We allocate space to store the output of one call to funnelsort; when the sort is completed, the elements can be copied back and the allocated space reused in the next call to funnelsort. When we are done sorting using funnelsort, the space is freed and space allocated for the funnel. From the proof of Lemma 4-1, we have that, a k-funnel takes up $\mathcal{O}(k^{(d+1)/2})$ space. With $k = \mathcal{O}(N^{1/d})$ that becomes $\mathcal{O}(N^{(d+1)/(2d)})$. For $1 < d \leq 3$, the space required for the funnel dominates while for $d > 3$, space requirements are bounded by output of the funnelsort. Both terms are bounded by $\mathcal{O}(N^{(d+1)/(2d)}+N^{(d-1)/d})$, which is $o(N)$ for all $d > 1$. Before continuing recursively on the second half, all workspace is freed. This way the total working space consumption is maximal at the top-level of the recursion. □