AN EFFICIENT EXTERNAL-MEMORY SORTING ALGORITHM

An Undergraduate Research Scholars Thesis

by

GABRIEL STELLA

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Dmitri Loguinov

May 2019

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

An Efficient External-Memory Sorting Algorithm

Gabriel Stella
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Dmitri Loguinov
Department of Computer Science
Texas A&M University

External-memory sorting is a well-versed subject, with a history going back several decades. However, current implementations of external-memory sorting algorithms are not able to fully take advantage of the power of modern hardware. The reason for this is twofold: (1) they use a computationally-expensive mergesort approach, which can create a bottleneck during processing; (2) their management of I/O resources prevents them from achieving maximum I/O bandwidth. In this paper, we outline the construction of an external-memory distribution sort algorithm that utilizes available resources much more effectively.

LIST OF FIGURES

1. INTRODUCTION

The problem of sorting is central to computer science. It is a common subroutine in other algorithms, helping to better structure data and thereby reduce runtime. The branch of in-memory sorting in particular has been thoroughly studied, and efficient implementations exist for many different in-memory sorting algorithms. These are not just asymptotically optimal; they also seek to reduce program runtime.

In the age of Big Data, external-memory algorithms are a vital part of many applications. These algorithms are designed to interact with secondary memory devices, such as hard disk drives (HDDs), when data is too large to fit in main memory. This interaction brings with it many challenges; HDDs are slow, sequential devices, not at all like RAM. The design of algorithms for external memory has a long lineage, including many algorithms for external-memory sorting [1], [2], [4], [5], [6], [7]. However, current implementations are unable to achieve maximum execution speed. Part of the reason for this is their complicated designs, including additional abstraction layers and overuse of expensive function calls, which can substantially increase computational overhead.

Another cause is the inaccuracy of the models that these implementations are based on. One common model is the parallel disk model used in [5]. This model attempts to account for the properties of D parallel HDDs in two ways. First, they note that transfers between main memory and secondary memory occur in blocks: assuming $S(x)$ denotes the speed of a transfer of x records, $S(x) = S(\lceil x/B \rceil B)$ where B is the block size for the particular external-memory device. This implies, for example, that $S(1) = S(B)$. Second, the model allows the transfer of D blocks concurrently using the parallel disks, so that $S(B) = S(DB)$. This novel feature of the model is not always applicable nowadays as multi-disk systems are often organized into RAID or some other system that hides the disks behind a single-disk interface.

As mentioned in [1], a much more realistic feature of HDDs is missing from this model: the problem of highly-expensive disk seeking operations. The model assumes that after an external-memory access to the block beginning at location x, an access at $x + B$ (the next block) is as expensive as an access *anywhere else on the disk*. In practice, use of HDDs in a random-access pattern significantly decreases throughput. The goal of reducing seeking is, in fact, *at least* as important as reducing the number of I/O operations.

Our approach to the problem of external-memory sorting is different from other works in that we explicitly aim to reduce runtime, rather than reducing the I/O complexity as in previous works. Simply dealing with the above problems is not enough to accomplish this; the specifics of the underlying hardware must be taken into account. By testing the system, a model can be created to predict the algorithm's performance with different parameters and calculate optimal settings.

Our sorting method consists of an automated optimizer, which runs tests and creates the system's performance model, and an external-memory key distribution algorithm. The optimizer uses system-specific performance curves to calculate an optimal recursive tree of input splitting. At each level of this tree, our key distribution algorithm aims to reduce seeking by using a highly effective buffer grouping approach. At the bottom of the tree, all files will be small enough to be sorted in RAM and appended to the final output file.

2. CHALLENGES AND OBJECTIVES

Working with external memory comes with many challenges. The work done in [5] solves several of these challenges; the work in this paper aims to solve several more. Our work also diverges in that our implementation attempts to explicitly optimize runtime, as opposed to optimizing the asymptotic number of I/Os.

2.1 Processing Time

While the main focus of an external-memory algorithm is to optimize along the I/O route, if one is not careful, processing time may become a substantial portion of the algorithm's runtime. This is especially true as external memory systems get faster; in our tests, the speed of a single-threaded splitter (without doing any I/O) was comparable to the maximum single-file bandwidth of a 24-disk RAID system.

Previous works attempt to optimally overlap I/O and computation so that this problem is mitigated [4]. Our algorithm continues this trend and employs a cheap bitwise splitting calculation that allows the key distribution thread to process several hundred million keys per second.

2.2 Large Files

Given an input file with N keys, and main memory that can store R keys, an external-memory sorting algorithm may deal with files such that $N \gg R$. To solve this problem, one must break apart the original file F into many smaller pieces. In our algorithm and in [5], a distribution sort is implemented to deal with this challenge. The distributor recursively splits files until they fit in RAM and can be sorted using an in-memory algorithm.

It is important to consider how each piece of the original file will be dealt with once the file is broken up. If the file is broken with no pattern, expensive merge operations must be used to create the sorted file. However, when using a distribution approach that ensures the relative order of keys in different files, the sorted file can be created via simpler means. Our distributor provides the following guarantee: $i < j \rightarrow \max(f_i) < \min(f_j)$ for intermediate files f_i, f_j of same depth, where i and j are the indices of the files corresponding to a split-tree traversal. Thanks to this guarantee, we can reconstruct the sorted file by simply appending each intermediate file in this order. This is discussed more in Section 4.2.

