Université libre de Bruxelles

Database Systems Architecture

Winter Semester 2017-2018

Multiway Merge Sort Implementation

Authors:
Syeda Noor Zehra Naqvi
(000455274)
Sofia Yfantidou
(000456361)

Supervisor:
Prof. Stijn
Vansummeren

January 6, 2018

Contents

1 Introduction and Environment

The goal of this project is to experiment on various input and output methods for reading and writing from and to file in a programming language of our choice, along with exploring, implementing and benchmarking the Multiway Merge Sort algorithm for externally sorting large files that do not fit in memory (a realistic scenario when it comes to the world of Big Data). We chose Java for this project and no external libraries were used. Finally for the benchmarking parts of this project the benchmarking data were generated randomly using Java's *Random* class. 2 to 30 files of 0.5M to 200M 4-byte integers were created. The largest amount of data we benchmarked with was 15 files of 200M integers each or in other words 12GB of data in total per approach. We ensured that when benchmarking with Merge Sort the same data file was used as input to produce meaningful results.

1.1 Hardware Used for IO Streams

The benchmarking for I/O streams was executed using a Dell Inspiron 7559 laptop with the following specifications:

- **OS Name:** Microsoft Windows 10 Home
- **System Type:** x64-based PC
- **Processor:** Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz, 2301 Mhz, 4 Core(s), 4 Logical Processor(s)
- **Installed Physical Memory (RAM):** 16,0 GB
- **L2 Cache Size:** 1024 KB
- **L3 Cache Size:** 6144 KB

1.2 Hardware Used for Multiway Merge Sort

We used ASUS K501U laptop for executing multiway merge sort with different parameters. Hardware specifications of this device are as follows:

- **OS Name:** Microsoft Windows 10 Home
- **System Type:** x64-based PC

- **Processor:** Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 2.59 GHz
- **Installed Physical Memory (RAM):** 16,0 GB
- **L2 Cache Size:** 512 KB
- **L3 Cache Size:** 4096 KB

2 Observations on Streams

As requested 4 different stream approaches were utilized. The **system calls**, which read and write one element at a time, the **buffered streams**, which implement their own buffering mechanism and read and write multiple elements at a time, the **parametrized buffered streams**, which use a buffer of user-defined size and read and write elements equal to this size and finally the **memory mapping**, which maps and unmaps a B-sized portion of the file to memory. Note that we experimented also with a fifth approach, **object streams**, which read and write one Java object at a time.

2.1 Expected Behavior

This section includes details on the implementation of each stream method. It also describes their expected behavior for varying parameters and provides cost formulas for the total number of I/Os per method. Note that in every formula we multiply by k, because we consider operations happening in multiple (k) files. The formulas that refer to a single file are the same without the multiplication by k.

Two classes were created, **InStream** for input streams and **OutStream** for output streams. The InStream class includes methods for opening streams for reading, reading the next element from stream and identifying the end of stream. The OutStream class includes methods for creating streams for writing, writing elements to stream and closing streams. The Java code is well-commented for each method.

2.1.1 System Calls

This approach reads and writes a single primitive data type at a time from and to file. In Java this behavior is accomplished using the *DataInputStream*[1] and the *DataOutputStream*[2] classes along with the *readInt* and *writeInt* methods.

An advantage of system calls is that they can read and write primitive Java data types from and to streams in a machine-independent way. They can also produce human-readable files.

However, system calls are expected to lack in velocity as they read one element at a time and I/Os are an expensive operation as discussed in class. The difference is expected to be even more noticeable as the file size grows. Amongst the suggested methods we expect system calls to be the worst performant. Moreover, system calls are not thread-safe according to documentation and concurrency control should be taken care of by the user.

The formula that gives the expected cost in I/Os for reading or writing N elements from or to k files is given below.

$$I/Os = k * N$$

2.1.2 Default Buffered Streams

This approach reads and writes 8192 bytes[3], or 2048 4-byte integers, at a time from and to file. To achieve this it uses an intermediate buffer of the same size. In Java this behavior is accomplished using the *BufferedInputStream*[4] and the *BufferedOutputStream*[5] classes along with the *readInt* and *writeInt* methods.

Compared to the system calls the default buffered streams are expected to be more performant. Their advantage is that they read and write 8092

[1]DataInputStream (Java Platform SE 7). 2017. Retrieved from: `https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html`.

[2]DataOutputStream (Java Platform SE 7). 2017. Retrieved from: `https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html`.

[3]GC: BufferedOutputStream - java.io.BufferedOutputStream. 2018. Retrieved from: `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/io/BufferedOutputStream.java#BufferedOutputStream`.

[4]BufferedInputStream (Java Platform SE 7). 2017. Retrieved from: `https://docs.oracle.com/javase/7/docs/api/java/io/BufferedInputStream.html`.

[5]BufferedOutputStream (Java Platform SE 7). 2017. Retrieved from: `https://docs.oracle.com/javase/7/docs/api/java/io/BufferedOutputStream.html`.

bytes at a time compared to 4 bytes of system calls. In other words, bytes can be written or read (ahead of time) to the underlying output or input stream without necessarily causing a call to the underlying system method for each byte written or read. This means less I/Os and thus less time.

A disadvantage of this approach is that the default buffer size may not be appropriate for all kinds of applications and hardware. The "ideal" buffer size is dependent on the file system block size. If the buffer size is configured to be a few bytes more than the disk block size, then the operations become inefficient, as a single buffer read would lead to two block reads by the file system for just a few extra bytes. That's why the buffer size should be equal or a multiple of the file system block size.

The formula that gives the expected cost in I/Os for reading or writing N elements from or to k files is given below.

$$I/Os = k * \left\lceil \frac{N \quad integers}{\left\lfloor \frac{8192 \quad bytes/block}{4 \quad bytes/integer} \right\rfloor} \right\rceil = \left\lceil \frac{N \quad integers}{2048 \quad integers/block} \right\rceil$$

2.1.3 Parametrized Buffered Streams

This approach is identical to the aforementioned one, but a user-defined buffer size is used instead of the default size of 8192 bytes. Thus, the implementation is identical except for the constructor called for *BufferedInputStream* and *BufferedOutputStream*, where a second parameter for the buffer size is used.

Similarly to the default buffered streams approach, the parametrized buffered streams are expected to perform better than the system calls for the same reason mentioned above. Also, the advantage of this approach over the default one is that the buffer can be tailored to the hardware used to run the application. Moreover, we expect all buffered approaches to perform better with larger buffer sizes.

