Chapter 3

Theory of IO Efficiency

Having now realized that instruction count is not always a good indication of the running time of an algorithm, we will be interested in obtaining a theoretical understanding of what might then influence it.

In this chapter, formal models capturing the effects of the cache-misses and page faults are introduced. Lower bounds on the complexity of sorting are proven and a cache-aware algorithm meeting that bound is given.

3.1 Models

Theoretical algorithm analysis has been founded on one fundamental activity: counting the number of instructions executed. Knuth pioneered this discipline in the work The Art of Computer Programming [Knu98]. The approach was to develop a hypothetical, yet then representative instruction set, dubbed MIX, then implement every algorithm in the book using this set and thoroughly analyze the number of instructions executed in each of them. The result was very precise statements on worst- and average-case instruction count of each algorithm.

Today, this attention to detail is not often seen; a high-level description of an algorithm is preferred, to ease understanding and to limit cluttering with instruction set specific details. In addition, when implementing algorithms we do so in high-level programming languages for portability and genericity. Exactly which instructions are hidden behind the high-level language constructs is not important for the analysis; a focus on only a subset of the instructions has been prominent. For sorting, the subset has traditionally been a comparison instruction, while numerical computations have been a particular floating-point operation, say multiplication. These instructions can easily be isolated, even if the algorithm is only described in a high-level language. To ease the burden of analyzing algorithms further, we use asymptotic notation, which allows us to discount low-order terms and constants in high-order terms.

3.1.1 The RAM Revisited

The strength of a model lies in part in its ability to support lower bounds on the complexity of interesting problems; without lower bounds, we cannot prove optimality of algorithms solving the problems.

Consider sorting in the RAM model. An easier problem than sorting is determining the permutation that will bring the elements in non-descending order. The number of permutations of n elements is $n!$. When doing a one comparison, the order of two elements is determined. This order can be chosen so at most half the remaining permutations are excluded. It thus takes at least $\log(n!)$ comparisons to exclude all but the right permutation. Recall, that in the RAM model, we are not allowed to examine the individual bits of the elements, so the only way of excluding permutations is by comparing elements. The argument is then that a correct sorting algorithm must be able to exclude all but the one permutation that will bring the elements in this order. Say it had excluded all but two permutations π_1 and π_2, and it simply picked one, say π_1, rather than do the last comparison the find the correct one. Then there would be an input to the algorithm that it would not be able put in correct order, namely the permutation π_2^{-1} of a sorted sequence. The argument that there exist inputs, on which the algorithm will be incorrect, is known as an *adversary argument*; an adversary decides the input, or rather answers queries about it in a consistent manner, and if it is possible to do so in a way that reveals flaws in the algorithm, it cannot yet have solved the problem. A similar, simpler argument shows that there exists an input to a correct sorting algorithm, on which the algorithm must make $n+1$ moves, assuming output must reside in the same locations as the input.

In the RAM model, all operations take unit time so a sorting algorithm must take at least $\Omega(\log(n!))$ units of time, even if it also only does n moves. Consider such an optimal sorting algorithm. On any given hardware implementation (and a given type of elements), there exist a constant $c > 0$, such that it takes c times longer to move elements, than to compare them. Discounting other operations, the running time of such an algorithm will be proportional to $\log(n!)+cn$. Using asymptotic notation, we would ignore the latter term, but what happens, when constants matter?

A floating-point division may for example take 25 times that of a multiplication, which takes five times that of integer addition. If an algorithm makes n divisions and $n\log n$ additions, we say that the complexity is $\mathcal{O}(n\log n)$, but when $\log n < 125$, that is, for all $n < 10^{37}$, the execution time will be dominated by the time it takes to perform the divisions, which is $\mathcal{O}(n)$. In the previous chapter, we saw that not even identical instructions execute in the same amount of time; an instruction executed in one context may execute in twenty-thirty, maybe even a million times the time in which the same instruction executed in a different context. We saw certain memory operations were a very large factor slower than any other operation.

Overzealous use of asymptotic notation hides important aspects of algorithm performance; however, the solution is not to abandon asymptotic notation, but rather the model. In the RAM model, memory operations are just another type of operation. This observation invites the idea, that in estimating running time of an algorithm, we should not count the simple operations, rather these types of memory operations.

3.1.2 The External Memory Model

In 1972, Floyd pioneered the notion of analyzing the number of transfers between primary and secondary storage, proving upper and lower bound on transfers incurred during matrix transposition [Flo72]. Nine years later, Hong and Kuhn derived a lower bound for Fast Fourier Transformation [HK81]. The next major step was taken in [AV88], when these results were proven in a more general setting: The *external memory model*. Specifically, what this new model was able to account for was that elements were transferred in blocks with a non-constant capacity that is independent of memory size. A lot of work has since been done, both practical and theoretical, in developing efficient algorithms in this model (see [Vit01] for survey). A multitude of other models, most more complex than the external memory model, such as the multi-level memory model, the hierarchical memory model, and the uniform memory model, that attempts to model the memory hierarchy has also since been proposed (see [FLPR99, Sect. 7] for an overview and references).