2.3 Amount of File I/O

An obvious constraint, given that external-memory I/O is so much slower than main memory, is that any external-memory algorithm should aim to reduce the amount of I/O done. As discussed in Section 3.1, a recursive algorithm that deals with a tree of files will have $O(N \log N)$ I/O operations. Producing an algorithm with this bound is the main focus of [5]; however, simply reducing this bound is not enough to produce an optimized algorithm. Our algorithm is able to minimize the number of I/Os in that at each level, it requires only a single pass over the N keys. We can do this because we assume that keys are uniformly distributed, and so we do not need to calculate partitioning elements from the data.

2.4 Disks are Sequential

The reality of working with external memory is much more complicated than simply reducing the number of I/O operations. The sequential nature of HDDs implies that the order of operations can drastically impact the speed with which they are completed; in other words, we have to deal with the problem of highly-expensive seeking operations. This is the main issue that will be dealt with by our key distribution algorithm.

This important problem is not given proper treatment in prior works. The parallel disk model assumes that disks are essentially random-access devices; the method employed to reduce seeking in [5] is simply to make B, the block size when interacting with the disk, large. However, this is not enough to reduce seeking; in modern systems, optimal block sizes for single-file access may be several Megabytes, while using this block size for multiple-file access (and naively switching between each file) would result in incredible seek delays.

2.5 Limited RAM

The problem of limited RAM is related to the previous problem. When deciding the order of I/O operations, we are also constrained by a small amount of main memory with which to buffer disk accesses. With more RAM, we can store more buffers, which (using the algorithm described in

Subsection 4.1.3) allows us to significantly reduce seeking. We therefore have to choose parameters based on our available resources, which involves a tradeoff between k, B, and the number of buffers C. The method we use is discussed in Subsection 4.2.1.

2.6 Objectives

With these considerations in mind, we more explicitly describe the goal of our algorithm. In order to sort, we construct a list of distribution factors $\{k_1, k_2, ..., k_d\}$ such that $\prod_{i=1}^{d} k_i = N/R$. This is equivalent to calculating a list $\{b_1, b_2, ..., b_d\}$ of bit counts used to split, where $k_i = 2^{b_i}$, such that $\sum_{i=1}^{d} b_i = \log_2 N - \log_2 R$. The constraints based on N and R result in bottom-level buckets small enough to be sorted in RAM[1].

To sort optimally, we must choose the split factors $\{k_1, k_2, ..., k_d\}$ such that $cost = \sum_{i=1}^{d} t(N, R, k_i)$ is minimized, where $t(n, R, k)$ is the time taken to distribute a file of size n into k buckets with main memory size R. The optimizer we implement runs tests on the system to estimate the function t then uses a simple dynamic programming algorithm to automatically calculate the optimal split sequence for any sort, given parameters N and R. This effectively tailors the method to the specifics of any system it runs on, which is vital in ensuring good performance. The next goal is to design an efficient distribution algorithm to decrease the value of t on arbitrary inputs.

[1]We omit the factor δ, assuming that internal-memory sorting will use an in-place algorithm.

3. BACKGROUND

There are two main approaches to external-memory sorting: file merging and file distribution. These methods work better than other approaches from in-memory sorting because they can be designed to access the disk in a sequential manner.

After creating all of the bottom-level files, either after distribution or before merging, an internal-memory sorting algorithm must be used. In general, this has to be a comparative sorting algorithm such as quicksort. If uniformly-distributed integer keys are assumed, as in our case, one can use a faster non-comparative algorithm such as radix sort. See Figure 3.1 for an illustration of the four different combinations. Our algorithm falls under the distribution sort category, and we can integrate a non-comparative sort for in-memory sorting of buckets. This is the fastest of the four combinations shown. The algorithm in [5] is also a distribution sort, while the algorithm in [4] is a merge sort; both use comparative algorithms for in-memory sorting.

3.1 File Merging

A common approach taken to external-memory sorting is similar to mergesort. In this approach, F is broken apart into m files of at most δR records each, where R is the number of records that can be stored in RAM and δ is some constant that accounts for extra space used by the in-memory sorter. These files, typically called *runs*, are all sorted individually and then progressively merged, k at a time. This results in a merge tree of depth $d = \log_k m$. Since $O(N)$ records are read and

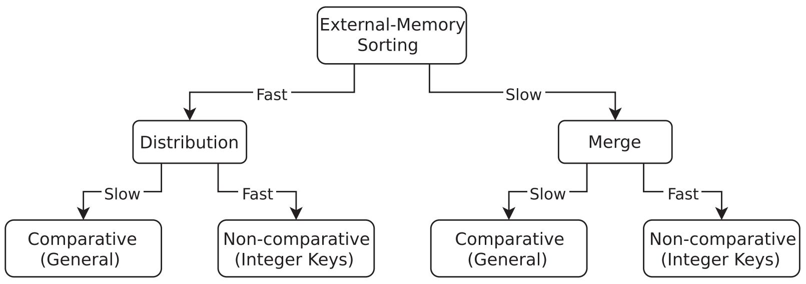

Figure 3.1: Comparison of EM Sorting Methods.

written at each level of the tree, $O(N \log_k m) = O(N \log_k (N/R))$ records are read and written in total.

The merging approach has the distinct benefit that it is resilient against skew in the input data. In other words, no matter what the contents of F are, I/O cost depends only on N. However, in-memory merging is an expensive process: merging k inputs each of size n requires $O(nk \log k)$ CPU steps. This also demonstrates that the parameter k must be optimized based on both CPU and I/O considerations: increasing k will reduce d, but will increase computation time. The following approach, based on a distribution sort, holds slightly different properties.

3.2 File Distribution

Another approach is to traverse *down* this tree by splitting rather than merging. At each level, the keys to be sorted are distributed into k buckets (each bucket corresponding to a file). This results in the same amount of file I/O as the merge-based approach; however, when the records have integer keys, the computational complexity of splitting a single file of size n into k files each of approximately size n/k is $O(n)$, as opposed to $O(n \log k)$ for an equivalent merge operation. Along with the decrease in CPU usage, this allows us to choose parameters based solely on I/O optimization.

