HIGH-PERFORMANCE EXTERNAL-MEMORY MERGESORT

An Undergraduate Research Scholars Thesis

by

NICHOLAS ROBERT

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

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dmitri Loguinov

May 2023

Major: Computer Engineering

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Nicholas Robert, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

High-Performance External-Memory Mergesort

Nicholas Robert
Department of Computer Science and Engineering
Texas A&M University

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

Many current solutions for sorting very large files today are incredibly slow. Given that virtually every widespread application requires some form of sorted data, these slow solutions total a massive waste of time and computing power. When sorting large datasets, the data can come from different files, parts of files, or streams, which poses a problem when trying to create truly high-performance algorithms since the underlying hardware can be slow, specifically in the I/O speed of magnetic drives. Popular solutions used today do not properly consider the performance impact that these I/O devices have on the overall speed. My thesis implements techniques that can reduce and minimize the number of seeks from these HDDs, therefore maximizing the overall performance of the external-memory merge sort algorithm. It also includes several other optimizations for merge rate, such as utilizing large continuous files at the front of the hard drive's address space.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Dmitri Loguinov, for their guidance and support throughout the course of this research.

Special thanks also goes to Arif Armin for developing Origami and giving me personal assistance numerous times on my programming. I also thank Evan Krohn for developing his continuous file program, which was used to reach higher speeds in my own research. Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Part of the code used for High-Performance External-Memory Mergesort was provided by Arif Armin, namely his Origami project. The strategy used in High-Performance External-Memory Mergesort was provided by Dr. Dmitri Loguinov.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work received no funding.

1. INTRODUCTION

Data holds an important place in today's world. In virtually every field, there exists a need for processing the relevant data to inform actions and decisions to improve, advance, maintain, or build in that field. This is easily understood when considering the present-day impact and widespread use of computers and the internet in general. The growth of the amount of data created, consumed, and stored has been growing exponentially over the past two decades, with the total amount of data being just two zettabytes (trillion gigabytes) in 2010, 64.2 zettabytes in 2020, and 181 zettabytes by 2025 by some estimates [1]. In order to discover anything useful out of this vast array of information, efficient algorithms need to be created, especially ones that take into account datasets that are bigger than RAM, often by orders of magnitude. To process those massive datasets, external-memory algorithms must be used, which poses unique challenges since all of the data cannot be loaded into the computer's memory at the same time. In many cases, it's useful for these massive datasets to be sorted, such as in large-scale applications like Apple or Google Maps, which use graphs as one of their underlying data structures. Many databases also require some of their data to be sorted in some fashion to have efficient query operations, such as NoSQL structured protocols. Sorting data for these and other big applications is important, but it can be a large time sink for any software implementation and especially so if an external sort must be used. Another example of this is in Google's MapReduce, where it must "sort [a reduce worker thread] by the intermediate keys so that all occurrences of the same key are grouped together" [2]. While these types of sorts are common, they can still be quite slow. This is the subject of my research.

When sorting large datasets, the data can come from different files, parts of files, or streams, which poses a problem when trying to create truly high-performance algorithms since the underlying hardware can be slow, specifically in the I/O speed of storage devices. Disk HDDs are still extremely common in the modern day since they're able to store a large amount of data cheaply when compared to alternative methods such as SSDs [3], and it's likely that physical disk storage devices will continue to hold that advantage into the foreseeable future with the emergence of laser-assisted heat recording technology known as HAMR. This new technology allows for a greater density of data [4] to be stored on each disk platter by using a small laser on each head to heat a tiny spot on the disk which allows for single bits at a time to be changed in just nanoseconds. Considering that HDDs will continue to be widely used, it's worth discussing their limitations.