The external memory model is a model of secondary storage, rather than of computation. In the external memory model, secondary storage consists of a random-access magnetic disk. All computation is done in *and only* in primary storage (memory), and data is transferred from secondary storage (and back) in *blocks*. The model is characterized by these parameters:

- Problem size N: the number of elements to be sorted.
- Memory size M: number of elements that can fit in memory.
- Block size B: number of elements that can be transferred in a single block.
- Number of disks P: number of blocks that can be transferred concurrently.

The following relation is said to hold: $1 \leq B \leq M$. For the discussion of the external memory model in this thesis, we will concentrate on the case $P = 1$. Note that these parameters are given by the concrete implementation of the model, i.e. the machine on which the algorithm is run. Figure 3-1 depicts such a machine.

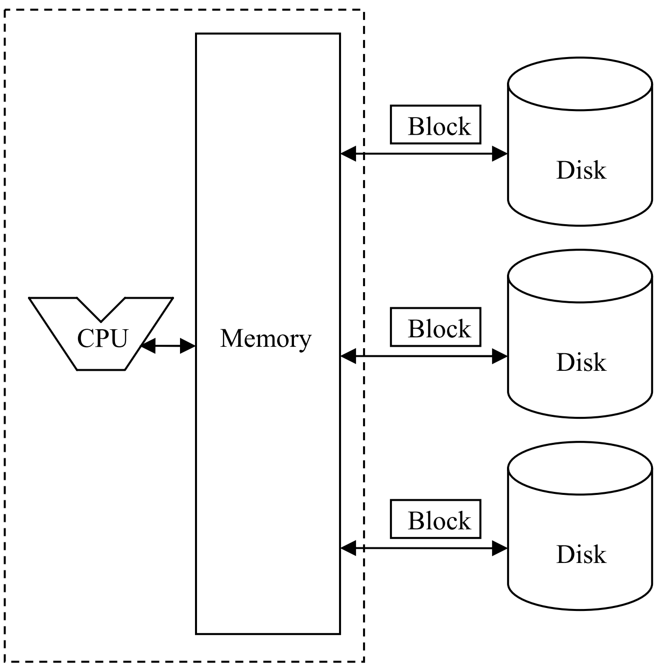

Figure 3-1. Illustration of the external memory model with $P = 3$ parallel disks.

The input of an algorithm in the external memory model is initially placed on disk, from where it can be read into memory in blocks (see Figure 3-1). There can be no more than M/B blocks in memory at any one time, so blocks may also have to be written back to disk, to not loose data; if a block is read into and the memory is already filled, some portion of the memory will be overwritten. If it matters, for correctness or complexity, it should be stated exactly where the block should be placed in memory; the model states that memory is fully associative, so we are free to choose where is should go. When blocks have been written to disk, they do not occupy space in memory. The complexity of the algorithm is measured in the number of times it transferred a block from or to disk. A transfer is also known as an I/O, which is shorthand for input or output.

Since the model is essentially that of a two-level storage system, where data is transferred in blocks, it may also be used to describe other two levels of the memory hierarchy, such as L2 cache and DRAM memory or TLB and page tables, the value of constants M and B are merely different. Note, in practice we have neither control of when the transfer of blocks should take place, nor where to put the blocks in cache. Indeed most L2 caches are not fully associative.

A Simple Example: Scanning

Consider the simple problem of computing an aggregate of N elements, for example the largest element or the sum of the elements. Algorithm 3-1 solves the problem in the external memory model. It assumes the elements are stored contiguously on disk, so that the i'th block contains elements $(i\text{-}1)N/B$ through $iN/B\text{-}1$. For simplicity, we assume $N = cB$, for some positive integer c.

Algorithm 3-1. EM_sum

```
sum = 0;
for( int i=0; i!=N/B; ++i )
{
    read the i'th block into memory;
    compute the sum s of elements in the block;
    sum += s;
}
```

We use the high-level description compute the sum s of elements in the block, because exactly how it is done does not matter in the analysis, and so we leave out the details. It is easy to see the algorithm is correct (provided s is computed correctly). The analysis is equally simple: the algorithm performs N/B reads. This is optimal because computation can only be done in memory and as with all aggregate problems, if an algorithm ignores one or more elements of the input then the algorithm will be incorrect on inputs, in which the aggregate depends on the ignored element, so all elements have to, at some point, have been in memory. We transfer elements in blocks of B elements, so at most B elements can get into memory per read. Since at least N elements have to be read in, a correct aggregate computing algorithm must perform at least N/B reads.