This distribution sort is the approach described in [5]. Their definition of the sorting problem differs from ours in an important manner: they generalize to any comparable data type, whereas we sort numerical keys. This changes the method used for splitting keys. In [5], a preprocessing step is required at each level to determine a set of "partitioning elements", which are then used to split the entire file. Then, for each element to be distributed, a binary search over the partitioning elements is required to determine which bucket to place it in. Our algorithm instead splits directly on the bits of each numerical key. This is a fast process, as it requires only simple bitwise operations to generate the index for each key, and it reduces the amount of required I/O. As long as keys are uniformly distributed, which is a realistic assumption for our use cases, the produced buckets will also be uniform.

3.3 Choosing k

The choice of k is important whether distributing or merging. Previous works derived formulas to determine k based on N, R, and B (the block size when interacting with external memory) without taking into consideration the specifics of the underlying hardware. In STXXL, a C++ library for external-memory data structures and algorithms, the merge factor is chosen using the following formula[2]:

$$x = \lceil \ln m / \ln(R/B) \rceil = \lceil \log_{(R/B)} m \rceil \tag{3.1}$$

$$k = \lceil m^{1/x} \rceil \tag{3.2}$$

In [5], the formulas for k were derived with the motivation of reducing the asymptotic I/O complexity of the problem and ensuring equal use of D parallel disks. Their distribution factor is chosen using one of two approaches, based on N. In the case when N is large, k is calculated from x:

$$x = \frac{\sqrt{R/B}}{\ln^2(R/B)} \tag{3.3}$$

So that k is the maximum of 1 and the largest power of 2 that is $\leq x$, i.e. $k = \max(1, 2^{\lfloor \log_2 x \rfloor})$. When N is small, the formula is simpler: $k = 2N/R + 1$.

The approach we use to calculate k is based on a dynamic programming algorithm meant to optimize runtime. Our method uses information gathered about the system, as is described in Subsection 4.2.1.

[2]This formula was extracted from STXXL version 1.4.1 code.

4. ALGORITHM

In this chapter, we outline the development of our external memory sorting algorithm. This algorithm has several subcomponents; the most central one is the key distributor. To construct a sorting algorithm using this, we organize a multi-level distribution sort. At each level, it splits files into buckets (intermediate files) based on a small number of bits, then recursively applies itself to each bucket. Eventually, these buckets will be either completely sorted[3] or small enough to be sorted in memory. Then, they are all recombined in proper order.

4.1 File Splitting

The process of key distribution can be described as follows. Let f be a parent file in the tree at level $i \in [0, d-1]$. We apply the key distribution function to f using split factor k_{i+1} to generate that many buckets (child files). We then apply the function recursively to each of those buckets. Beginning with F at level $i = 0$, this results in DFS-order creation of the splitting tree, allowing us to (1) reuse intermediate files, thereby decreasing storage requirements, and (2) create the sorted file F' by simply appending all level d buckets in order of creation.

We begin by discussing the operation in terms of a single level, i.e. the functionality of the file splitter. The goal of the file splitter is to read a single file and split the data into 2^b output files. Let us first focus on the operation of the input splitter and file writer system, for now ignoring the specifics of file reading. For this setup, we assume the existence of some large input buffer, without any knowledge of how it was created. We show how to incorporate file reading in Subsection 4.1.5.

For subsections 4.1.1 to 4.1.3, we use the following definitions: let R be the size of RAM (specifically, the maximum number of keys the splitter may store in memory), N the number of keys in the input buffer, b the number of bits to split on, $k = 2^b$ the number of files to split into, B the size of memory buffers, and C the number of memory buffers (such that $R = BC$). In Subsection 4.2.1 we show how to optimally calculate these parameters.

The basic operation of a key splitter is shown in Algorithm 1. The specific details of the

[3]This implies the bucket contains copies of a single unique element.

`write(buffer)` function are described later; this is the focus of the following stages of development. In general, the splitter loops through the input buffer, places each key into the correct output buffer based on some function (a simple operation to extract b bits, which correspond to the index), and writes buffers when they are full. In the following sections, we develop this basic algorithm into a high-performance multi-file splitter, then show how to incorporate it into an external-memory sorting algorithm.

4.1.1 Naive Splitter

The most naive implementation of a splitting algorithm is a single-threaded splitter that writes one block at a time, as soon as any buffer fills up. It then waits for the I/O request to complete, continues splitting, and so on. This is very obviously not optimal, as it forces both splitting and I/O to constantly wait on each other. A better algorithm would parallelize these two operations.

4.1.2 Splitter-I/O Parallelism

The parallel system runs two threads, one for splitting keys and one for writing to the output files. The general idea is that the threads share a set of output buffers; the splitter writes to the buffers, and the I/O thread writes the buffers to the disk. We generally allocate $C = 2k$ memory buffers[4], so $B = R/2$.

When the splitter thread fills a buffer, it puts the buffer and its file index into a queue (`qFull`) to go to the I/O thread, then requests an empty buffer from another queue (`qFree`) which is populated by the I/O thread. The I/O thread waits for buffers to be put into `qFull`, issues a single write request per buffer, and enqueues the newly-freed buffer into `qFree`. This is an archetypal bounded producer-consumer problem, so the basic solution is trivial. To encapsulate this behavior, let us define a simple API as seen in Algorithm 2. In this scheme, the `write(buffer)` function from earlier is replaced by calls to `push_full` and `pop_free`.