However, this can easily become a disadvantage if the buffer size is not chosen wisely. Moreover, both the default and the parametrized buffered streams utilize blocking I/Os. This means that for a single thread, when it invokes a read or write method, that thread is blocked until there is some data to read, or the data is fully written. This being said the buffered approached are expected to be less performant with multiple open channels for a single thread.

The formula that gives the expected cost in I/Os for reading or writing N elements from or to k files into a buffer of size B is given below.

$$I/Os = k * \left\lceil \frac{N \quad integers}{\left\lfloor \frac{B \quad bytes/block}{4 \quad bytes/integer} \right\rfloor} \right\rceil$$

2.1.4 Object Streams

This approach is an alternative of the parametrized buffered streams approach. It takes advantage of the fact that arrays (our buffer is modeled as an array) are Java objects and thus natively serializable. So, every time the buffer is full it is read or written from or to file as an object. Note that for object streams it is only possible to identify the end of file on a block-basis. In other words, the last block in the file can be identified, but the last element cannot. That's why a try-catch block (EOFException) is utilized in this approach to identify the last element.

The advantage of this approach is that the buffer size is expected to not depend on of the file system block size as the buffer object is read or written straight from or to file, thus no extra RAM or cache latency is imposed and object streams are expected to be more performant than buffered streams.

However, serialization and object streams come with a lot of disadvantages. Class changes between Java or application versions can easily make the serialized data files unreadable. Also, reading object files from non-Java code is almost impossible. Finally, the files are not human-readable as they include objects rather than elements.

Note here that we are using buffered streams to enhance system calls of simple input and output streams, however using buffer streams we can even further enhance object streams.

The formula that gives the expected cost in I/Os for reading or writing N elements from or to k files into a buffer of size B is given below.

$$I/Os = k * \left\lceil \frac{N \quad integers}{\left\lfloor \frac{B \quad bytes/object}{4 \quad bytes/integer} \right\rfloor} \right\rceil$$

2.1.5 Memory Mapping

For Java IO package (all aforementioned approaches are included in it), user processes issue system calls (read and write) to transfer data between the

file system pages in kernel space (disk) and a memory area in user space (memory). There might or might not be a buffer in between depending on the approach. However, for the Java NIO package and memory-mapped files things work differently.

A memory-mapped file is a segment of virtual memory which has been assigned a direct correlation with some portion of a file that is physically present on-disk. In other words, memory mapping uses the file system to establish a virtual memory mapping from user space directly to the file. Once present, this correlation between the file on disk and the memory space permits applications to treat the mapped portion of the file as if it were in RAM[6], modifying the file by reading and writing directly to the memory.

In our implementation we get a channel (*FileChannel*) on a random access file (*RandomAccessFile*) and then we establish the virtual memory mapping by using the *FileChannel.map()* method. The map mode is *READ_ONLY* for input and *READ_WRITE* for output. Note that for closing the connection we need to close both the channel and the random access file file. However, closing the channel does not destroy the mapping. For it to be done, the Java garbage collector should collect the *MappedByteBuffer* object. Now, in Windows even an explicit invocation of *System.gc()* does not destroy the mapping and thus the file deletion throws an *java.nio.file.FileSystemException* (try-catch block, known Windows bug). However, in Linux, the deletion is expected to work properly.

The main advantage of memory mapping is that according to documentation it is the fastest I/O method in Java (at least for larger files), which makes sense as the I/O operations are handled by the operation system and not the application itself. In other words, the application sees the file data as memory, so there is no need to invoke read and write system calls, which is expected to lead in faster performance. Moreover, Java NIO uses non-blocking streams (in contrast with buffered streams), which means that threads are nor blocked during read and write operations and can perform I/O operation on other channels meantime. That is, a single thread can now manage multiple channels of input and output, thus memory mapping is expected to work better with multiple open streams.

A disadvantage of the NIO approach is that while in IO design (buffered

[6]Java NIO - Memory-Mapped Files with MappedByteBuffer - HowToDoInJava. 2015. Retrieved from: `https://howtodoinjava.com/java-7/nio/java-nio-2-0-memory-mapped-files-mappedbytebuffer-tutorial/`.

streams, system calls) you read the data byte by byte from an InputStream e.g. *readLine() or readInt()*, in NIO you read a specified amount of bytes from the mapped buffer. This means that when the read method returns it is not known if all the needed data is inside the buffer. All that is known is that the buffer contains some bytes[7]. However, in our case we were not affected that much by this issue.

In this case the operating system handles the I/O operations and thus no formula can be given on the application side.

2.2 Experimental Observations

Experiments were done on different values of N, B and k keeping 2 of these parameters constant while changing the value of the third one. 3 executions were performed for each approach and averages were considered for the results. **The different I/O approaches are numbered in the tables. System Calls as 1, Default Buffered Streams as 2, Object Streams as 3a, Parametrized Buffered Streams as 3b and Memory Mapping as 4.**

2.2.1 Changing N Values

For the first experiment we kept B and k steady and changed N. **N** was given **values between 0.5M (2MB file) and 200M (800MB file) integers**.

First, we kept k equal to 15 streams and the percentage of B steady at 10% of N, so for instance if N=1M then B=100K (0.4MB) integers. The results can be seen in Table 1. As can be seen in the table, working with system calls was too time consuming for more than 5M integers (only 20MB per file!). For a file of $N = 5M$, the systems calls required 3000x more time than the memory mapping. The same applies for working with the buffered streams (both default and parametrized) for more than 10M integers (only 40MB per file!). For $N = 10M$ the default buffered streams required 20x more time than the memory mapping, while the parametrized buffered streams required more than 6x more time than the memory mapping. As can be seen in Figures 2 and 3 this is mainly due to discrepancies in read and write performance. That's why we only experimented on object streams and memory mapping for larger N, whose performance was almost the same with

[7]Java NIO vs. IO. 2014. Retrieved from: `http://tutorials.jenkov.com/java-nio/nio-vs-io.html`.