The technique used in Algorithm 3-1 is simple, yet it illustrates some important points in dealing with I/Os. The placement of the input on disk is essential; were the elements scattered randomly around the disk, we would not be able to get B elements into memory in a single read. The region of memory where the block read in is stored, is called a *buffer*. When laid out contiguously, we were able to touch elements at an amortized cost of B^{-1} reads per element. We will call a collection of elements with this property a *stream*. Streams are often either input or output streams, where output streams can store a collection of elements on disk at an amortized cost of B^{-1} writes per element. This is done by collecting elements in the buffer, writing it to disk only when it becomes full. Note that if we read in the next block to the place in memory the previous block was stored, the algorithm would still be correct. Thus, a stream requires no more memory than one block occupies.

The third point is specific to the model; the values of M and B (and P) are specific to a concrete instance of the model. When implementing e.g. Algorithm 3-1, we would either have to decide on a value for B, in which case it would be suboptimal when run a machine with block size $B' \neq B$, or we would have to figure out what B to use at runtime, information that is not in general available. Not having access to the value of M and B in an implementation implies, that either the implementation is not optimal, or they will become suboptimal, when the values change e.g., when the memory of a computer is upgraded.

Binary search

Another simple and illustrative problem is that of finding a particular element among sorted set of elements. In the RAM model, this can be done efficiently using a balanced binary search tree in which elements are stored in the nodes. Elements stored in the left subtree of a node are all smaller than the element in the node and all elements in the right subtree are greater. This property is used to navigate down through the tree to isolate the desired element, in time proportional to the height of the tree, which is $\mathcal{O}(\log(n))$. That this is optimal can be realized by noting that each comparison can be chosen by the adversary to reduce the set of candidate keys by at most a factor of two. After $\Omega(\log(n))$ comparisons, we have thus eliminated all but the right key.

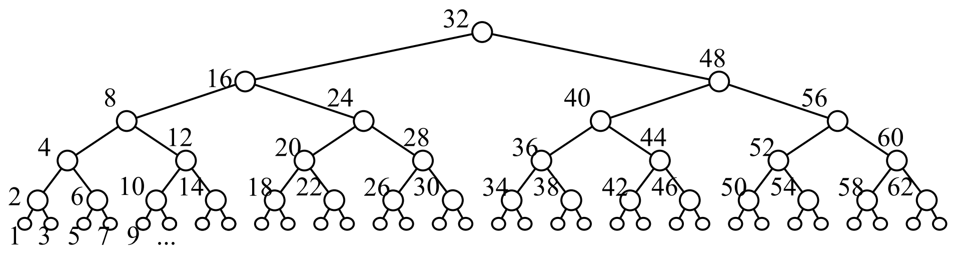

Figure 3-2. A binary search tree. The number by the nodes indicates the rank of the element stored there.

In the external memory model, reading a block for each new level of a binary tree will be suboptimal. Instead, we use B-trees, where each node contains a block of elements and has $B+1$ subtrees. The elements in the block functions as partitioning elements for the elements in the subtrees; all elements in the i left most subtrees of a node are smaller than the first i elements of the block in the node. When doing a search the block associated with a node is read in, using one I/O. Using the elements in the block, the node in which to continue the search is determined. This way we still do one I/O per level in the tree, but the height of the tree is only $\mathcal{O}(\log_B N)$, and thus so is the complexity. This too is optimal by the same argument as above, except an I/O now read in B elements and the adversary can only choose an outcome that reduces the set of candidate keys by a factor of at most B.

3.1.3 The Cache-Oblivious Model

As discussed in Chapter 2, well-established ways of managing the reads and writes from and to disk, so that the programmer need not know the details of disk input/output, already exist namely using virtual memory. Programmers have become used to the same abstraction on all levels of the memory hierarchy. The *cache-oblivious model*, introduced by Frigo, Leierson, Prokop, and Ramachandran in 1999 captures this way of abstracting memory transfers away from algorithms and their implementations [FLPR99].

The idea of the cache-oblivious model is strikingly simple; design the algorithm for the RAM model, but analyze it in the external memory model. Perhaps the most profound consequence of this is, since the algorithm is oblivious to the structure of the memory hierarch, an optimal cache-oblivious algorithm is *automatically optimal on all levels of the memory hierarchy, and on all hardware implementations of these hierarchies*. We now no longer refer specifically to memory and disk, so in this thesis, when discussing cache-oblivious algorithms, we will use the term *memory* for general storage, *cache* for the faster level of storage that, caches elements stored in memory, and a *memory transfer* to refer to the process of moving a *block* from one level to another. The complexity of an algorithm in the cache-oblivious model is both the work done in the RAM model and the number of memory transfers.

