CSC443 Assignment 1.1 and 1.2

Renata Boodram (boodram8, 999872320)
Tyler Pham (phamtyle, 999643382)

Part 1: Random versus Sequential I/O

All experiments were run 5 times each as required. As suggested, the input filename was changed between the 5 runs and between testing on different block sizes to minimize the operating system's data buffering. The machine used for all experiments was one of the computers in the Teaching Labs which runs on Ubuntu 14.04.5 LTS.

Experiment 1: Finding Optimal Block Size

Using the short C program on the online stats man page (http://man7.org/linux/man-pages/man2/stat.2.html), the block size on the Teaching Lab computers (CDF) was determined to be 4096 bytes.

The first experiment involved running write_blocks_seq with various block sizes in order to determine the most optimal one (the one with the lowest run time). The average results of the 5 runs are shown in the tables below (and graph):

Block size (bytes)	Data rate (MB/s)
512	34.907
1024	35.090
4096	34.862
8192	35.626
16,384	35.478
32,768	35.311
1,048,576	35.866
2,097,152	35.901
4,194,304	35.826

According to the results, the optimal block size is 8 KB. This differs from the system block size of 4 KB. The data rate keeps increasing until it reaches 8 KB. However, there is not a strictly non-increasing sequence in data rate as the data rate raises when using a block size of 8 KB. The inconsistency could be due to some underlying OS caching despite best efforts to avoid so. The method in which we attempted to avoid OS caching involved us rebooting the physical machine, copying the CSV file to a new filename associated with a run number (e.g., “edges_run0.csv” for the first run) and deleting the output binary file in between each call to write_blocks_seq.

Running write_lines resulted in a higher throughput - the values of the 5 runs are:

- Run 1: 132.626 MBPS
- Run 2: 132.626 MBPS
- Run 3: 151.152 MBPS
- Run 4: 143.166 MBPS
- Run 5: 154.230 MBPS

The average of write_lines was 142.76 Megabytes/second. The reason why write_lines has a higher throughput despite writing more data (there are more lines than blocks) could be attributed to system calls. Depending on the implementation of the system call used to write the file (e.g., fputs versus fwrite), this could cause the OS to automatically buffer data. Also, there was less processing involved in this program (no need to parse and read the lines into a struct) which may also account for the higher data rate in write_lines.

Experiment 2: Sequential vs. Random Read Rate

Read_blocks_seq **[1024*1024][1 MB] [hard disk]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
MAX:	80699	80699	80699	80699	80699	80699
Average	9.84	9.84	9.84	9.84	9.84	9.84
Data rate (MBPS)	110.686	108.343	105.274	115.034	121.340	112.1354

Read_blocks_seq **[1024*1024][1 MB] [SSD]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
MAX:	80699	80699	80699	80699	80699	80699
Average	9.84`	9.84	9.84	9.84	9.84	9.84
Data rate (MBPS)	1729.754	1765.790	1765.794	1513.534	1614.437	1677.8618

Read_ram_seq **[FIRST LOAD +MAIN MEMORY] [hard disk]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
MAX:	80699	80699	80699	80699	80699	80699
Average	9.84	9.84	9.84	9.84	9.84	9.84
Data rate (MBPS)	1541.053	1541.053	1599.206	1500.140	1474.051	1531.1006