Since these hard drive devices utilize a platter and head, there is a large time cost when the disk's head needs to reach its next position on the platter, which is known as seeking. Seeking is the main reason why disk I/O is much slower than computer CPU and RAM rates, often to several orders of magnitude, especially when the platter or head must move for any read or write operation. On average, RAM runs at around 2-3k MB/s, and a typical hard drive can be expected to run at about 100-200 MB/s for sequential read/write operations. In the extreme case when switching between many very small files (4KB), that performance drops significantly to just around 1 MB/s for most hard drives, which is due to the physical constraint of the head and platters when seeking to a new location. As an aside, SSDs can run the same extreme case at up to 200 MB/s, but since that is an unusual case and the per-byte cost of SSDs are much higher than that for magnetic drives, HDDs continue to be the standard choice for bulk data storage. Many of the current solutions involving external memory today do not take this seek time into account in the application runtime, leading to less-than-ideal results, sometimes to a large degree. Some of these modern solutions including Spark [5], Hadoop [6], and Facebook Cassandra [7], all of which are extremely widely used.

Given my research aims to create a high-performance external-memory sorting algorithm, these I/O costs must be considered. In order to get the most speed possible, the number of seeks that the disk does must be minimized, and there are fortunately several ways to do this. As an example, consider the naïve solution for external sorting where the available memory (M) is divided equally among the number of files (N) such that each file has the same number of elements in RAM (M/N). As those elements are consumed across all sections, it follows that after M/N total values are taken out of memory, one of the sections becomes empty and must read more values from its corresponding file. It'll be shown later in the paper that by changing how many values each file has in memory at the start of the program, more values on average will be consumed before needing to access the disk again, which decreases the overall number of seeks. Another change that can be made is by changing how many values are refilled from each file, which will again be discussed in more detail later in the paper. Using other such similar techniques, a more optimal runtime can be achieved.

Another important part of my algorithm is the internal sorting portion. This is necessary since, when writing external sorts for data larger than RAM, the files cannot be directly merged since all of the values aren't known at one time. This means that a smaller value than those in RAM at the start could be stored deep into a file and would be read and considered too late. To solve this, each file is first divided into chunks the size of the available RAM, and those chunks are sorted individually. Then the Mergesort portion merges these sorted chunks together to fully sort the entire dataset. In-memory sorting algorithms are a common problem in computer science, and every solution is usually limited by the nature of the issue such that even the best algorithms can usually only get so fast when working with random data. A high-performance internal-memory sorting algorithm I will use is called Origami [8], which is being developed by Arif Armin, a PhD student under Dr. Loguinov. This solution utilizes special registers on the CPU and new extensions to the x86 instruction set called Advanced Vector Extensions (AVX). AVX utilizes sixteen registers, all of which can perform simultaneous operations on either 256 or 512 bits, depending on the type of AVX being used (AVX2 vs AVX-512). Since many more mathematical operations can be done on one clock cycle of the CPU, this greatly improves sorting speeds, and in fact a similar approach can be done to create a highly efficient merge sort as well using tree merge. Part of my benchmarking later in the paper will show Origami's speed compared to other approaches.

The data used throughout the paper was randomly generated using a linear congruential generator, which is a type of pseudo-random number generator which generates keys in a uniform distribution. With the correct parameters, this method can pass the formal tests for randomness. Other types of distributions were tested, but not as extensively as the LCG generation. It's also worth mentioning that the current solution does not use multithreading, which can offer even more speed by having one thread load data in while another thread consumes it in the Mergesort. The solution when utilizing Origami is also currently confined to files that are powers of 2.

While my solution will utilize Origami, it'll be created in such a way that other sorts can easily be used instead if desired. By the end of this paper, I hope to show significant progress towards a high-performance external-memory sorting algorithm that takes into account the number of seeks to create a faster solution than those currently used by popular solutions mentioned previously.

2. METHODS

2.1 General External-Memory MergeSort Structure

It's first worth discussing the overall strategy that the algorithm employs. The individual elements in this strategy will be examined more in-depth in their corresponding sections following this one.