Stating that an algorithm, described for the RAM model, is optimal on any level in a memory hierarchy when analyzed in the external memory model obviously requires some assumptions. Aside from assuming the levels work like the external memory model, in that the first level is fully associative and inclusive, and elements are transferred in blocks, we need an assumption on the page replacement strategy. These assumptions transform the external memory model to the *ideal cache model*, in which algorithms are then analyzed. The assumptions made in the ideal cache model are:

- *Optimal replacement*: When blocks are transferred to memory, the underlying replacement policy is the optimal offline algorithm.
- *Exactly two levels of memory*: There are no more than two levels of memory, that is, more than one level of cache. This is not a restriction compared to the external memory model, but it is compared to other, more sophisticated models like multi-level models.
- *Automatic replacement*: Blocks are transferred automatically, not explicitly by the algorithm. This is in contrast to models like the external memory model, where memory management is done explicitly.
- *Full associativity*: Blocks can be placed anywhere in memory.
- *The cache is tall*: See details below.

With the introduction of the ideal cache model, detailed proofs were given that algorithms that are optimal in the ideal cache model are also optimal in other models, such as the external memory model, where e.g. automatic replacement is not present, and multi-level models [FLPR99]. The computers, that are the targets of the engineering effort of this thesis (modern computers as discussed in Chapter 2), naturally fulfill most of these assumptions. Specifically, will we use the convenient virtual memory abstraction, so that automatic replacement and full associativity is a given. As for limited associativity in lower level caches, the construction used in the justification argument given in [FLPR99] is not of practical use due to a large overhead, so we would have to find another way of dealing with limited associativity anyway. We will analyze the algorithms in a two-level memory model, and prove they are optimal, between these two levels, regardless of the architectural parameters. This way we prove they are optimal between any two levels but not that they are optimal on all levels simultaneously. That they indeed are is also proven in [FLPR99], but using an assumption that all levels of cache are inclusive, which is not always the case, specifically not at the levels of virtual memory.

Optimal Replacement Assumption
One assumption no computer will ever fulfill, however, is that of optimal replacement. Since cache-oblivious algorithms are unaware of the underlying caching mechanisms, they are unable to control them. Elements are transferred between levels in blocks by an underlying mechanism; if a particular element is not in cache, it must be brought in. Now, the mechanism must decide where to put it. In the ideal cache model, the choice is simply to pick the optimal replacement. This eases the analysis, in that when arguing upper bound on the complexity, we may simply say that the page replacement mechanism chose whatever we needed it to choose; if it did not, it must have chosen something even better, since it is optimal, thus improving the complexity of the algorithm. However, is it too unrealistic? Most real life caching mechanisms use LRU, or some more or less crude approximation.

The argument for optimal replacement being a reasonable assumption is based on a result by Sleator and Tarjan. It states that the number of cache misses Q_{LRU} on a cache using LRU is $M_{\mathrm{LRU}}/(M_{\mathrm{LRU}}\text{-}M_{\mathrm{OPT}}+1)$-competitive with an optimal replacement strategy [ST85], that is

$$Q_{\mathrm{LRU}} \le \frac{M_{\mathrm{LRU}}}{M_{\mathrm{LRU}} - M_{\mathrm{OPT}} + 1} Q_{\mathrm{OPT}} \tag{3.1}$$

with Q_{OPT} being the number of cache misses incurred by a sequence of memory requests with an optimal replacement strategy, and M_{LRU} and M_{OPT} being the size of the caches for the LRU and optimal strategies respectively. An algorithm incurring $Q(N,M/\gamma,B)$ memory transfers on an optimal replacement cache of size M/γ, for some constant $\gamma > 1$, will thus incur no more than $\gamma M/(\gamma M\text{-}M+\gamma)\,Q(N,M/\gamma,B) \le \gamma/(\gamma\text{-}1)\,Q(N,M/\gamma,B)$ transfers on an LRU cache of size M. So if $\gamma Q(N,M/\gamma,B) = \mathcal{O}(Q(N,M,B))$, which is known as the *regularity condition*, the number of transfers on an LRU cache, $\gamma/(\gamma\text{-}1)\,Q(N,M/\gamma,B)$, is $\mathcal{O}(Q(N,M,B))$, which was the number of transfers incurred according to the analysis in the ideal cache model.

Tall Cache Assumption
When solving non-trivial problems, it is common to exploit a certain level of granularity in the cache; put simply we would like the cache divided into more blocks than there can be elements in one block. To achieve I/O optimality cache-obliviously, we thus assume that there exist a (positive) constant c, such that $M/B \ge cB^{2/(d\text{-}1)}$, with the value of d being specific to the implementation of the algorithm, though strictly larger than 1. In case of the problem of sorting, this assumption has been proved both necessary ([BF03]) and sufficient (Chapter 4).