As stated, a basic implementation of these functions is trivial. However, in anticipation of the next algorithm, we will define them in a non-obvious way by implementing a shared list of buffers for each file, called `buffers`. To be explicit, let `lockFree` and `lockFull` be two mutex

[4]This helps ensure double-buffering, which allows both threads to work concurrently.

Algorithm 1: Basic Key Splitting Algorithm

```
Func split (Key[] input) returns None
    for (i = 0; i < N; i++) do
        index = index_of(input[i])    ▷ Decide which file this key goes to
        offset = offsets[index];    ▷ The position at which to insert this element in its buffer

        outputBuffers[index][offset] = input[i];    ▷ Insert at current end of buffer

        offset++;    ▷ Assume this updates the value of offsets[index]
        if (offset == C) then    ▷ Check if buffer is full
            write(outputBuffers[index]);
            offsets[index] = 0;    ▷ Reset the buffer position
```

Algorithm 2: Simple Parallel Buffer Exchanging API

```
// Pass a full buffer from the splitter to the writer
Func push_full (Key[] buffer, integer fileIndex) returns None

// Retrieve a full buffer from the splitter
Func pop_full () returns (Key[], integer)

// Pass a free buffer from the writer to the splitter
Func push_free (Key[] buffer) returns None

// Retrieve a free buffer from the writer
Func pop_free () returns Key[]
```

locks and `semaFree` and `semaFull` be two semaphores (each of maximum value C). In this implementation, `qFull` only keeps track of file indices. See Algorithm 3. The implementations of `push_free` and `pop_free` are the same as in a basic implementation, i.e. they do not interact with the shared buffer list, so they are omitted here.

By parallelizing the splitting and I/O, the total time taken becomes $\max(splitter, I/O)$, while the naive implementation requires $splitter + I/O$. Next, we will see how to make I/O faster.

4.1.3 Grouping

One small improvement to the previous method produces a significant I/O speedup. This is to group full buffers by their file index. Once buffers are grouped, the I/O thread can `pop` a file index i from `qFull` and issue write requests for all buffers corresponding to i. The modifications to

Algorithm 3: Parallel Buffer Exchanging Implementation

```
// Pass a full buffer from the splitter to the writer
Func push_full (Key[] buffer, integer fileIndex) returns None
    Lock(lockFull);

    qFull.push(fileIndex);
    buffers[fileIndex].add(buffer);

    Unlock(lockFull);

    Release(semaFull);

// Retrieve a full buffer from the splitter
Func pop_full () returns (Key[], integer)
    Wait(semaFull);

    Lock(lockFull);

    integer fileIndex = qFull.pop();
    Key[] buffer = buffers[fileIndex].remove();      ▷ Remove a single buffer from the list

    Unlock(lockFull);
    return (buffer, fileIndex);
```

the buffer-passing process are detailed in Algorithm 4. Note the size test on line 4. The modified behavior is that each element in `qFull` does not refer to a new full buffer; it refers to a file that has one or more buffers ready to be written. Therefore, when a buffer is filled, i only gets added to the queue if the buffer is *the first for its file* since `buffers[i]` was emptied by the I/O thread. Consequently, `semaFull` counts the number of files that have buffers ready.

This modification produces many benefits. It drastically decreases the amount of seeking; as is shown in Section 5.7, the grouping algorithm is able to reach a stable rate of writing $\frac{2}{k+1}C$ buffers between every seek. Due to this fact, the I/O thread must quickly release freed buffers back to the splitter thread once any write request has completed. This helps retain the double-buffering property of the previous algorithm; without this feature, the grouping algorithm may even become slower than the non-grouping one.

4.1.4 Notes on File Writing

There are several important things to note about the methods we use to write to files. The main goal of the splitter's design is to reduce seeking; for that reason, we try to write as much as possible

Algorithm 4: Parallel Buffer Exchanging Implementation With Grouping

```
// Pass a full buffer from the splitter to the writer
Func push_full (Key[] buffer, integer fileIndex) returns None
    Lock(lockFull);

    bool pushValue = buffers[fileIndex].isEmpty();    ▷ Buffer list size test

    buffers[fileIndex].add(buffer);
    if(pushValue) qFull.push(fileIndex);

    Unlock(lockFull);

    if(pushValue) Release(semaFull);

// Retrieve a full buffer from the splitter
Func pop_full () returns (List<Key[]>, integer)
    Wait(semaFull);

    Lock(lockFull);
    integer fileIndex = qFull.pop();

    List<Key[]> allBuffers = buffers[fileIndex];    ▷ Take all buffers from the list
    buffers[fileIndex] = List();    ▷ Put back an empty list

    Unlock(lockFull);

    return (allBuffers, fileIndex);
```

to a single file at once. It is also important to never issue write requests to two files at once; before making requests to a different file, all open requests must be completed.

We use asynchronous (overlapped) I/O requests because in some cases they can improve I/O speed. The principle behind using asynchronous calls is to create a "sliding window" of some small, fixed number (4-8) of pending requests that travels along the list of requests to be made. When the oldest request finishes, a new request can be made at the "front" of the window. The overhead of this method is negligible, and it should always be at least as fast as non-overlapped I/O. When the delay associated with making a write request is not negligible compared to the delay of the request's completion, having multiple requests issued concurrently can increase throughput.

4.1.5 File Reading

As covered in the previous section, one major goal of our algorithm is to reduce seeking. In order to do this, when incorporating file reading into the splitter, we must prevent the reader and writer from working concurrently. In our design, the I/O system runs two threads, one for reading and one for writing. Both threads share a mutex, `lockIO`. The writer thread will wait for `semaFull`, lock `lockIO`, write the buffers for a single file, release `lockIO`, and repeat. The reader thread simply runs a loop in which it locks `lockIO`, reads all currently-free reader buffers, and releases `lockIO`.