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write	Read	Total
N/A	N/A	0.5M	15	1	105560	103323	208883	108095	107659	215754	105465	111035	216500	106373	107339	213712
N/A	N/A	0.5M	15	2	819	23057	23876	531	25724	26255	423	25365	25788	591	24715	25306
10%	50K	0.5M	15	3a	315	30	345	279	38	317	195	42	237	263	37	300
10%	50K	0.5M	15	3b	612	25495	26107	373	24451	24824	346	24426	24772	444	24791	25234
10%	50K	0.5M	15	4	131	77	208	74	93	167	41	101	142	82	90	172
N/A	N/A	5M	15	1	1093930	1083010	2176940	1054831	1104210	2159041	1120430	1023547	2143977	1089730	1070256	2159986
N/A	N/A	5M	15	2	13550	241017	254567	10228	271802	282030	14443	263995	278438	12740	258938	271678
10%	500K	5M	15	3a	246	28	274	536	140	676	375	162	537	386	110	496
10%	500K	5M	15	3b	3892	244855	248747	3535	271980	275515	2709	255627	258336	3379	257487	260866
10%	500K	5M	15	4	704	177	881	348	331	679	423	250	673	492	253	744
N/A	N/A	10M	15	2	27773	466493	494266	21442	568099	589541	32154	620515	652669	27123	551702	578825
10%	1M	10M	15	3a	749	309	1058	674	298	972	590	286	876	671	298	969
10%	1M	10M	15	3b	8365	473348	481713	6493	590919	597412	4611	483026	487637	6490	515764	522254
10%	1M	10M	15	4	575	218	793	1016	527	1543	530	469	999	707	405	1112
10%	5M	50M	15	3a	2518	1691	4209	2816	1270	4086	2889	1222	4111	2741	1394	4135
10%	5M	50M	15	4	3361	420	3781	2107	1917	4024	2152	1897	4049	2540	1411	3951
10%	10M	100M	15	3a	6180	2100	8280	5573	2504	8077	6760	2552	9312	6171	2385	8556
10%	10M	100M	15	4	5077	1276	6353	5224	5974	11198	4303	5905	10208	4868	4385	9253
10%	20M	200M	15	3a	11465	5801	17266	91423	8496	99919	9159	4645	13804	37349	6314	43663
10%	20M	200M	15	4	15341	4839	20180	44367	3718	48085	15695	29143	44838	25134	12567	37701

Figure 1: I/O Performance with Varying N, k=15 and B=10% of N

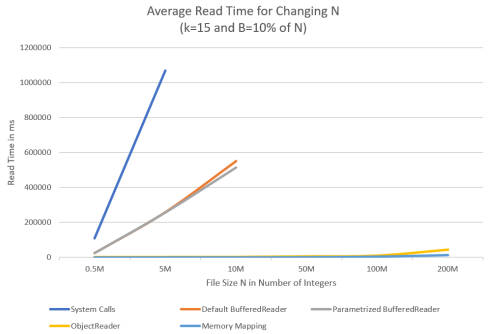

Figure 2: Average Read Time for Changing N (k=15 and B=10% of N)

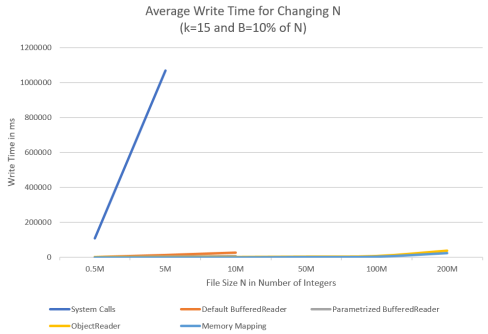

Figure 3: Average Write Time for Changing N (k=15 and B=10% of N)

memory mapping slightly outperforming object streams. A visualization of this comparison can be seen in Figure 4.

However, 15 files of 800MB would require 80MB of memory each for I/O operations or 1.2GB of memory in total with the given percentage of B (10%), which is not totally realistic. That's why we decided to experiment with changing file size N, using a realistic steady B size of 2000 integers (8KB) as can be seen in Table 5. In Figures 6, 7 and 8 it can be seen that the more the file size grows the time increases polynomially. Although memory mapping still outperforms object streams, the difference is not so defining. It is worth mentioning that for both these approaches there are no discrepancies between reading and writing times as noticed in the buffered

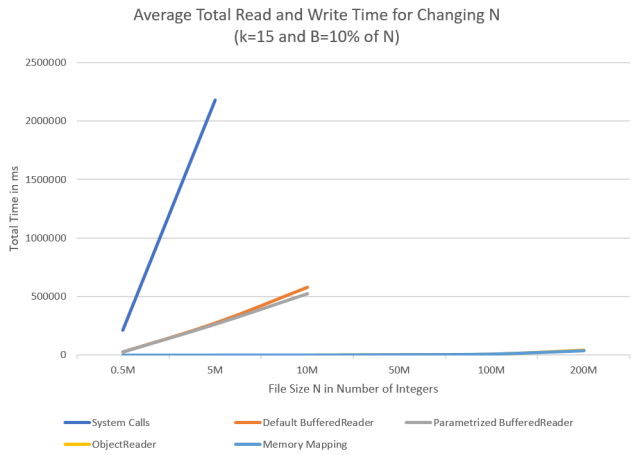

Figure 4: Average Total Read and Write Time for Changing N (k=15 and B=10% of N)

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write	Read	Total
	2K	5M	15	3a	572	388	960	344	313	657	350	316	666	422	339	761
	2K	5M	15	4	354	269	623	381	251	632	410	270	680	382	263	645
	2K	10M	15	3a	664	575	1239	559	581	1140	702	556	1258	642	571	1212
	2K	10M	15	4	512	482	994	461	453	914	469	451	920	481	462	943
	2K	50M	15	3a	2193	2205	4398	2101	1987	4188	2427	2063	4490	2240	2085	4359
	2K	50M	15	4	2342	2106	4448	1965	1901	3866	2003	2013	4016	2103	2007	4110
	2K	100M	15	3a	4460	4246	8706	4351	4121	8472	4331	4201	8532	4381	4189	8570
	2K	100M	15	4	5125	3661	8786	3835	3602	7437	4125	3569	7694	4362	3611	7972
	2K	200M	15	3a	8538	8397	16935	8980	8428	17408	12775	8664	21439	10098	8496	18594
	2K	200M	15	4	7926	7741	15667	8348	9672	18020	11612	8875	20487	9295	8763	18058

Figure 5: I/O Performance with Varying N, k=15 and B=2000

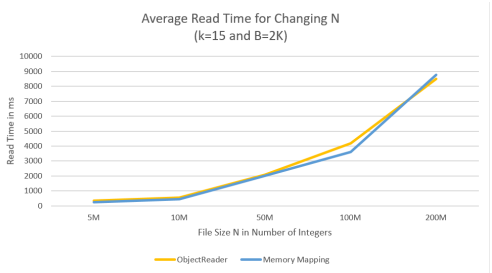

Figure 6: Average Read Time for Changing N (k=15 and B=2000)

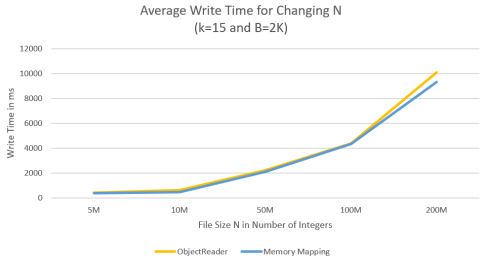

Figure 7: Average Write Time for Changing N (k=15 and B=2000)

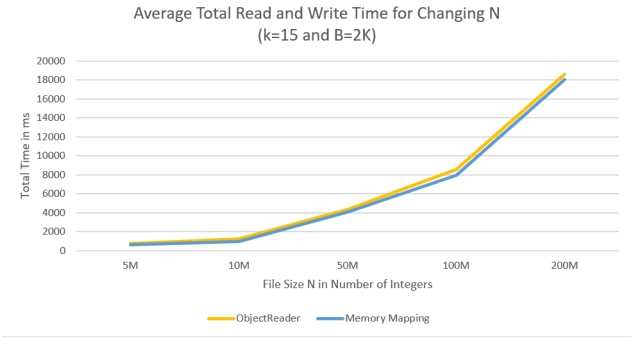

Figure 8: Average Total Read and Write Time for Changing N (k=15 and B=2000)

streams.