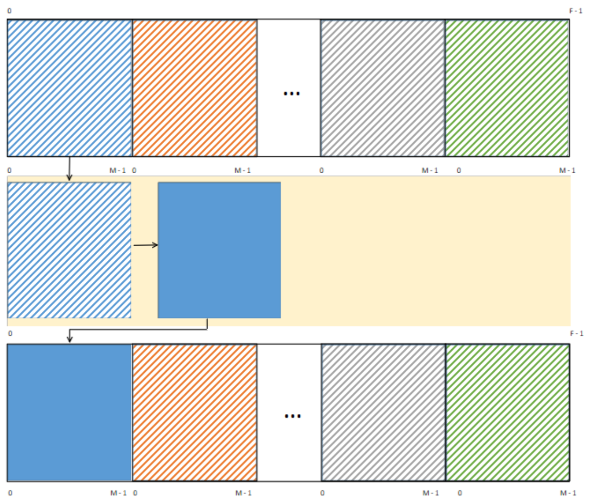

Figure 1: A visualization of the in-memory chunk sorting phase of the algorithm.

As the name implies, external-memory sorting must be used when the amount of data that needs to be sorted is greater than the amount of RAM available. As such, it cannot all be sorted in one pass for reasons explained prior. In this paper, the input is considered to be a single file stored on a HDD. Given that the unsorted data is already generated and is stored in the very large file, the next step is to divide the file into RAM-sized chunks. These chunks are individually loaded into memory, sorted, and written back into a separate file from the original. If the total file size is F bytes and the total memory size is M bytes, then the number of chunks sorted in this manner is $k = \frac{F}{M}$. This chunk-sorting yields a new file with k sorted chunks. This process is shown in figure

1.

The next phase is similar to the “merging k sorted arrays” problem, which is a common one in computer science. It works by loading just a portion of each chunk into memory at a time, sorting those values until one chunk’s portion runs out of values. Once a chunk’s portion, which we will call a stream, is empty, the next portion of values are read from the chunk into memory and the sorting operation continues. This lasts until all streams are empty and all chunks have been read completely, resulting in a fully sorted file of F bytes.

While the general strategy is simple, the overall speed of the naïve implementation of such an algorithm can be surprisingly slow. The following sections will discuss the approaches used in order to create a more efficient external-memory MergeSort.

2.2 Data Generation

The data used in this research was a series of pseudorandom numbers that were generated on each run of the algorithm. A unique set of pseudorandom numbers was created each time using a linear congruential generator (LCG), which were then written to a file. The specifics of the LCG algorithm are out of scope for this paper, but it was chosen due to its simplicity and speed, allowing for quicker tests of the overall algorithm while still providing good randomness and distribution. The parameters were chosen to maximize speed, and due its nature, it allowed for a very fast implementation by unrolling the loop which generates these numbers. For example, instead of generating one value per iteration of a for-loop, one can find much better performance by generating eight values per iteration while still keeping the code simple. Interestingly, the performance benefit stopped at the eight-level unroll; anything more than that would actually result in a decrease in the amount of values generated per second.

2.3 In-Memory Sort

Given a file of F bytes and a memory size of M bytes, the in-memory sort phase involves k number of chunks, where $k = \frac{F}{M}$. The first implementation utilized the standard C++ library sort function, which simply takes an array of unsorted numbers and sorts them in the same array.

The final implementation utilizes Arif Armin’s Origami [8] algorithm, which is an extremely high-performing sorting solution, including in-memory sorts such as these. It's able to achieve higher speeds by utilizing special CPU registers called Advanced Vector Extensions (AVX) and its predecessor, Streaming SIMD Extensions (SSE). While typical registers can only hold one value per CPU instruction execution, these registers allow for multiple values to be loaded into them at the same time, thus increasing the throughput by however much larger they are. For example, SSE registers can hold four 32-bit integers at a time. Therefore, when sorting an array of 32-bit integers, the CPU can theoretically sort it four times faster than it would otherwise be able to when using an SSE register.

While Origami does run much faster than the standard C++ library sort, it does have a disadvantage. Origami's in-memory sort is an out-of-place sort, which means that it requires another array of the same size as the input array. This new array is where the fully sorted sequence of numbers is written to. This means the amount of data it can sort per iteration is cut in half, and so the chunk sizes must also be cut in half to $M/2$ which also doubles the number of chunks, $2k$. Despite this limitation, the performance increase gained is still very much worth using Origami's internal sorting capabilities.