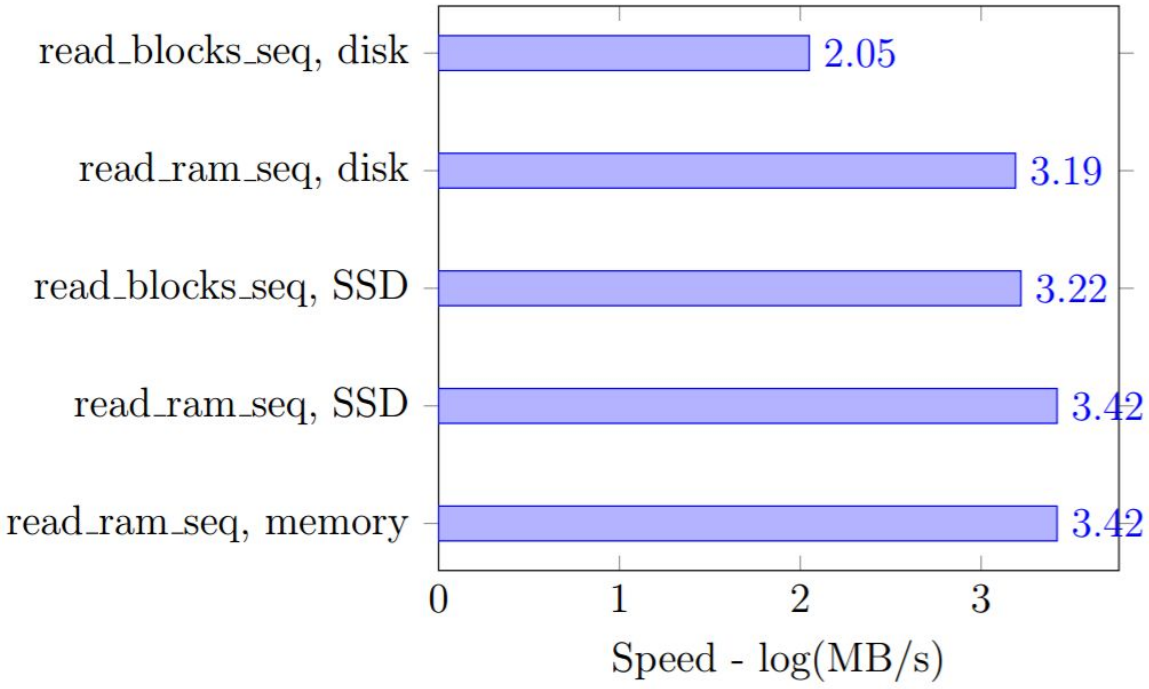
Read_ram_seq **[FIRST LOAD +MAIN MEMORY] [SSD]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
MAX:	80699	80699	80699	80699	80699	80699
Average	9.84	9.84	9.84	9.84	9.84	9.84

Data rate (MBPS)	2648.685	2690.728	2690.728	2648.685	2690.728	2673.9108

Read_ram_seq **[STARTING IN MAIN MEMORY] [SSD]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
MAX:	80699	80699	80699	80699	80699	80699
Average	9.84	9.84	9.84	9.84	9.84	9.84
Data rate (MBPS)	2648.685	2607.936	2648.675	2607.936	2648.685	2632.3834

Results for random access with a file larger than available main memory:

big.dat **[1024*1024] 1MB** **[hard disk]**	**test1**	**test2**	**test3**	**test4**	**test5**	**AVERAGE**
Read_blocks_rand (hard disk)	15.810	42.301	41.017	23.563	39.494	32.437
Read_ram_rand (hard disk)	2000.00	2083.333	1785.714	2500.000	1818.182	2037.4458
Read_blocks_rand (SSD)	1538.462	1754.386	1333.333	1351.351	1538.462	1503.1988
Read_ram_rand (SSD)	2631.579	2702.703	2702.703	2631.579	2702.703	2674.2534



[HARD DISK RESULTS]
The ratio of sequential read rate of secondary storage[HARD DISK] vs primary storage [MAIN MEMORY] is:
[53.2M values/sec]/[358.2M values/sec] = 0.1485

The observed ratio from our experiments of sequential read rate of secondary storage[HARD DISK] vs primary storage (First load + MAIN MEMORY) is:
[112.1354 MBPS]/[1531.1006 MBPS] = 0.0732

The observed ratio from our experiments of sequential read rate of secondary storage[HARD DISK] vs primary storage [CACHED INTO MAIN MEMORY] is:
[112.1354 MBPS]/[2632.3834 MBPS] = 0.0426

[SSD RESULTS]
The ratio of sequential read rate of secondary storage[SSD] vs primary storage (Main memory) is:
[42.2M values/sec]/[358.2M values/sec] = 0.1178

The observed ratio from our experiments of sequential read rate of secondary storage[SSD] vs primary storage [FIRST LOAD + MAIN MEMORY] is:
[1677.8618 MBPS]/[2673.9108 MBPS] = 0.6373