Overall, for changing N the most performant approaches identified were the object streams and the memory mapping.

2.2.2 Changing B Values

For the second experiment we kept N and k steady and changed B. **B** was given **values between 2% and 20% of the file (really large)**. We experimented only with parametrized buffered streams, object streams and memory mapping as B does not play a role in the other approaches.

First, we experimented with $N = 5M$ integers (20MB file), so that we can include the parametrized buffered streams approach in the experimentation (See Table 9). As can be see in Figure 10 the parametrized buffered streams approach is way slower than the other two (up to 400x slower), thus it was

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write	Read	Total
N/A	N/A	5M	15	1												
N/A	N/A	5M	15	2												
5%	250K	5M	15	3a	314	100	414	317	98	415	306	97	403	312	98	411
5%	250K	5M	15	3b	8851	239775	248626	9284	244895	254179	9562	240398	249960	9232	241689	250922
5%	250K	5M	15	4	414	290	704	338	259	597	306	281	587	353	277	629
N/A	N/A	5M	15	1												
N/A	N/A	5M	15	2												
10%	500K	5M	15	3a	377	160	537	536	140	676	375	162	537	429	154	583
10%	500K	5M	15	3b	3892	244855	248747	3535	271980	275515	2709	255627	258336	3379	257487	260866
10%	500K	5M	15	4	435	253	688	348	331	679	423	250	673	402	278	680
N/A	N/A	5M	15	1												
N/A	N/A	5M	15	2												
12%	600K	5M	15	3a	431	223	654	435	234	669	422	239	661	429	232	661
12%	600K	5M	15	3b	9390	239253	248643	9264	239296	248560	9301	239549	248850	9318	239366	248684
12%	600K	5M	15	4	303	284	587	315	293	608	322	283	605	313	287	600
N/A	N/A	5M	15	1												
N/A	N/A	5M	15	2												
15%	750K	5M	15	3a	495	348	843	517	312	829	460	279	739	491	313	804
15%	750K	5M	15	3b	8906	238766	247672	9660	236909	246569	9092	238302	247394	9219	237992	247212
15%	750K	5M	15	4	332	245	577	272	334	606	378	271	649	327	283	611
N/A	N/A	5M	15	1												
N/A	N/A	5M	15	2												
20%	1M	5M	15	3a	533	391	924	562	373	935	598	415	1013	564	393	957
20%	1M	5M	15	3b	8881	238004	246885	9356	242140	251496	9190	239268	248458	9142	239804	248946
20%	1M	5M	15	4	340	305	645	330	274	604	322	269	591	331	283	613

Figure 9: I/O Performance with Varying B, k=15 and N=5M

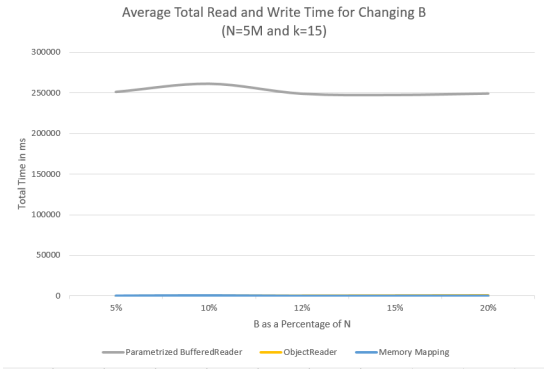

Figure 10: Average Total Read and Write Time for Changing B (k=15 and N=5M)

excluded for the next experiments.

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write	Read	Total
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
2%	1M	50M	15	3a	1303	286	1589	1578	277	1855	1294	284	1578	1392	282	1674
5%	2.5M	50M	15	3b												
2%	1M	50M	15	4	2171	1777	3928	2117	1819	3936	2152	1896	4048	2147	1831	3971
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
5%	2.5M	50M	15	3a	1846	628	2474	1840	1840	2476	1883	659	2542	1856	1042	2497
5%	2.5M	50M	15	3b												
5%	2.5M	50M	15	4	2757	1837	4594	2174	1854	4028	2141	1851	3992	2357	1847	4205
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
10%	5M	50M	15	3a	2848	1524	4372	2553	1519	4072	2837	2837	4127	2746	1960	4190
10%	5M	50M	15	3b												
10%	5M	50M	15	4	2228	1900	4128	2217	2017	4234	2280	1943	4223	2242	1953	4195
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
12%	6M	50M	15	3a	3261	1587	4848	3038	1743	4781	2956	1868	4824	3085	1733	4818
12%	6M	50M	15	3b												
12%	6M	50M	15	4	2101	1867	3968	2061	1845	3906	2110	1870	3980	2091	1861	3951
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
15%	7.5M	50M	15	3a	3816	1925	5741	3217	1977	5194	3393	3074	6467	3475	2325	5801
15%	7.5M	50M	15	3b												
15%	7.5M	50M	15	4	1935	1809	3744	1893	1855	3748	1900	1808	3708	1909	1824	3733
N/A	N/A	50M	15	1												
N/A	N/A	50M	15	2												
20%	10M	50M	15	3a	4105	2809	6914	4423	2692	7115	4311	2905	7216	4280	2802	7082
20%	10M	50M	15	3b												
20%	10M	50M	15	4	2114	2616	4730	2242	2622	4864	2245	2797	5042	2200	2678	4879

Figure 11: I/O Performance with Varying B, k=15 and N=50M

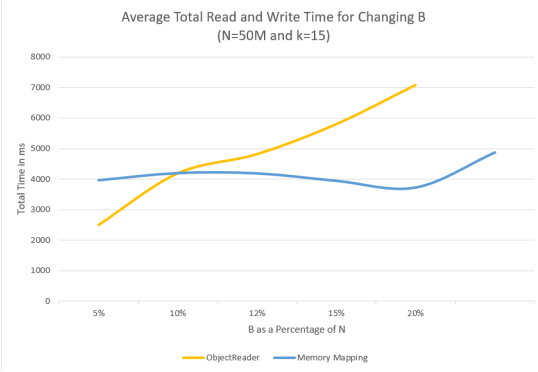

Figure 12: Average Total Read and Write Time for Changing B (k=15 and N=50M)

We also wanted to experiment with larger file size between the two most performant approaches, so we kept N steady at 50M integers, $k = 15$ and kept changing B as can be seen in Table 11. In Figure 12 we can see that the

more B increases the object streams performance decreases while the memory mapping approach either remains steady or increases slightly. This being said it can be noted that the memory mapping approach is less dependent on the size of the buffer. Also, for B less than 10% the object streams approach seems to perform better, while for B greater than 10% the memory mapping approach performs better.