2.4 MergeSort

The last phase of the external-sort strategy is the merging of all the streams. The naïve implementation is to create a priority queue, where the minimum value is always at the start of the queue. First, the amount of memory that each chunk can give values to is defined. The intuitive solution is to give each chunk the same amount of memory such that, given memory size M, each chunk will have M/k bytes of memory it can load values into. This M/k block of memory per chunk will be referred to as a chunk's memory portion. The portions are loaded from their respective chunks, with the values at the start of each chunk being taken first since the individual chunks are sorted. It's also worth mentioning here that the file that contains all the chunk's values is stored on the hard drive. Then, the first value from each portion is read from its place in memory and put into the priority queue, where each value inserted is automatically sorted. Once all portions have a value in the priority queue, the queue is popped. The popped value is guaranteed to be the minimum of the values in the queue, which at the start, is also the minimum out of all values in the file. That value is stored in an array called the write buffer, and the portion where that value came from inserts its next number into the queue. This process continues until a portion runs out of values to give to the queue.

Once a portion empties, the algorithm checks to see if the chunk the portion belongs to is also empty. If it is not, then the next portion of values is read from the chunk and loaded into memory at the same location as the original portion, overwriting the old values. The old values can be overwritten since they are all correctly sorted in the write buffer (or already written to the in-progress fully sorted file, depending on if the write buffer was filled). Then, the algorithm continues on until the next portion runs out of values. If the chunk the portion belongs it is empty, then no more values are loaded in from that stream and the size of the priority queue decreases by one. The algorithm continues until all streams run out of values, which also means the priority queue empties. The end result is a file of size F bytes in fully sorted, ascending order.

This naïve implementation explained above can be improved in a couple of ways. The first is by utilizing Armin's Origami solution, which contains a fast merge function utilizing the same special registers in the in-memory sort function. Origami replaces the priority queue in the above implementation and instead uses a tree of many binary comparisons, where the leaves of the tree correspond to each stream's memory portion. The logic for refilling these portions remains the same.

The second optimization is less intuitive and involves the size of each stream's portion. The question we pursue is, if each portion was a different size, would there be a performance increase [9]? We define the first portion to be some size Δ bytes, the second to be 2Δ bytes, and so on until the last portion is $k\Delta$ bytes. Since the sum of all portions must be equal to M,

$$M = \sum_{i=1}^{k} \Delta i = \Delta \frac{k(k+1)}{2} \tag{1}$$

From the above equation, it follows that

$$k\Delta = \frac{2M}{k+1}. \tag{2}$$

It can be seen from Fig. 2 that, on average, the algorithm will go through $k\Delta$ values before a portion becomes empty. The former implementation, where every portion was the same size (M/k), it would go through M/k values on average before encountering an empty portion. Since $k\Delta = \frac{2M}{k+1} > M/k$ by about 2 (where M, $k > 1$), this new implementation sorts about twice as many values on average before needing to read new values from the disk, thus requiring less seeks overall. Since the disk is such a big bottleneck, needing fewer seeks grants a huge speed increase.