The observed ratio from our experiments of sequential read rate of secondary storage[SSD] vs primary storage [CACHED INTO MAIN MEMORY) is:
[1677.8618 MBPS]/[2632.3834 MBPS] = 0.6374

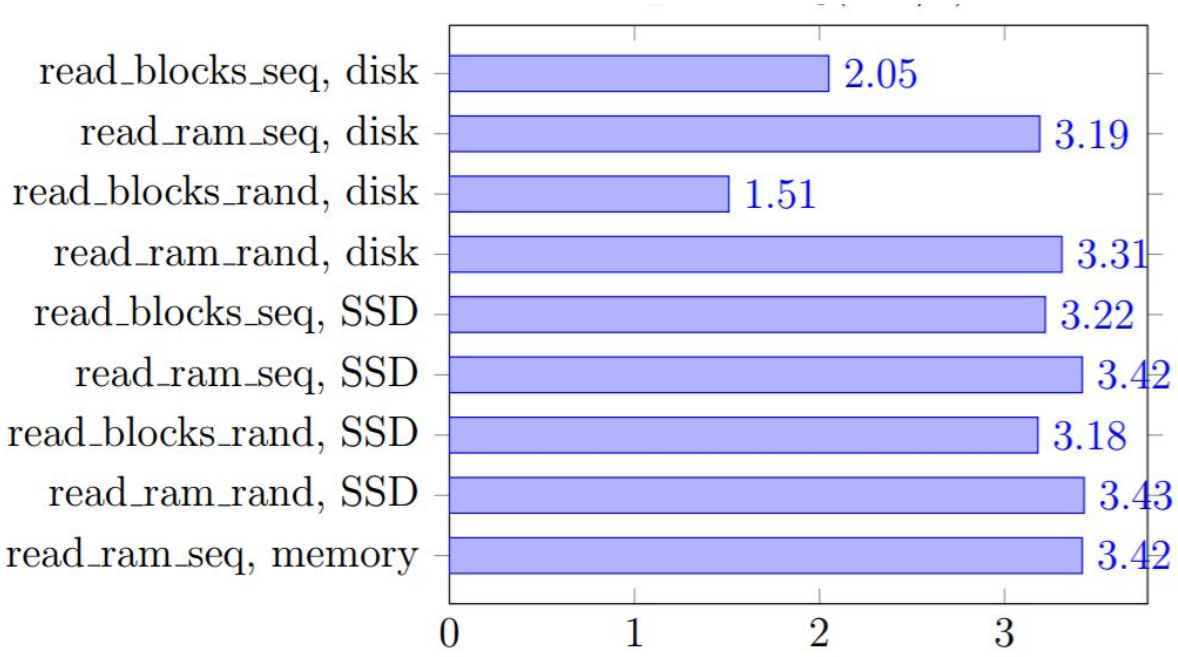
as per the results the ratios (Jacobs) discussed in class for secondary storage vs primary storage sequential reads were as follows
0.1485 for HARD DISK
0.1178 for SSD

The ratio for the hard disks were very similiar [0.1485 vs 0.0732 and 0.0426] though the ratio for the SSD were not, the ratio differing by a factor of almost 6. One explanation of the ratios being very different could be the specifications of the SSD used vs Jacobs, another could possibly be due to human error in the programming of the programs.

Not much are known abouts the specs of the “SSD” drive we used for the experiment, however we do know from the Jacobs diagram that the drive they used was a
“Intel High Performance SSD”. Since not all SSD’s are built equal if Jacob’s was significantly faster in read and write times then that would explain why our ratio would be larger.

As per the results the ratio discussed in class for primary vs secondary storage sequential reads were as follows:
0.1485 for HARD DISK
0.1178 for SSD

The ratio for hard disks are very similiar [0.1485 vs 0.0732 and 0.0426] though the ratio for SSD are not the ratio differing by a factor of almost 6. One explanation of the ratios being very different would be the specs of the SSD. Not much is known about the specs of the SSD used for testing and SSD's can very greatly in access times. Without knowing the specifications of the hardware it is hard to conclude if the tests are comparable. However, the general hierarchy of access times were held with sequential accessing being faster than random and data loaded into primary memory being faster than secondary storage (which can be seen in the below graph). At the bottom of the Jacobs model all the hardware was cited before a valid comparison is made a performance comparison must be made with the CDF computers to determine if the ratio was more strictly held. Until then this loose correlation will suffice.