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total Time (ms)	Write Time (ms)	Read Time (ms)	Total Time (ms)	Write Time (ms)	Read Time (ms)	Total Time (ms)	Write	Read	Total
	N/A	50M	15	1												
	N/A	50M	15	2												
	100	50M	15	3a	27136	41551	68687	28272	42913	71185	25984	42075	68059	27,130.67	42,179.67	69,310.33
	100	50M	15	3b												
	100	50M	15	4	2108	2144	4252	2036	1932	3968	2270	2008	4278	2,138.00	2,028.00	4,166.00
	N/A	50M	15	1												
	N/A	50M	15	2												
	500	50M	15	3a	6472	8130	14602	6011	7988	13999	6516	8182	14698	6,333.00	8,100.00	14,433.00
	500	50M	15	3b												
	500	50M	15	4	1933	1995	3928	1983	2056	4039	2081	2068	4149	1,999.00	2,039.67	4,038.67
	N/A	50M	15	1												
	N/A	50M	15	2												
	1K	50M	15	3a	3543	4243	7786	3571	4134	7705	3616	4096	7712	3,576.67	4,157.67	7,734.33
	1K	50M	15	3b												
	1K	50M	15	4	2169	2056	4225	2004	1930	3934	1993	1933	3926	2,055.33	1,973.00	4,028.33
	N/A	50M	15	1												
	N/A	50M	15	2												
	2.5K	50M	15	3a	2446	1875	4321	1961	1819	3780	3436	1685	5121	2,614.33	1,793.00	4,407.33
	2.5K	50M	15	3b												
	2.5K	50M	15	4	2013	1855	3868	2234	2020	4254	1933	1857	3790	2,060.00	1,910.67	3,970.67
	N/A	50M	15	1												
	N/A	50M	15	2												
	5K	50M	15	3a	1577	868	2445	1589	837	2426	1526	893	2419	1,564.00	866	2,430.00
	5K	50M	15	3b												
	5K	50M	15	4	2080	1937	4017	2077	1826	3903	2093	1880	3973	2,083.33	1,881.00	3,964.33

Figure 13: I/O Performance with Varying Realistic B, k=15 and N=50M

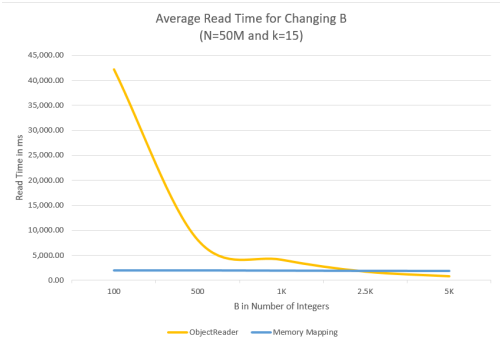

Figure 14: Average Read Time for Changing Realistic B (k=15 and N=50M)

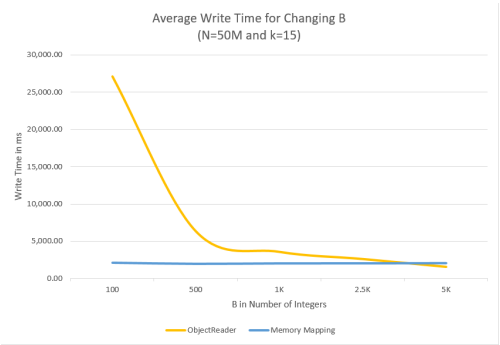

Figure 15: Average Write Time for Changing Realistic B (k=15 and N=50M)

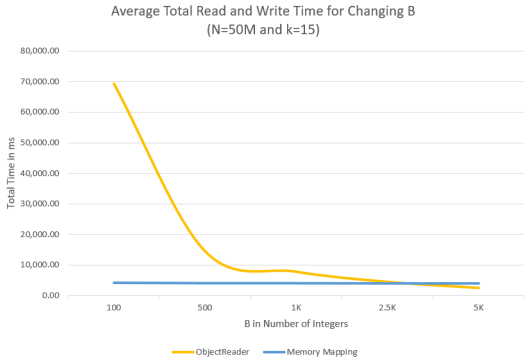

Figure 16: Average Total Read and Write Time for Changing Realistic B (k=15 and N=50M)

However, to make things clearer and find the best performer, we also experimented with more realistic B values, **between 100 and 5K integers** (buffer block of 0.4KB and 20KB respectively). Note that the default B value for buffered streams is 2048 4-byte integers. The results can be found in Table 13. Figures 16, 14 and 15 also verify how dependent object streams are on the buffer size. Specifically, for really small buffers, it requires more than 16X more time when $B = 100$ integers, or more that 3.5X more time when $B = 500$ integers. The gap closes when it comes to default sizes (1024 or 2038 integers, 4096 or 8192 bytes respectively) and for a buffer of 5000 integers object streams are slightly more performant.

However, given that memory mapping shows way more steadier performance, almost independent of the buffer size, we would say that it is the best performer when it comes to varying B values.

2.2.3 Changing k Values

For the third experiment we kept N and B steady and changed k. **k** was given **values between 2 and 30 open streams at the same time**. We experimented only with object streams and memory mapping as these two are the most performant and there was no point of experimenting with less performant approaches.