In theory, the ideal cache model is justifiable *in asymptopia*, but it is still a major open question, whether the performance of cache-oblivious algorithms is influenced by the less than ideal memory systems of real life. In particular, when faced with a direct-mapped cache we cannot, in practice, build up the simulation based on 2-universal hashing suggested by Prokop *et al.*, due to the very high overhead. Likewise, the tall cache assumption is in some sense always true, since we are free to choose c, however being forced to choose c small to fulfill the tall cache assumption, will impact the performance in a way, that is hidden in the asymptotic analysis.

None of the assumptions made by the ideal cache model avoids the basic need for blocks being transferred from level to level; computation can only be done in cache, and the only way to bring data there is by block transfer. This in turn means that the lower bounds that hold in the external model also hold in for the I/O complexity of cache-oblivious algorithms. Likewise, the lower bounds of the RAM model also hold for the work done by a cache-oblivious algorithm.

Scanning Revisited

Let us return to the simple aggregate computation example from the external memory model and see how it looks in the cache-oblivious model. In the cache-oblivious model, we do not control the disk, and so may seemingly not be able to control the layout of elements in memory, in a way that was so crucial to the performance of Algorithm 3-1.

Instead, we use the *array* construct. Arrays are collections of elements that are placed contiguously in the address space. On all levels of cache, a block of elements contains B elements that are contiguous in the address space, so accessing B elements that are contiguous in the address space, will cause no more than two blocks to be transferred, indeed accessing N elements contiguous in the address space cause no more than $N/B+2$ memory transfers. This is exactly what splitting up memory in blocks is designed to be efficient at. While arrays may not guarantee that elements are placed contiguously in secondary storage, they do provide us with what we need. In practice, however, the blocks that constitute an array may be scattered around the disk. Hence, finding the block containing the element next to the last element of a previous block may not be as simple as taking the next block on disk, so the time it takes to serve the individual memory transfers may be higher than when controlling the layout directly. It is still considered a constant, though. Algorithm 3-2 computes the sum of N elements.

Algorithm 3-2. CO_sum(Array A)

```
sum = 0;
for( int i=0; i!=N; ++i )
    sum += A[i];
```

It really could not be simpler. Notice that the assumption that $N = cB$, for some positive integer c made in the analysis of Algorithm 3-1, is not needed here. The discussion of the array construct above, gives us that Algorithm 3-2 incurs no more than $N/B+2$ memory transfers, which is asymptotically optimal. Further, the work done is $\mathcal{O}(N)$ which is also optimal, so Algorithm 3-2 is optimal in the cache-oblivious model.

The stream concept is the same as in the external memory mode, except in the cache-oblivious model, its description is simpler; we can use arrays to realize streams, either an entire array or just a part of an array, called a *subarray*. The term buffer now also simply applies to an array or a subarray, used to store elements; elements can be inserted into or extracted from buffers in a streaming fashion, incurring B^{-1} memory transfers per such operation amortized

Binary Search and the van Emde Boas Layout

For binary search in the cache-oblivious model, we use search trees, we also need to consider the layout. The standard way to do this in the RAM model is to lay out the nodes in-order, that is, simply keep the elements in an array in sorted order. This way, the rank of an element is also the position in the array, so the number by the nodes in Figure 3-2 indicates the position of the node in the array. Unlike with scanning-type problems, however, it turns out to be insufficient to simply adapt the RAM model algorithm and use an array to guarantee locality; the only place we gain from this is at the bottom of the tree. More precisely, the subtrees of height $\Omega(\log B)$ at the bottom of the tree will be stored contiguously and elements within them can all be accessed after incurring one memory transfer, however, no level above that exhibit locality. Thus, a root to leaf traversal incurs $\mathcal{O}(\log N\text{-}\log B)$ memory transfers, which is suboptimal.

It is old wisdom in computer science that dealing with datasets in a recursive fashion yields good locality. The intuition behind this is that when the problem size becomes small enough to fit in cache or in a single block, we can likely solve them optimally and combining solutions to subproblems are often trivial. What we need for binary search is a way to recurse on the paths from the root to the leaves. van Emde Boas first presented a way to recurse on trees vertically, leading to an $\mathcal{O}(\log\log U)$ query time priority queue [EKZ77]. The idea was to build a heap of elements from a universe of size U recursively from heaps of size $\sqrt{U}$, denoted bottom[$0,1,\ldots,\sqrt{U}-1$], and use a single heap of size $\sqrt{U}$, the top, to represent the presence of elements in a given bottom-heap. A query would then first go to the top and then to one of the bottom heaps. The binary search tree equivalent of this structure is a tree that is conceptually cut in half at the middle level of edges. This is so far only conceptual; the search procedure is still the same – querying the *top tree* amounts to the first half of the root to leaf traversal and locates the *bottom tree* in which to proceed. Querying that bottom tree is the rest.