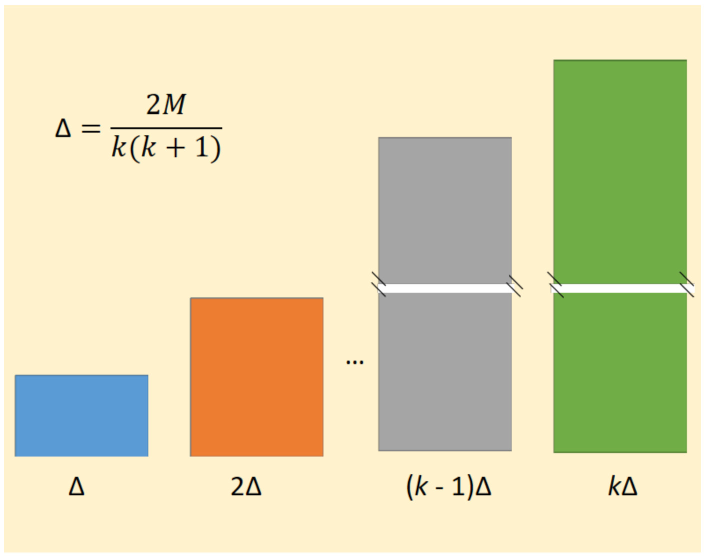

Figure 2: A visualization of the different size portion initialization in the MergeSort phase of the algorithm.

In this variable portion size solution, whenever a portion runs out of values, the portion then gains $k\Delta$ bytes. This can lead to a memory allocation issue as shown in Fig. 3. In the first refill, the first portion was initialized to a size of Δ bytes. If it was done in a continuous array, then the $k\Delta$ bytes refill would be split among all k portions, which becomes obviously untenable after even just a couple of portion refills. The solution to this involves a queue of previously allocated memory blocks of around 1 MB. Each portion contains their own queue, containing the minimum number of blocks necessary and deleting the blocks from the queue as they empty and appending them to a global free-block queue. Then, on refill, the empty portion can simply take memory blocks from the free queue and load its values into them.

It's worth mentioning that another optimization can be done where, instead of one bucket having Δ bytes, the next 2Δ bytes, and so on, the first $\sqrt{k}$ buckets (group 1) all contain Δ bytes each, the next $\sqrt{k}$ buckets (group 2) contain 2Δ bytes each, and so on. The initial sizes z for each group i and number of streams k and the size of each group c is given by equation 3 as described in the bowtie data streaming paper [9]:

$$z_i = \frac{2M}{k/c+1}(1 - \frac{ic}{k}) \tag{3}$$

Since z_i is the size of each group, the size that each bucket actually gets in memory is z_i/k. In cases where the $\sqrt{k}$ is not a whole number, it can be rounded down or up to achieve nearly the same optimized rates.

These optimizations yield a much faster solution than the naïve implementation, but there are several other improvements that can be made which are described in the next section.

2.5 Other Considerations

One major optimization has to do with how many bytes are written to or read from the disk at a time. It's generally known that it's more efficient to have a write buffer of around 1 MB since the disk can write that amount of bytes relatively fast, and it also allows for an optimal amount of unsorted values to be loaded at a time. However, it's also more efficient for read operations to also be limited to 1 MB. Even when gigabytes of continuous data needs to be read from the disk for the in-memory sort phase, it's better to call the read API many times for 1 MB at a time rather than issuing one read API call for all the data needed. It's unclear as to why this is the case for the read, since intuitively it should be more efficient to read that data all at once. Nevertheless, splitting up the read operation similar to the write yields speeds up to the speed specifications of the hard drive, which is necessary for a high efficiency algorithm.

Another useful optimization is related to where the file is physically stored on the hard drive. File system abstractions can make it difficult to ensure that large files are actually stored continuously in the disk. Continuous storage is desirable due to the nature of how hard drives function. Hard drives contain platters magnetic platters where the actual bits are stored and heads which seek along the platters. The heads can move along a line on the platter and the platters can spin, which allows for the entire surface area of the platter to be covered. The slowest part of a hard drive disk is the platter spinning, so it's better for the platter to spin as few times as possible. When files are read or written sequentially, the platter does not need to spin as much, and most of the burden of the work falls on the head. The speed of moving the head is much faster than that of the platter, and so sequential read/write operations become much faster than random read/write operations since they require the platter to spin much more frequently.

Using Evan Krohn's program, a large file was created on the hard drive in the test environment. This file was also created in the front of the drive since the hard drive works faster on areas in the front than those elsewhere. It also ensures that the disk will be performing sequential read and write operations, which is much faster than random access read and write. A significant performance increase can be gained by utilizing such a file, where it's divided into thirds: the first third contains the unsorted data, the second contains the chunk-sorted data, and the third contains the fully sorted, merged data.