Based on Figures 18, 19 and 20 we can see that again memory mapping slightly outperforms object streams, while both behave almost linearly for an increasing number of streams, which makes sense as the total amount of

					1st Try			2nd Try			3rd Try			Averages		
B %	B	N	k	Method	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write Time (ms)	Read Time (ms)	Total TIme (ms)	Write	Read	Total
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	2	**3a**	976	286	1262	1045	326	1371	1164	282	1446	1062	298	1360
	2K	50M		**3b**												
	2K	50M	2	**4**	934	282	1216	915	297	1212	857	284	1141	902	288	1190
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	5	**3a**	1270	688	1958	1331	685	2016	1302	704	2006	1301	692	1993
	2K	50M		**3b**												
	2K	50M	5	**4**	1238	675	1913	1082	682	1764	1116	680	1796	1145	679	1824
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	10	**3a**	1921	1450	3371	1827	1426	3253	1758	1395	3153	1835	1424	3259
	2K	50M		**3b**												
	2K	50M	10	**4**	1884	1406	3290	1567	1318	2885	1569	1292	2861	1673	1339	3012
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	15	**3a**	2301	2054	4355	2189	2070	4259	2252	2061	4313	2247	2062	4309
	2K	50M		**3b**												
	2K	50M	15	**4**	1998	2015	4013	1933	1913	3846	1945	1900	3845	1959	1943	3901
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	20	**3a**	2826	2927	5753	2754	2746	5500	2681	2777	5458	2754	2817	5570
	2K	50M		**3b**												
	2K	50M	20	**4**	2980	2680	5660	2401	2720	5121	2599	2669	5268	2660	2690	5350
	N/A	50M		1												
	N/A	50M		2												
	2K	50M	30	**3a**	3582	4135	7717	3671	4177	7848	3715	4186	7901	3656	4166	7822
	2K	50M		**3b**												
	2K	50M	30	**4**	3820	3795	7615	3343	3986	7329	3387	3813	7200	3517	3865	7381

Figure 17: I/O Performance with Varying k, B=2K and N=50M

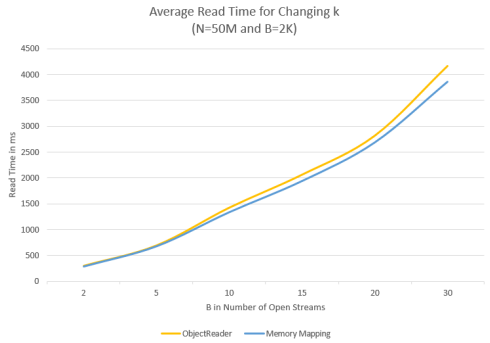

Figure 18: Average Read Time for Changing k (B=2K and N=50M)

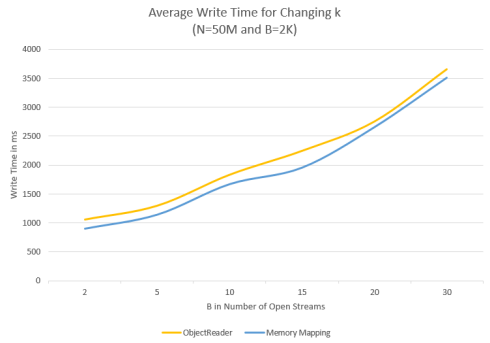

Figure 19: Average Write Time for Changing k (B=2K and N=50M)

data to be read or written also increases linearly. We can conclude that both these approaches performance does not decrease because of multiple open streams.

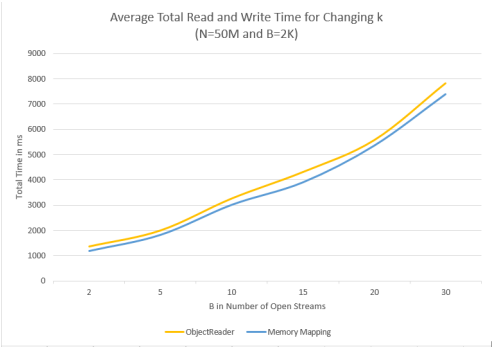

Figure 20: Average Total Read and Write Time for Changing k (B=2K and N=50M)

2.3 Expected vs Experimental Behavior

Generally the benchmarking verified our expected behavior. As mentioned in Section 2.1, we expected system calls to be the slowest of all, buffered streams to be more performant and object streams and memory mapping to be the most performant.

Indeed, system calls were almost impossible to benchmark for larger files, while buffered streams (both default and parametrized) were considerably less performant at least in reading compared to the other two methods. Now, this is a behavior we did not expect, namely the discrepancies between reading and writing times in buffered streams. However, giving it a second thought, it is quite reasonable. Files are not always contiguous on disk, which can affect performance drastically depending on how much space is available and how fragmented the disk is. The more fragmented and full the disc is the less performant these two approaches become. Thus, we believe that a read buffer block might contain more than just file data, which are spread out on disc and subsequently more blocks should be read than the ones described in the strict formulas of Sections 2.1.2 and 2.1.3 in order to fully read each file.

Also, we expected that increasing the B size would positively affect performance. However, for very large B values it either did not affect the performance significantly (See Figure 10) or negatively affected it (See object streams in Figure 12). This might happen because the buffer block size efficiency still depends on the file system block size as discussed earlier, which only depends on the operating system. So increasing B might be limited

by the file system block size, while it also introduces extra cache or RAM latency, which may be the reason for the decreased performance of object streams for unrealistically large B values.

Another interesting conclusion regarding memory mapping is that is appears to not be very dependent on the buffer size contrary to what we expected. The reason behind this behavior might be that the use of mapped files can significantly reduce I/O data movement, since the file data does not have to be copied into process data buffers, as is done by read and write invocations. As buffers are not in the middle layer, B might become less relevant.

Overall, **for the implementation of the Multi-way Merge Sort we choose to utilize Memory Mapping**, because it is the most performant (fastest) of all for most cases according to our benchmarking, it behaves more steadily compared to the object streams regardless the parameters and finally it is not application-bound as happens with object streams (See Section 2.1.4).

3 Observations on Multi-way Merge Sort

3.1 Expected Behavior

Multiway merge sort is a sorting algorithm which keeps on combining d sorted data streams into a single sorted stream until only one sorted stream is left. It follows a divide and conquer approach. In the first pass we read M integers at a time, sort them in memory and write them in a file as a block. This results in N/M sorted files each containing M number of integers. In the following passes, we keep on merging d sorted files into a larger sorted file until we obtain a single sorted file. We take N, d and M as input parameters where N is the number of integers in the initial unsorted file, d is the number of streams to merge together at a time and M is number of integers that can fit in memory. We used mapping and unmapping approach to read and write in the files as this method was proven to be the fastest for reading and writing as seen in Section 2. As this method requires to read and write B element portion of the file together as a block, we decided the block size to be $B = \lceil M/d + 1 \rceil$. We chose this block size because we know that we can merge only d files at a time, so we need to fit d blocks in the memory at a time, one from each file. We also need to keep one block in memory to store

the output so that we can write it all together as a block in the output file. Keeping these constraints into consideration this Block size is ideal to utilize all the available memory. We keep all the files that needs to be merged in a priority queue.