4.1.6 EM-Flux

Splitting one file into k files for sorting is a specific formulation of a general pattern in which l input files are streamed into memory and used for computation, resulting in data being streamed into k output files. One of the goals of our project, beyond writing a high-performance sorting algorithm, is to work towards creating a system to provide this functionality. Once this is complete, splitting can be redefined as a simple application of this pattern, with the details of external memory management handled transparently. The platform we are building to support this functionality is called *EM-Flux*. Our framework will allow the user to specify a list of input and output files and supply a simple stream interface to interact with each file. This will greatly simplify sorting-related tasks, most obviously merging and distributing, and has many more possible applications.

4.2 The Sorting Algorithm

With the multi-file key distribution system fully functional, constructing a sorting algorithm is relatively simple. The sorting algorithm simply applies the key distribution algorithm recursively to each bucket until one of two criteria is met: (1) the bucket is entirely sorted, (2) the bucket is small enough to be sorted in RAM. If a bucket is created that meets one of these criteria, it is sorted in memory (if necessary) then appended to F', which is initially empty. The construction of our algorithm guarantees that F' is generated with keys in ascending order. This allows it to skip any expensive merging operations that may be necessary in other systems.

This guarantee comes from the order in which the splitter looks at bits: most significant first. The splitter at level i will skip the most significant q bits (which have already been looked at by previous levels), where $q = \sum_{x=1}^{i-1} b_x$. It will then split using the next most significant b_i bits, guaranteeing the property discussed in Section 2.2 that $i < j \rightarrow \max(f_i) < \min(f_j)$. The choice of ordering (files are split in DFS order, i.e. the system recurses on f_i before moving to f_{i+1}) makes this property simpler to implement.

The question now is the choice of b, C and B; specifically, at each level i, we must decide b_i, $C_{i,out}$ and $B_{i,out}$ for output buffers, and $C_{i,in}$ and $B_{i,in}$ for input buffers. These values are chosen by an automated optimizer.

4.2.1 Automated Optimizer

The performance of the splitter depends upon the specifics of the system it is running on. Therefore, to generate models and decide the values of b, C and B, we must run tests on the system to determine its behavior. We can then use these tests to extrapolate optimal settings for the splitter. At the core of this decision is a simple dynamic algorithm that determines the set of split factors $\{b_1, b_2, ..., b_d\}$ based on the rate that the system can support for any given split factor. Then, once b_i has been chosen, we ensure that $C_{i,out} \geq 2k_i$ with the added constraint that B_i should be no larger than necessary for good performance. To determine these values more specifically, we run two sets of tests.

The first test determines the speed of the file system when writing to a single file using different buffer sizes. This lets us discover the maximum necessary buffer size, B_{max}. This test begins with buffers of 2^{10} B $= 1$ KiB and progresses through each power of 2 until a maximum rate is reached; this final buffer size will be B_{max}. Once this value is known, we can choose general B and C according to Algorithm 5. The choice of r, the amount of RAM dedicated to input buffers, is explained in Section 5.5.

The motivation for setting $C > 2k$ when $2kB_{max} < R$ is that dealing with larger C helps ensure that there is a constant flow of data through the program (e.g. so neither thread, splitter or I/O, has to wait as often). As long as the cost of passing buffers between threads is negligible, and the buffers are large enough to achieve maximum I/O bandwidth, using more buffers is better.

This becomes important when the splitting thread begins to wait on the I/O thread. For a simple example, take $b = 1$, $k = 2$, and examine the cases where $C = 4 = 2k$ and $C = 100 > 2k$. Given that $B = R/C$, in the first case $B = R/4$ and in the second $B' = R/100$; that is, $B = 25B'$. Now, imagine that there are no buffers in `qFree`, and the I/O thread is about to write $C/2$ of the buffers to one of the files. Remember that the grouping algorithm will release each buffer as soon as its request has finished. In the first case, the splitter thread must wait 25 times as long as in the second case before it can start writing again. In practice, this can significantly increase the amount of time that the splitter thread spends waiting in `pop_free`, noticeably reducing overall speed.

We now have to choose b_i. The next test determines the speed of the splitter system, running a single-level split with different values of b in a small range (beginning at $b = 1$). This test uses Algorithm 5 to choose C and B based on the value of b being tested. The results of this test are used to compute the fastest multi-level split sequence for any number of bits using the dynamic programming algorithm shown in Algorithm 6. We can recompute this dynamic algorithm at each b while running the test, then stop testing once the single-level split rate drops below the multi-level split rate prediction.

The limiting factor when determining splitting speed is the size of RAM, R. A larger R allows us to increase C, reducing seeking. Consider that, holding C constant, every unit increase in b doubles the number of files k. This effectively halves the number of buffers available to each file. Once seeking becomes noticeable, this will increase its effect. However, doubling R would offset this effect, restoring the number of buffers per file. Thus, assuming S is the splitting speed, it should be that $S(b, R) \approx S(b + 1, 2R)$. In Section 5.4, we show that this is the case. This result allows us to sample the splitting curve using a single value of R and calculate rates for any other value. We can then run the dynamic algorithm on these new rates.