Experiment 3: Sequential vs. Random Read Rate

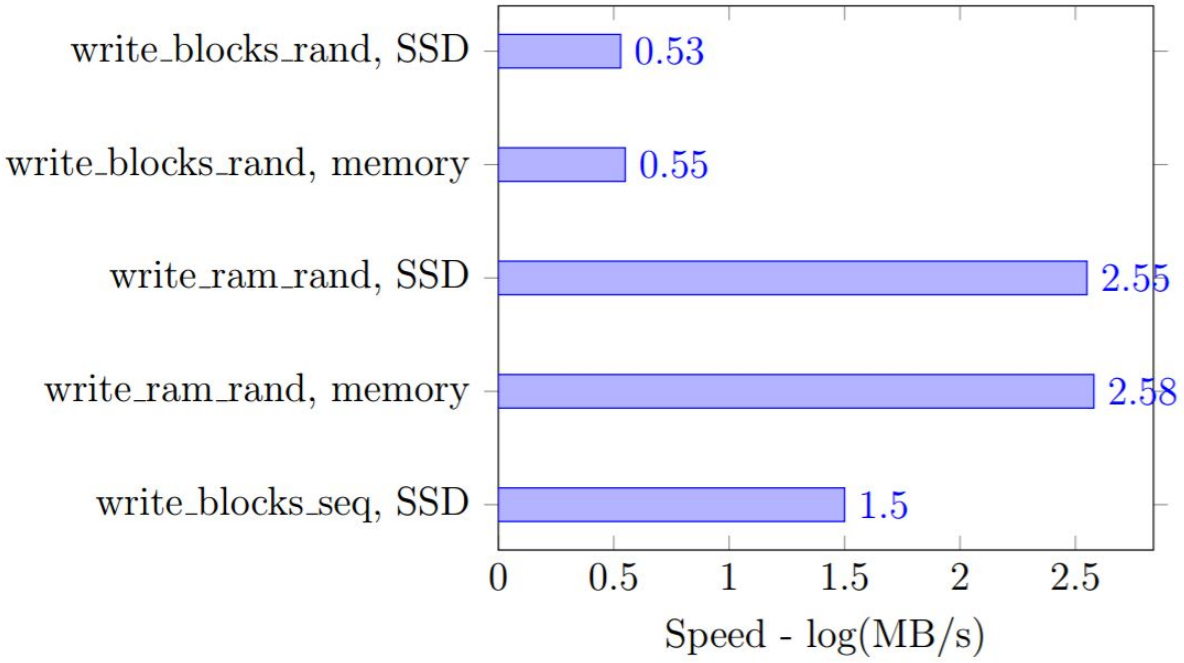
During this experiment, we observed that running write_ram_rand was much more efficient than write_blocks_rand. Although we were unable to capture the data rate when using write_ram_rand with a smaller number of updates (due to the granularity of the clock), we were able to compare for at least 10,000 updates and above.

Number of updates	write_ram_rand Average (SSD) (MB/s)	write_ram_rand Average (main memory) (MB/s)	write_blocks_rand Average (SSD) (MB/s)	write_blocks_rand Average (main memory) (MB/s)
100	Unavailable	Unavailable	0.763	0.763
1000	Unavailable	Unavailable	2.925	2.543
10,000	76.264	76.294	3.292	2.725
1,000,000	356.0386	381.470	3.406	3.516

For 10,000 updates, write_blocks_rand only performed at about 4% of write_ram_rand. This is expected as writing blocks to disk takes longer due to the physical location of the disk within the computer.