We keep hash maps to keep track of the file channel, input stream, number of integers in a file, number of integers of a file in the priority queue, and total number of integers read from a file against the file number for each file. And we remove this information from all the hash maps once we are finished with a file. Let's go through all of these factors one by one to see why do we need to keep a track of it. We need to know which file channel and input stream belong to a particular file in order to close them, once all of the integers in that file are already written in one of the output files. We should know the number of integers left from one file in the priority queue so that if all the integers from one file are already polled out from the priority queue for the output, we should bring the next block of that file in the memory and priority queue of integers. We need to keep track of the total number of integers in each file to know the size of each file in terms of number of integers. Number of integers read from a file help us in calculation how many remaining integers need to be in the last block of a file.

We keep on dequeuing the files that have been merged from the priority queue containing all files and adding the larger merged file in the same queue until the queue only has one file left, which is then the final sorted file.

We expect the algorithm to have more passes but faster when d is smaller, and to have slower but less number of passes when d is larger. As the file size increases (N) the sorting will be slower.Larger memory size (M) means less number of I/Os which should make the algorithm faster. Following is our calculation of the cost formula:

We know that:

File Size (in integers) = N
Memory Size (in integers) = M
Number of Streams to merge in one pass = d
Block Size = $B = \lceil M/d + 1 \rceil$
Total Number of Blocks = $\lceil N/\lceil M/d + 1 \rceil \rceil = \lceil N(d+1)/M \rceil$
Total Number of Passes = $\log_M \lceil N(d+1)/M \rceil$

As we read and write all the integers of a file in each pass as blocks the total cost in terms of I/O is estimated to be:

$$Cost = 2\lceil N(d+1)/M \rceil \log_M \lceil N(d+1)/M \rceil$$

Table 1: Multiway Merge Sort Performance with Varying N (M = 500K, d = 15)

N (Integers)	Avg. time (ms)
100000	95
500000	228.3
1000000	9519
5000000	42493
10000000	168555
20000000	325266
50000000	1375174
100000000	2891721

3.2 Experimental Observations

Experiments were done on different values of M, N, d keeping 2 of these parameters constant while changing the value of the third one.

In the first experiment we kept value of M constant as 500,000 integers and value of d constant as 15 streams.We increased valued of N in the range of 100 thousand to 100 million. We did not increase the size any further because a file with 100M integers (around 400 MB) was already taking around 48 minutes to complete the sorting. Table 1 shows how average time increases with the increase in total number of integers in the file (N). Figure 21 shows the graph plotted against these values. N as number of integers is plotted on x-axis whereas time in ms is plotted on y axis. It can be seen from the graph that the time taken by multiway merge sort algorithm to sort the file increase linearly with the increase in size just as we expected, as more integers in file means more I/O operation.

In the second experiment we kept value of N constant as 10,000,000 integers and M equals to 100,000 integers. We increased d gradually in the range of 5 to 30. Table 2 has the value of time against each value of d. We increased d by a factor of 5. We did not go any further than 30, as the instructions say that 30 should be used as the maximum value of d unless a lower value is restricted by the operating system. It can be seen from Figure 22 that the graph plotted against time and d shows that as value of d increase, time consumed by multiway merge sort decreases. As with larger d more files are merged together so the algorithm requires less number of passes and this improves performance.

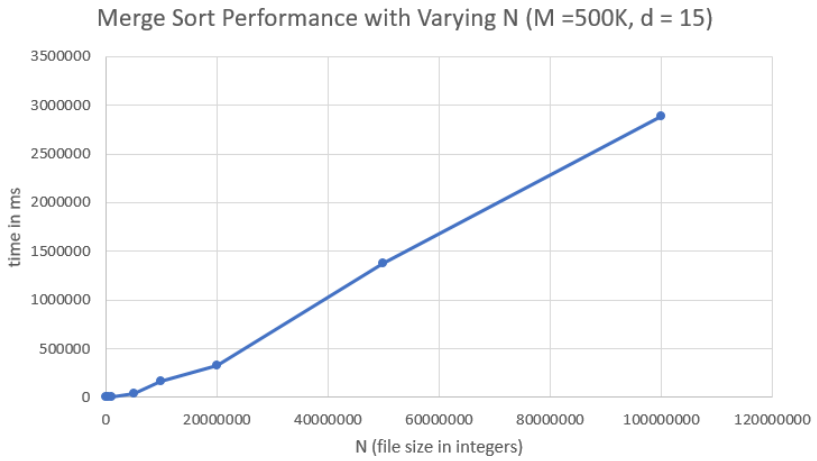

Figure 21: External Multiway Merge Sort Performance Varying N

Table 2: Multiway Merge Sort Performance with Varying d (M =100K, N = 10M)

d	**Avg. time (ms)**
5	135481
10	60594.5
15	58920
20	34877.5
25	30827.33
30	26622.67

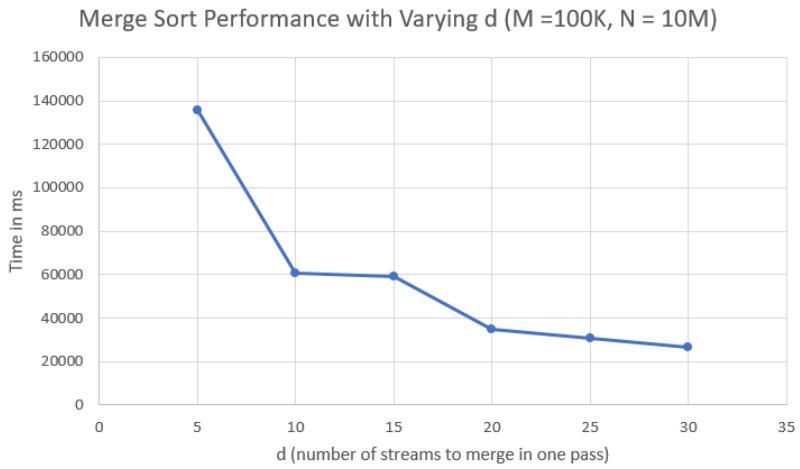

Figure 22: External Multiway Merge Sort Performance Varying d

In the third experiment we changed the value of M while keeping N = 15,000,000 integers and d = 15. The range of M was kept between 1000 to 3,000,000. Table 3 shows time in ms against different values of M. When these values were plotted in a graph with M on the x-axis and time on y-axis, we noticed that at first increasing M was decreasing the time, hence improving performance. But after one point (25000), increasing M was also increasing the time it takes to sort the data. The performance kept on worsening with the increase in M. However at one point (1,000,000) we noticed that performance improved a little, but after this point the total time started to increase again linearly with increasing M. Larger memory size means that we can fit bigger blocks in the memory which should result in less I/Os but it could be possible that after a certain point, I/Os become faster than sorting large number of integers in the memory because larger memory size means that more integers fit into memory, which means more comparisons need to be done to sort all the integers in Memory (See Figure 23.