Algorithm 5: Determining C and B given b and R

```
Func Dynamic_Buffers(b, R) returns ($C_{out}, B_{out}, C_{in}, B_{in}$)
    $r = \frac{1}{4}$    ▷ Input buffer allotment
    $C_{max} = R/B_{max}$

    $R_{out} = (1-r)R$
    $C_{out} = (1-r)C_{max}$
    $B_{out} = B_{max}$    ▷ Equivalent to $R_{out}/C_{out}$

    if ($C_{out} < 2^{b+1}$) then    ▷ Ensure $C_{out} \geq 2k$
        $C_{out} = 2^{b+1}$
        $B_{out} = R_{out}/C_{out}$

    $R_{in} = rR$
    $B_{in} = B_{max}$
    $C_{in} = R_{in}/B_{in}$

    return ($C_{out}, B_{out}, C_{in}, B_{in}$);
```

Algorithm 6: Determining b for a number of bits

```
// Calculate the dynamic table up to the given number of bits
Func Dynamic_table(rates, bits) returns (b[])
    len = length(rates) - 1

    times = double[len + 1]
    for (i = 1; i <= len; i++) do
        times[i] = 1/rates[i]

    table = int[bits + 1]    ▷ The dynamic table of split factors
    table[0] = 0
    timeTable = double[bits + 1]    ▷ Holds runtime predictions for splitting any number of bits
    timeTable[0] = 0

    for (i = 1; i <= bits; i++) do
        minb = -1
        minTime = $\infty$

        for (b = 1; b <= len; b++) do    ▷ Traverse the table backwards
            j = i - b
            if (j >= 0) then
                time = timeTable[j] + times[b]
                if (time < minTime) then
                    minb = b
                    minTime = time
            else
                break

        table[i] = minb
        timeTable[i] = minTime
    return table;

// Calculate a specific split sequence using a dynamic table
Func Dynamic_bs(table, bits) returns ($\{b_1, b_2, ..., b_d\}$)
    bs = List()

    for (bitsRemaining = bits; bitsRemaining > 0;) do
        b = table[bitsRemaining]
        bitsRemaining -= b

        bs.add(b)

    return bs;
```

5. RESULTS

In this chapter, we present the results that motivate the structure of our sorting algorithm. They are presented in an order that roughly reflects the process of testing the optimizer goes through, and also demonstrates the development process of our algorithm.

All of these tests were run on a machine with 32 GB of RAM and a 4 GB/s RAID.

5.1 Single-File Output

For completeness, we present results demonstrating different methods of writing to a single file. The three methods we present are: non-overlapped I/O (called "Simple" in the plot), concurrent overlapped requests ("Overlapped"), and a high-performance I/O library developed by the Internet Research Laboratory here at TAMU ("Stream"). The results for single-file writing speed using blocks of size B_{max} are shown in Figure 5.1. On this particular system, all three methods result in about the same transfer speed; on other systems, non-overlapped I/O may be significantly slower. The "Overlapped" method was used to get results for later sections, as the streaming framework does not yet support our multi-file output algorithm.

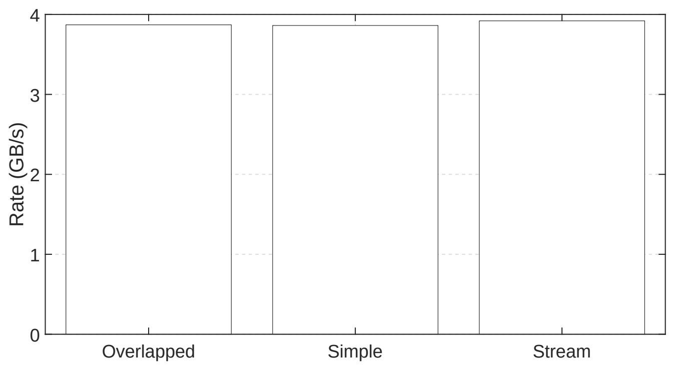

Figure 5.1: Comparison of File-Writing Strategies.

5.2 Finding B_{max}

To find the value of B_{max}, we run a simple single-file write test on the machine. Figure 5.2 shows an example of the resulting data. There is a clear plateau after $\log_2 B = 20$; thus, in this example, $B_{max} = 2^{20}$.

5.3 Multi-File Output

To make the splitter as fast as possible, we had to investigate different ways of writing to multiple files concurrently. As was covered in Section 4.1, the best method we found was to group writes together. This method allows us to significantly reduce the amount of switching between files, which consequently reduces seeking. We compare this method to three others here: a similar non-concurrent method that doesn't group (the algorithm detailed in Subsection 4.1.2), a naive algorithm that does not parallelize I/O (detailed in Subsection 4.1.1), and an algorithm that writes concurrently to all files. Respectively, these methods are referred to as "Sync Grouping", "Sync Non-Grouping", "Simple", and "Concurrent". The results can be seen in Figure 5.3. The grouping method is clearly much more effective than the others. These results were obtained with $R = 8$ GiB, and C and B were chosen using Algorithm 5 from Subsection 4.2.1. Note that for these tests, file input was not enabled, so $r = 0$. The splitter simply read repeatedly from a large memory buffer filled with uniformly-distributed random numbers.

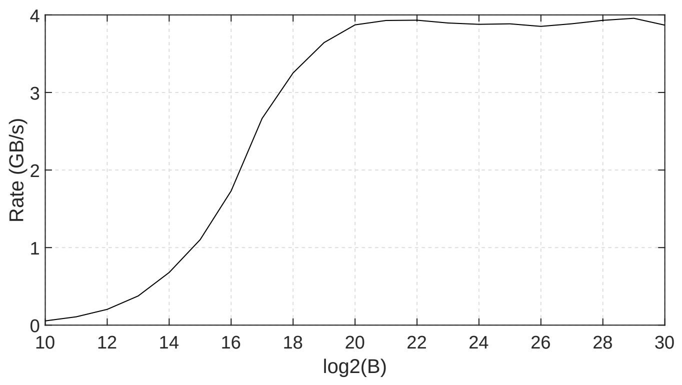

Figure 5.2: Determining B_{max}.