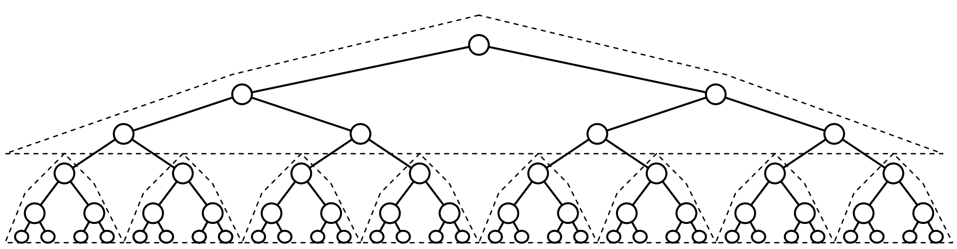

Figure 3-3. A binary search tree of size 63 split into a top-tree of size 7 and eight bottom trees also of size 7.

To apply this to get better locality, we use the concept of top and bottom trees in the storage of the tree. We lay out the nodes according to the van Emde Boas recursion: the $\sqrt{N}$ top elements are stored recursively at the head of an array and after them, in no particular order, the elements in the $\sqrt{N}$ bottom trees are stored recursively. This particular way of laying out a tree is now called the *van Emde Boas layout*. The effect of this is that the nodes on the first half of the path from the root to the leaves are stored contiguously, as are the nodes on the second half. To realize that using the van Emde Boas layout suffices to get optimal search cost, we conceptually follow the recursion until subtrees are of size at most B and a least $\sqrt{B}$. Such a tree has height at least ½log$B$. Nodes of subtrees of the recursion are stored contiguously and so these elements can occupy at most two blocks. On the path from the root to the leaves, we thus visit no more than $2\log N/\log B$ such trees and each visit costs no more than two memory transfers for a total of $4\log_B N$ memory transfers.

3.2 External Memory Sorting

Let us now turn to the problem of sorting. In this section, we study sorting in the external memory model. Chapter 4 is dedicated to optimal sorting in the cache-oblivious model.

3.2.1 Lower Bound

An adversary argument similar to the one in the previous section proves a lower bound on the number of I/Os incurred by a correct sorting algorithm. This proof was done in [AV88], with the introduction of the external memory model, in the general case that included parallel disks. We will here show the bound in the case of a single disk ($P = 1$). For the proof, we shall need the following lemmas:

Lemma 3-1. $n\log n - \frac{n-1}{\ln 2} \le \log(n!) \le n\log n$.

Proof. We have

$$\log(n!) = \log\prod_{i=1}^{n} i = \sum_{i=1}^{n}\log i\,. \tag{3.2}$$

Since log is a continuous increasing function, the following bounds hold

$$\begin{gathered}\int_2^{n+1}\log(i-1)di \le \sum_{i=1}^{n}\log i \le \int_1^{n+1}\log i di \Leftrightarrow \\ n\log n - \frac{n-1}{\ln 2} \le \sum_{i=1}^{n}\log i \le (n+1)\log(n+1) - \frac{n}{\ln 2} \le n\log n\end{gathered} \tag{3.3}$$

□

Lemma 3-2. Assuming $m \ge 2b$, $\log\begin{pmatrix} m \\ b\end{pmatrix} \le 3b\log {}^{m}\!/\!_{b}$.

Proof. We have

$$\log\binom{m}{b} \le \log\left(\frac{m^b}{b!}\right) = b\log m - \sum_{i=1}^{b}\log i\,. \tag{3.4}$$

Using (3.3), we get

$$\begin{aligned}\log\binom{m}{b} &\le b\log m - b\log b + \frac{b-1}{\ln 2}\\ &\le b\left(\log m - \log b + \frac{1}{\ln 2}\right)\\ &= b\left(\log m/b + \log e\right)\\ &\le 3b\log m/b\end{aligned} \tag{3.5}$$

with the last inequality using $m \ge 2b \ge b\log e$. □

Theorem 3-1. Assuming $M \ge 2B$, a correct sorting algorithm must incur $\Omega(N/B\log_{M/B}(N/B))$ I/Os in the external memory model.

Proof. As argued in the previous section, a correct sorting algorithm must be able to exclude all but the one permutation that will bring elements in order. The only operation in the RAM model that decreased the number of possible permutation was a comparison. In the external memory model, the only operations allowed are reads and writes; however, having done a read, an external memory algorithm may exclude many more than half of the permutations. When a block is read in, the position of the B elements in the block among all M elements already in memory may be determined, reducing the number of permutations by a factor $\binom{M}{B}$. Furthermore, when a block first gets read in (or when it is read in later, but no more than once), the algorithm may determine the position of all elements in the blocks among themselves, eliminating a further factor of $B!$ permutations. We call the latter reading an untouched block, while the other reads read touched blocks. The process of writing a block back to disk does not reduce the number of possible permutations.