In experiment four we also kept file size and d constant but with values N = 50M and d = 15 to see the behavior of increasing memory size with an even larger file and check if this file performs better with larger memory size (M). Table 4 shows the values used for this experiment which were plotted in a graph as seen in Figure 24. It can be seen that when M was very small (5000), it took a lot of time to finish. First with the increase in M the performance improved but later increasing M was decreasing the performance in terms of

Table 3: Merge Sort Performance with Varying M (N =15M, d = 15)

M	Avg. time (ms)
1000	204348
5000	72849
10000	59231
20000	56631
25000	44112.66
30000	59278
50000	69822.33
100000	83700
300000	213280.5
500000	347736
750000	344217
1000000	271381
1500000	372535.66
2000000	495705.66
3000000	687483.33

Table 4: Merge Sort Performance with Varying M (N =50M, d = 15)

M	Avg. time (ms)
5000	223749
10000	174810
20000	155229
25000	155296
100000	297319
1000000	1907003
1500000	2842011
2000000	3102787

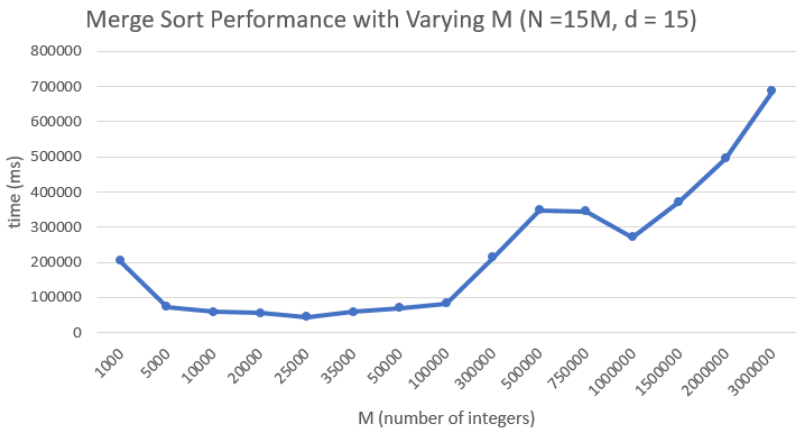

Figure 23: Merge Sort Performance with Varying M

time it is taking to finish the sorting even for very large file (50M integers).

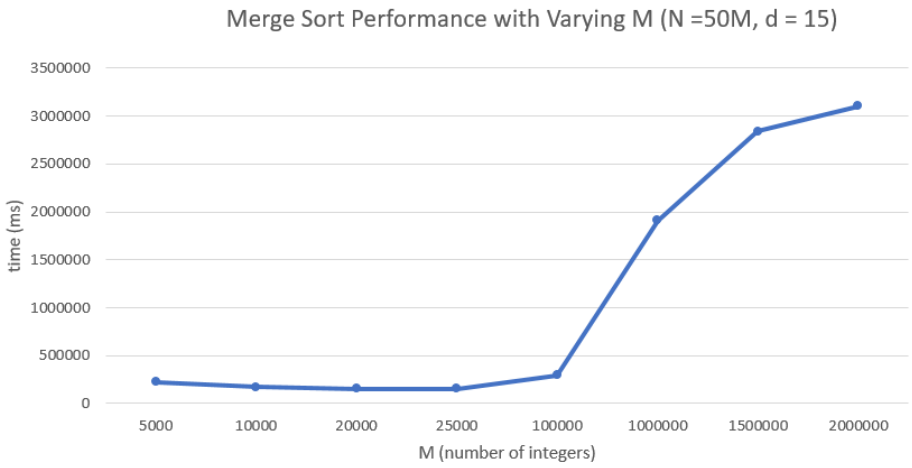

Figure 24: Merge Sort Performance with Varying M, N = 50M

3.3 Experimental Vs. Expected Behavior

We noticed while experimenting with different values that increasing the size of the file (N) also increases the time it takes to finish the sorting. This was exactly what we expected. More integers in the file result in more I/Os, as in each pass we read and write all the integers. Hence the algorithm takes more time to finish the sorting. We also noticed that on varying d the results were also as we expected. When the value of d was increased the performance of algorithm improved. As larger d means we are merging more files together at a time, which will result in less passes and faster completion. When N (file size) was kept really small, even smaller than M, we noticed that the time taken for it to finish sorting was less than a second which is comparable to the in-memory sorting algorithms.

While experimenting with M we noticed that the behavior was different than what we expected. We expected that as M will increase the performance of the algorithm will improve as the block size will be larger and it will require less I/Os. In the start the performance was as expected when we increased M from 1000 until around 25000 but after this point further increasing M was worsening the performance of algorithm which was on contrary to what we were expecting. This could be because when memory can fit more integers,

this results in more comparisons in the heap to sort those integers which takes more time.

For smaller input files we can keep M equals to N and value of d as N/M, so that we can merge all the files at once in the memory, similar to in-memory sorting algorithms.Even for really large input files like 500M if you keep N = M it is really fast because it is like an in-memory algorithm sorting all the integers at once in the memory, but practically M can not be as big as the data. For large input files we should keep M between 25000 to 27000 and we can keep d as 30 to get better performance. If we compare Table 1 and Table 3 we can see that larger memory size even for very large files is taking way more time to finish than if we keep the memory size smaller.

4 Conclusion

In this project, first we tried 5 different methods of reading and writing to files and we noticed that if we write to the files integer by integer the process is very slow as compared to if we do these operations block by block. The size of block also plays a very important role in the performance of these operations. We noticed that memory mapping method was the fastest method to read and write into the external memory. It reads and writes B blocks at a time by mapping and unmapping them into the internal memory using memory mapping. So we chose memory mapping method for reading and writing to external memory in our multiway merge sort algorithm. We then implemented external memory multiway merge sort using JAVA. While implementing multiway merge sort we came across and solved the issues like integers in a file will not always be multiples of B and sometimes last block will have less number of integers. We derived the cost formula for multiway merge sort using the cost formulas we studies in the class. We experimented with different values of N, M and d to see the performance of the algorithm. We noticed that if size of memory is small than cost of memory operations are negligible but if size of memory is really big than in memory operations take more time because it has to deal with a large number of integers altogether so size of memory is also very important. We also noticed that increasing d improves the performance because more files are merged in one pass which results in less number of passes.