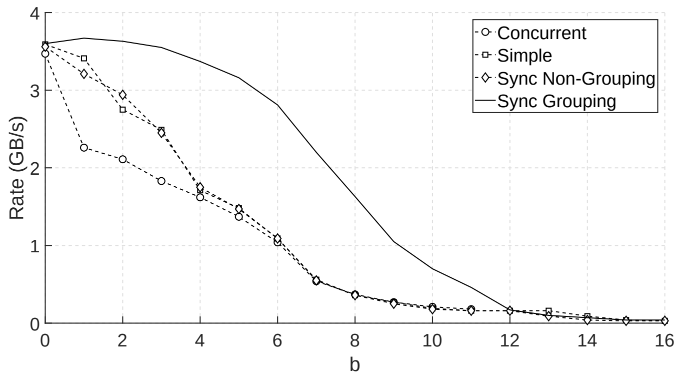

Figure 5.3: Multi-File Output Algorithms.

5.4 Impact of R on Splitting Speed

An important part of the model is to optimize given any system constraints, including RAM size R. Figure 5.4 shows the speed of our file splitter for four different values of R, each double the previous. It can be seen that the effect of doubling R is essentially to slide the curve to the right by one unit of b. Input keys for this test, as well as the choices of C and B, were made in the same way as in the previous test. If file input is enabled, the relationship between the curves deviates from this behavior when b is low; we will investigate this relationship more in future work.

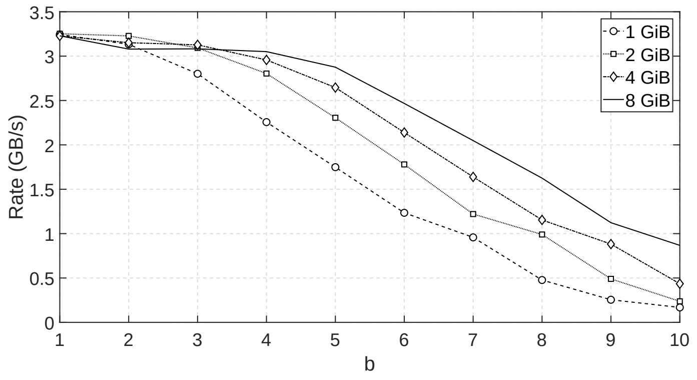

Figure 5.4: Splitter Speed with Varying R.

5.5 Impact on r on Splitting Speed

The splitter's parameter r determines the proportion of main memory that is reserved for file input buffers. Given that the grouping algorithm writes $2C_{out}/(k+1)$ buffers per seek, it may seem that the reader should read that many buffers per loop. However, this seems not to be the case; in our tests, a static reservation of $r = \frac{1}{4}$ performed the best. See figures 5.5 and 5.6. For these tests, the splitter was run on data read from a large (≥ 64 GiB) file of random numbers. In some of our earliest tests, we found that RAM spent on input buffers generally produced a larger performance increase than RAM spent on output buffers; these results may be related to that observation. We leave possible further investigation of r to future work.

5.6 Splitting

When b is large, the performance of the splitter drastically reduces. The reason for this effect is twofold. First, beginning at mid-range b, the splitter has to seek more. Then, at larger b, in order to satisfy $C \geq 2k$, it must be that $B < B_{max}$. These both contribute to a drastic reduction in performance. To remedy this, our algorithm includes an optimizer that can determine optimal *multi-level* split sequences for any b. Figure 5.7 demonstrates the quick descent of a single-level splitter, while the predicted performance of a multi-level splitter remains relatively high and is much more stable as b changes. The marked point on the figure after which Single-Level performance drops below

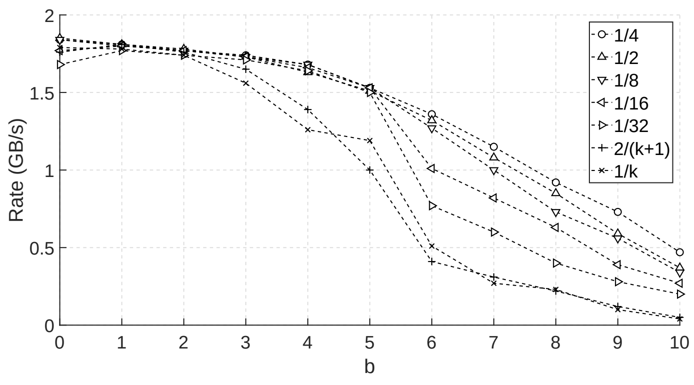

Figure 5.5: Splitter Speed with Varying r, R = 8 GiB.

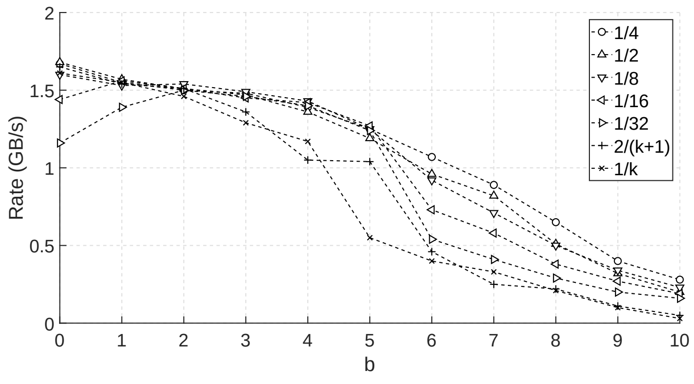

Figure 5.6: Splitter Speed with Varying r, R = 4 GiB.