Multithreading can yield yet even more performance bonuses. The idea is relatively simple. On a single thread, whenever the program read/writes the disk, it must wait for that read/write operation to complete before continuing. Since the disk is so much slower than the CPU, a significant amount of time is wasted by the CPU idling. If the program was allowed to continue sorting data while the read/write operation was executing at the same time, then a performance increase could be seen. In the MergeSort case, this can be accomplished by having two write buffers. Once one buffer is filled, one thread executes a write command to the disk for the full buffer. The other thread continues to sort the values in memory, writing the results to the second write buffer. Once the second write buffer is filled, it checks that the first write operation completed, then it issues another write command to the disk while the other thread starts to refill the other buffer that just finished writing. By allowing the CPU and disk to work in parallel like this, overall throughput, and thus performance, can increase. However, it's possible that this method could also result in a decrease in performance overall if utilized in the internal-memory chunk sorting stage.

When utilizing the multithreading described above, two buffers must be created instead of just one in the naïve case. This means that, when used to sort each individual memory-sized chunk, the available memory must be split in two, which decreases the size of every chunk by a half. This doesn't result in a tangible performance difference in this stage, but it does decrease the performance of the merge stage. The speed of the merging stage depends on the number of chunks, and there is a substantial performance drop even when the chunks are doubled just one time to the point where applying the multithreaded writing strategy in the chunk sorting stage isn't worth the cost.

3. RESULTS

The test machine runs on a 12-core Intel i7-4930K @ 3.40GHz with 32 GB of DDR3 RAM and a disk system connected by two Areca 1880 PCIe 2.0 RAID controllers. The overall volume of the file system is 40 TB with a 2.8 GB/s sequential read/write speed. The tested CPU contains SSE registers, which can only hold up to 128 bits of data at one time.

First, it's worth mentioning the slow speeds of Hadoop [6] and Spark [5], which are popular big-data frameworks used for tasks such as sorting and MapReduce. As outlined in the bowtie data streaming paper, these existing solutions are so slow when executing large sorts that it's impossible to create comparable benchmarks, but some idea of their inefficiency can be understood. Hadoop's sort was ran on a 100 GB file with 25 GB of RAM and resulted in 415 million attempted seeks and a total time of 6.6 hours [9]. Spark ran the same sort with only 10 GB of RAM and resulted in 34 million seeks and a total time of 10 hours [9]. It's clear from these few measurements that both Spark and Hadoop, despite their popularity, clearly lack the high-performance sort speed that's needed, even when ran on significantly more powerful systems than the solution outlined in this thesis.

The benchmarks shown in tables 1 and 2 were generated using different versions of the MergeSort algorithm. Each version utilized the continuous large file at the start of the drive and had the optimal read/write buffer sizes at 1 MB. Multithreading the writing was not implemented for these results for reasons discussed earlier, and each run utilized 16 GB of RAM. The first two runs utilized the ideal memory portion size for each chunk as shown in equation 2, with the first utilizing a MinHeap as the chunk merging logic and the second utilizing Origami's merge. As expected, those two implementations resulted in the same number of disk seeks, but as can be seen in table 2, the Origami merger outperforms the MinHeap by up to $4.4\times$. It's likely that for even larger file sizes, an even greater performance difference can be seen.

Table 1: Number of Seeks

File Size (GB)	MinHeap	Origami Single Refill	Origami Multi Refill
32	10	10	10
64	37	37	28
128	104	104	94
256	367	367	318
512	1406	1406	1210
1024	5518	5518	4474
2048	21712	21712	17650

Table 2: Merge Rate (MB/s)

File Size (GB)	MinHeap	Origami Single Refill	Origami Multi Refill
32	247	991	818
64	204	727	646
128	177	599	553
256	157	497	479
512	140	403	420
1024	127	335	362
2048	65	286	315

The last column shows results for an implementation of a multi bucket refill as shown in equation 3. As expected, the number of seeks is lower than the previous two tests, but the merge rate is lower than the Origami single refill method for files less than 512 GB. The difference between the number of seeks for files of that size is relatively small compared to the overhead required to implement the multi bucket refill strategy which explains the performance decrease. For files 512 GB and larger, however, the number of seeks drastically increases for both methods, meaning that the performance increase gained from the multi bucket refill then justifies the implementation cost, leading to an overall increase in the merge rate for larger files.