Let $\varphi(t\text{-}1)$ denote the number of remaining possible permutations after t-1 reads or writes. Depending on the type of operation the t'th is, $\varphi(t) \ge \varphi(t\text{-}1)/X$ possible permutations remain, with

- $X = \binom{M}{B}$, in case of a read of a touched block,
- $X = \binom{M}{B}B!$, in case of a read of a untouched block, and
- $X = 1$, in case of a write.

Since we can read an untouched block no more than N/B times, we get

$$\varphi(t) \geq \frac{\varphi(0)}{\binom{M}{B}^t (B!)^{N/B}} \tag{3.6}$$

With $\varphi(0) = N!$, we are interested in the smallest t, such that $\varphi(t) \leq 1$:

$$\begin{aligned} 1 &\geq \frac{N!}{\binom{M}{B}^t (B!)^{N/B}} \Leftrightarrow \\ t \log \binom{M}{B} &+ N/B \log(B!) \geq \log(N!) \end{aligned} \tag{3.7}$$

Using Lemma 3-1 and Lemma 3-2, we get

$$\begin{aligned} & t \log \binom{M}{B} + N/B \log(B!) \geq \log(N!) \Rightarrow \\ & t\left(3B \log M/B\right) + N/B\left((B+1)\log B - \frac{B}{\ln 2}\right) \geq N \log N - \frac{N-1}{\ln 2} \Rightarrow \\ & t\left(3B \log M/B\right) \geq N \log N - N\frac{B+1}{B}\log B + N \log e \Rightarrow \\ & t \geq \frac{N \log N/B + \log e}{3B \log M/B} \end{aligned} \tag{3.8}$$

as desired. □

3.2.2 Multiway Merge Sort

A variant of merge sort achieves optimal complexity in the external memory model. In this thesis, we refer to it as multiway mergesort. It relies on an abstract data structure we call a *k-merger*. A k-merger is capable of merging up to k sorted streams; input streams are attached to the merger, and when invoked, the merger outputs the elements of the input streams in one sorted stream. An efficient implementation would use an efficient priority queue, such as a binary heap, of size k, storing pairs (s,e) of stream identifiers s and elements e with e being a copy of the next element of the stream identified by s. The priority of the pair is e. Additionally, from each stream a block of elements is kept in memory. An element is merged by doing a `delete_min` on the queue, returning (s,e), extract the next element e' from s, insert (s,e') into the queue and output e. According to the discussion of Algorithm 3-1, all accesses are in a streaming fashion, provided memory has the capacity to hold k blocks and the priority queue.

Multiway mergesort works in two phases. First, the "run formation" phase reads in the input, one memory load at a time, sorts the memory loads internally, and outputs the sorted run. Second, M/B sorted runs are merged using an M/B-merger, reducing the number of sorted runs by a factor of M/B. More accurately, an $(M/(B{+}1)\text{-}1)$-merger should be used to make room for the queue and leave one block for streaming the output. This is repeated until one sorted run remains. The procedure is illustrated in Algorithm 3-3. For simplicity, we assume $N = c_1 M = c_1 c_2 B$ for some positive integers c_1 and c_2.

Algorithm 3-3. multiway_mergesort

```
runs = N/M
repeat runs times do
    read in M/B blocks
    sort the M elements
    write out M/B blocks
od
while runs > 1 do
    repeat for groups of M/B runs do
        construct an M/B-merger
        attach M/B runs as its input streams
        merge all elements in the runs
    od
    runs = runs/(M/B)
od
```

That Algorithm 3-3 is indeed optimal, is given by

Theorem 3-2. Algorithm 3-3 incurs $\mathcal{O}(N/B\log_{M/B}(N/M))$ I/Os in the external memory model.

Proof. The first phase incurs N/B reads and N/B writes in total. So does each iteration in the second phase. After the first phase, there are N/M runs. An iteration in second phase reduces the number of runs by a factor of M/B, thus there are $\log_{M/B}(N/M)$ iterations in the second phase for a total of

$$\begin{aligned}\frac{2N}{B}+\frac{2N}{B}\log_{M/B} {}^{N}\!/\!_{M} &= \frac{2N}{B}\left(1+\log_{M/B} {}^{N}\!/\!_{M}\right)\\ &= \frac{2N}{B}\left(\log_{M/B} {}^{M}\!/\!_{B}+\log_{M/B} {}^{N}\!/\!_{M}\right)\\ &= \frac{2N}{B}\log_{M/B} {}^{N}\!/\!_{B}\end{aligned} \tag{3.9}$$

I/Os, which is correct in asymptopia, even if $N = c_1 M = c_1 c_2 B$ does not hold. □

Note that, only when $N \gg M$ does the logarithmic term become significant. It might be interesting to see what the complexity is for more sensible N. When $N \leq M$, the second phase is never invoked, so a total of than $2\lceil N/B\rceil$ I/Os are incurred. For all $N \leq M(M/(B{+}1)\text{-}1)$, only one iteration is needed in the second phase, and we incur $4\lceil N/B\rceil$ I/Os. With M elements taking up one half gigabyte and $B{+}1$ elements 4kB, the second condition is met for all input that takes up less than 64 TB. Assuming one millisecond per I/O, sorting a data set of that size would take at least two years.