Multi-Level performance denotes the last optimal single-level split value[5]. After testing the next value of b and seeing that the single-level split speed is less than the optimal multi-level split speed, the optimizer will halt its tests, knowing that no further single-level split could be optimal. Further points are shown here simply for completeness. The shown single-level points were collected using the same settings as the previous test, set at $r \approx \frac{1}{8}$, and were directly used to generate the multi-level prediction curve shown.

[5]Before and at this point, all optimal splits are single-level.

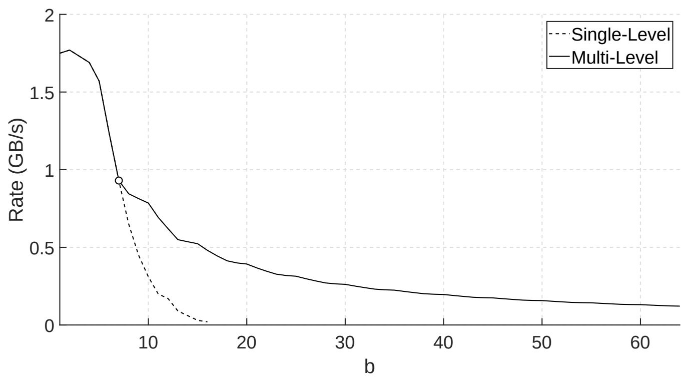

Figure 5.7: Comparison of Single-Level and Multi-Level Splitting Speeds.

5.7 Effectiveness of I/O Grouping

In order to demonstrate the effectiveness of the buffer grouping algorithm, we gathered data about the fraction of buffers written by the I/O thread during every loop. Because our single-file writing is sequential, this is approximately the fraction of the total number of buffers C_{out} that is written between every seek. In Figure 5.8, we compare the observed ("Actual") fraction of buffers written to two functions of k. the first is $1/k$, which would mean that each file has an equal share of the buffer pool. The second is $2/(k+1)$, which is approximately twice the former (k is exponential in b). It can be clearly seen that the actual fraction $\alpha(k) \approx 2/(k+1)$. The plot is log-scaled in the y axis to accentuate the small error, which is likely due to the discreteness of buffer allocation (αC_{out} must be an integer).

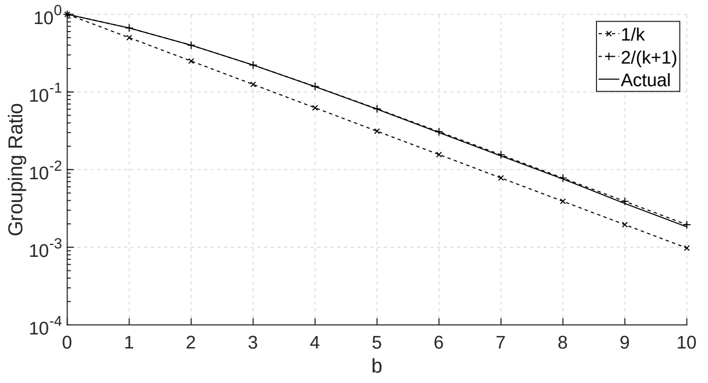

Figure 5.8: Fraction of C Written Between Seeks.

6. CONCLUSIONS

This paper presented an efficient algorithm for sorting uniformly-distributed integer keys in external memory. The overarching goal of the algorithm is to reduce runtime for any sorting job; in doing so, it attempts to solve several problems, most importantly increasing CPU processing rate and I/O throughput.

To increase CPU processing rate, we use a simple bitwise key splitter that is able to process several hundred million keys per second with a single thread. In order to maximize I/O throughput, we group output buffers by file index, which significantly reduces the amount of seeking that is required. To ensure minimal runtime in all cases, we run an automated optimizer that chooses the parameters for each sort.

However, there are still improvements to be made. Our algorithm assumes that its input is uniformly-distributed numerical keys; in future work, we will examine the problem of sorting more general records. There are also several aspects of the performance that we aim to improve. The first is the key distribution process; preliminary tests in parallelization have been able to reach speeds of almost two billion keys per second during in-memory splitting. This will help our algorithm make use of faster external memory devices.

Even with a faster in-memory splitter, we face the problem of slow external-memory splitting when b is large. This is a natural consequence of the limited amount of available RAM. We overcome this problem by applying a multi-level distribution algorithm; however, it is still worthwhile to design algorithms that can scale better to larger b. This is left to future work.

REFERENCES

[1] L. Arge, O. Procopiuc, and J. Scott Vitter, "Implementing I/O-efficient Data Structures Using TPIE," in *Proc. Algorithms — ESA 2002*, R. Möhring and R. Raman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 88–100.

[2] R. Barve, E. F. Grove, and J. Vitter, "Simple Randomized Mergesorting on Parallel Disks," 01 1997.

[3] R. Dementiev, L. Kettner, and P. Sanders, "Stxxl: Standard Template Library for XXL Data Sets," in *Proc. Algorithms – ESA 2005*, G. S. Brodal and S. Leonardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 640–651.

[4] R. Dementiev and P. Sanders, "Asynchronous Parallel Disk Sorting," in *Proc. Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '03. New York, NY, USA: ACM, 2003, pp. 138–148. [Online]. Available: http://doi.acm.org/10.1145/777412.777435.

[5] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory, I: Two-level memories," *Algorithmica*, vol. 12, no. 2, pp. 110–147, Sep 1994. [Online]. Available: https://doi.org/10.1007/BF01185207.

[6] J. Vitter and D. Hutchinson, "Distribution Sort with Randomized Cycling," *Journal of the ACM*, vol. 53, 11 2000.

[7] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Comput. Surv.*, vol. 33, no. 2, pp. 209–271, Jun. 2001. [Online]. Available: http://doi.acm.org/10.1145/384192.384193.