Tables 3 and 4 show the benchmarks for the best version of the discussed MergeSort compared against STXXL [10], nsort [11], and Tuxedo [9]. The benchmarks for the latter three methods are again taken from the bowtie paper and were gathered using a much more powerful system than the results from the MergeSort. Despite this limitation, the MergeSort implementation outperformed the STXXL and nsort solutions in every case by up to $6.5\times$, as shown when comparing the merge rates between them all. The number of seeks that the implemented MergeSort algorithm performed was also orders of magnitude lower than STXXL and nsort in all runs except for the 1 TB file with 2 GB of memory. In that case, MergeSort only had half the number of seeks that STXXL did while Tuxedo was still orders of magnitude more efficient. This performance difference can also be seen in the merge rate, where MergeSort performed worse in that case when compared to the others. It also performed worse than Tuxedo in the other cases, but this performance difference could be because the system MergeSort was tested on was much weaker than that used for the Tuxedo benchmarks. More testing on the same system is needed to know for sure.

Table 3: Number of Seeks

M (GB)	File Size (GB)	STXXL	nsort	Tuxedo	MergeSort
1	8	4,165	11,232	63	94
2	128	66,126	139,772	2,675	4,474
2	1,024	575,718	2,181,102	18,263	276,226
8	512	262,679	598,086	2,648	4,474

Table 4: Merge Rate (MB/s)

M (GB)	File Size (GB)	STXXL	nsort	Tuxedo	MergeSort
1	8	57	56	561	374
2	128	56	69	554	249
2	1,024	51	50	434	87
8	512	56	55	650	334

One big advantage this MergeSort implementation has is its durability against different data distributions. Table 5 shows different distributions and how the correctness and speed of the discussed MergeSort is impervious to them. In each case, 32 GB were sorted using 16 GB of RAM. The first two cases are the LCG (Linear Congruential Generator) and Mersenne-Twister, both of which produce a uniform distribution of pseudo-random numbers. The last three cases show both the internal sort rate and the merge sort rate for each other distribution tested, namely the fibonacci, pareto [12], and zipf [13] distributions, all of which are non-uniform. In a distribution sort method, these non-uniform cases would result in large slow downs and, in many cases, wouldn't output a correctly sorted file. Both of these issues are avoided by using a merge sort method. As shown in table 5, the MergeSort is not sensitive to non-uniform distributions. In all cases, the internal memory chunk sort phase ran at around 750 MB/s and the merge sort phase ran at around 815 MB/s. The number of seeks in all cases was the same since the file size, memory size, and portion sizes were all the same for every run, so they were not included.

Table 5: Merge Rate (MB/s)

Distribution	**Internal Sort**	**Merge Sort**
LCG	735	808
Mersenne-Twister	761	825
Fibonacci	753	810
Pareto	752	814
Zipf	751	819

4. CONCLUSION

Creating a high-performance external-memory merge sort algorithm can be challenging. However, the time and computing power saved by the implementation proposed here is well worth the effort. Virtually every commonly used application requires some amount of sorted data, and much time is wasted by current popular solutions such as Hadoop and Spark. Several different strategies were implemented and tested to prove the most optimal version, including a MinHeap merge strategy and how different chunk portion sizes can help minimize the number of seeks performed by the HDD. Even somewhat faster algorithms such as STXXL and nsort are magnitudes slower than the final MergeSort solution, even when tested on slower hardware than those done for the former.