3.2.3 Distribution Sort

Distribution sorting is a recursive process, like mergesort, but it solves a different problem at each level [Knu98]. A set of S-1 partitioning elements are used to partition the input in S disjoint *buckets*. All the elements in one bucket are smaller than elements in the next bucket.

The adaptation of distribution sort to the external memory model follow that of mergesort; problems larger than M are split into problems a factor of M/B smaller. Distribution generates subproblems a factor of S smaller, so we choose $S = \Theta(M/B)$. When problems become smaller than M, we incur no more I/Os, so we get $\mathcal{O}(\log_{M/B}N/B)$ levels of recursion. If we can distribute N elements evenly into M/B buckets using $\mathcal{O}(N/B)$ I/Os, distribution sort becomes optimal in the external memory model. [AV88] shows how to distribute N elements into $\sqrt{M/B}$ buckets (the square root effectively only doubling the number of recursion levels), however the constants involved in the $\mathcal{O}(N/B)$ bound are much larger than those of multiway merge sort. It involves a pre-sorting phase, that much like multiway mergesort sorts memory loads (albeit in memory, thus incurring only $2N/B$ I/Os) *before* the elements are distributed in buckets and then sorted recursively (again), so the instruction count also has a high leading term constant.

3.2.4 Cache-Oblivious Sorting

The same lower bounds hold for a cache-oblivious algorithm as for an external memory algorithm, so the goal of optimal cache-oblivious sorting algorithms is to match the I/O bound of Theorem 3-1 and the work bound of $\mathcal{O}(N\log N)$, while not referring explicitly to the memory system as Algorithm 3-3 does. Using the tall cache assumption that $M = \Omega(B^{(d+1)/(d-1)})$, we can rephrase this bound in the cache-oblivious model. We have

$$\begin{aligned}
\Omega\left(\frac{N}{B}\log_{M/B} N/B\right) &= \Omega\left(\frac{N}{B}\frac{\log N/B}{\log M/B}\right) \\
&= \Omega\left(\frac{N}{B}\frac{\log N/B}{\log M^{1-\frac{d-1}{d+1}}}\right) \\
&= \Omega\left(\frac{d+1}{2}\frac{N}{B}\log_M N/B\right) \qquad (3.10) \\
&= \Omega\left(d\frac{N}{B}(\log_M N - \log_M B)\right) \\
&= \Omega\left(d\frac{N}{B}\log_M N\right)
\end{aligned}$$

Since all RAM algorithms are also cache-oblivious algorithms, the popular and efficient quicksort is also a cache-oblivious sorting algorithm. Indeed, it follows recursive divide-and-conquer strategy, which is intuitively good for cache locality. Efficient implementations use a constant time median approximating scheme to partition the elements in a set of small elements and a set of large elements, and then recurse on each set. The partitioning of n elements can be done with $\mathcal{O}(n)$ operations and $\mathcal{O}(n/B)$ memory transfers without knowing B. At each level, a total of N elements are partitioned for a total of $\mathcal{O}(N)$ operations and $\mathcal{O}(N/B)$ memory transfers. When considering the work done by the algorithm, the recursion continues until a constant number of elements are left in the set, for a total of $\mathcal{O}(\log N)$ recursion levels. However, for the I/O complexity, by virtue of the recursive structure, when $n < M$ the partitioning will incur no more memory transfers for that particular subproblem. The number of recursion levels to consider for the I/O complexity is thus expected to be the number of times N can be halved before it becomes smaller than M, namely $\mathcal{O}(\log N/M)$. Hence, quicksort does $\mathcal{O}(N\log N)$ work, incurs $\mathcal{O}(N/B\log(N/M))$ memory transfers and is thus fortunately not asymptotically optimal. However, it does come very close, which it is a testament to the efficiency of quicksort.

We saw that in practice, multiway mergesort performs no more than $4\lceil N/B\rceil$ I/Os. [LL99] estimates the factor on the $\lceil N/B\rceil$ term in quicksort to be $2\ln(N/M)$ on uniform data when counting cache-misses. With M half a gigabyte and N two gigabytes and including writes, this constant is roughly 5.54, so when sorting less than two gigabytes, quicksort incurs no more than 38.6% more memory transfers than the optimal multiway mergesort. Obviously, on the lower levels of the memory hierarchy $2\ln(N/M)$ can get higher for reasonable N, however the penalty of being suboptimal on those levels are not as high.