The results gathered show much promise in this implementation, but some more progress can be made. The algorithm tops off at around 800 MB/s in the MergeSort phase when sorting just four chunks, but the hard drive where these benchmarks were taken runs at a speed of 2.8 GB/s. Since it's still desirable for the bottleneck to only be on the external I/O, a true high-performance algorithm should run at the speed of the disk. The main reason for the lack of speed on the system tested is the type of registers available. That CPU only includes SSE registers, which hold 128 bits or four 32-bit integers, but AVX2 registers can handle 256 bits or eight 32-bit integers and AVX512 holds 512 bits or sixteen 32-bit integers. Since AVX512 registers can hold four times the number of bits as SSE registers, it follows that a performance increase of up to $4\times$ can be acquired on a system which includes AVX512, assuming that there are no other bottlenecks. This would result in 3.2 GB/s, which is greater than the speed of the disk on the tested system and would result in the bottleneck being the disk I/O as desired. However, there does exist even faster HDDs; some can read and write up to 4 GB/s, so further improvements need to be made to ensure the bottleneck lies on the disk.

One important improvement that can be done is implementing a multipass merge. In cases where there are a lot of chunks to merge together, such as over 1000, it can be inefficient to merge them all at one time since the memory is split into so many small pieces. The solution to this problem involves taking the square root of the number of chunks, defined as n, and doing n separate MergeSort runs then merging all n outputs together into one final file. This would decrease the overall number of seeks done when compared to the single pass, which would improve the speed in turn. One case where this optimization would result in a large performance increase is in the 1 TB file with 2 GB of memory case as outlined in tables 4 and 5. The current implementation showed a huge slowdown in this test case due to the number of independent chunks that had to be merged together in the merge phase. A multipass merge would drastically improve the speed of this to be more comparable to the results for the other cases tested.

One other potential improvement involves the size of the memory blocks used. In this implementation, the portion of memory given to each chunk at any one time is divided into an array of 1 MB blocks of memory. Internally, once the first block has all its values read, a function is called by Origami's merge algorithm to either advance the array to the next memory block or to refill the array with a new sequence of filled memory blocks with new values read from the file chunk. By increasing the size of the memory block, it's possible that a performance increase could be seen since that function would not need to be called as often. However, the RAM would also end up being more fragmented and could result in memory space issues if the memory blocks were too large.

Although not every optimization was included in this implementation, such as the multipass merge, these initial tests show that the strategies discussed in the bowtie data streaming [9] do result in large performance gains. In addition, since this implementation is a merge sort instead of distribution sort, it's unimpeded by non-uniform data distributions, giving it a large advantage.

REFERENCES

[1] A. Woodie, “Big Growth Forecasted for Big Data.” https://www.datanami.com/2022/01/11/big-growth-forecasted-for-big-data/.

[2] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, pp. 107–113, 2008.

[3] R. Bauer, “HDD vs SSD: What Does the Future for Storage Hold?.” https://www.backblaze.com/blog/ssd-vs-hdd-future-of-storage/.

[4] J. Hruska, “Seagate Wants to HAMR the Competition, Ship 100TB HDDs By 2025.” https://www.extremetech.com/computing/280190-seagate-wants-to-hamr-the-competition-ship-100tb-hdds-by-2025.

[5] “Apache Spark.” [Online]. Available: http://spark.apache.org/.

[6] “Apache Hadoop.” [Online]. Available: http://hadoop.apache.org/.

[7] P. M. Avinash Lakshman, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, April 2010.

[8] D. L. Arif Armin, “Origami: a high-performance mergesort framework,” *Proceedings of the VLDB Endowment*, pp. 259–271, October 2021.

[9] G. Stella and D. Loguinov, “On High-Latency Bowtie Data Streaming,” *IEEE BigData*, December 2022.

[10] R. Dementiev, L. Kettner, and P. Sanders, “STXXL: Standard Template Library for XXL Data Sets,” *Software: Practice and Experience*, vol. 38, pp. 589–637, May 2008.

[11] “Nsort.” [Online]. Available: http://www.ordinal.com/.

[12] “Pareto.” [Online]. Available: https://www.sciencedirect.com/topics/computer-science/pareto-distribution.

[13] "Zipf." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/zipf-distribution.