Comparing this to the results obtained in 3.1 for write_blocks_seq, write_blocks_rand performed much slower, never exceeding 4.0 MB/s while on average, write_blocks_seq ran for 32.05 MB/s on average. This demonstrates that although both programs are writing to disk, writing sequentially is much quicker

than writing randomly to disk. If everything is written sequentially, there is no need to jump across sectors or blocks, reducing the transfer time. These results can be seen in the graph below.



Summary

Our group learned that accessing data will always be faster if you need to access it sequentially. In order to access it in the fastest way possible you will need to manipulate things such as block size in order to gain the desired speeds. These experiments persuaded our group that different algorithms are needed if working with primary or secondary storage. If one was writing a program where multiple writes were involved, they would want to use a buffer to cut down on the amount of secondary storage accesses. They would have to consider this more carefully when operating with secondary storage as secondary is much more expensive to operate on than main memory.

Part 2: External sort in- and out-degree distribution

Introduction

Disk_sort is a program that uses the 2-Phase Multiway Merge Sort Algorithm to sort Twitter data records in a binary data file by UID2. For all experiments except the timing experiment, the input memory had to be halved as qsort would consume about 200 MB of memory if used. The memory consumption of qsort was tested within the disk_sort program by commenting out any calls to qsort.

Experiment 1: Timing

The timing experiment involved running the disk_sort program with 200MB of available memory. The results of the 5 runs using the Unix built-in time program is shown below:

Run number	Real time (seconds)	Maximum Resident Set Size (kilobytes)
1	18.47	206280
2	18.85	206296
3	17.86	206336
4	19.60	206276
5	17.86	206308
Average	18.528	206299.2

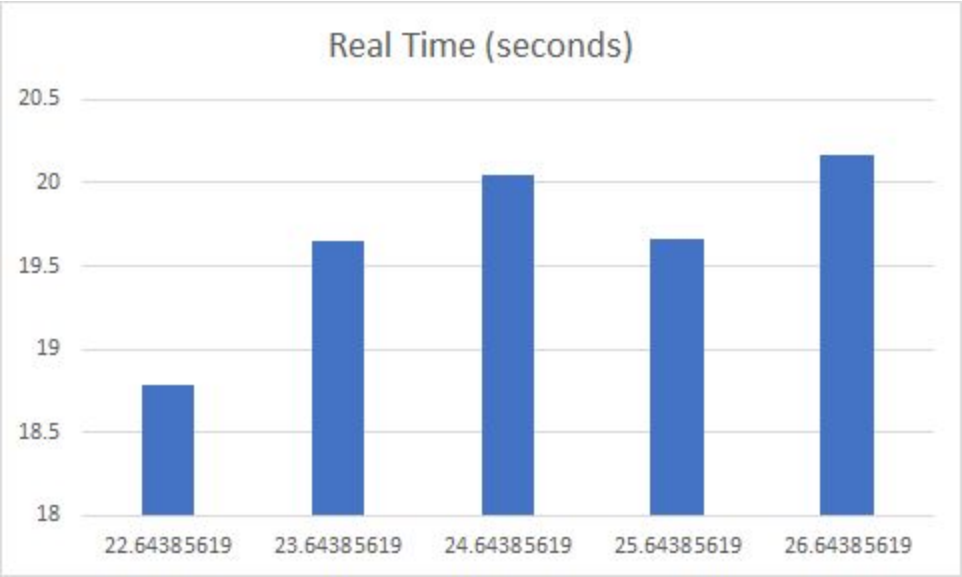
The small over-consumption that can be seen could be due to qsort. According to http://en.cppreference.com/w/c/algorithm/qsort, qsort makes no guarantee about complexity and thus cannot be guaranteed to make consumptions about memory.

Experiment 2: Buffer size

The buffer size experiment involved using fractions of the original 200 MB until the 2-pass algorithm could not hold the number of required sublists and block size in memory. The experiment was run on various power-of-2 fractions ($\frac{1}{2^n}$) of the original total memory, 200 MB. As in the previous experiment, the small overconsumption of memory can be due to the qsort call used in the disk_sort program.

The results for the experiment can be seen on the next page:

Fractional relation to 200 MB	Total Memory (bytes)	Real time (seconds)	Maximum Resident Set Size (kilobytes)
1/2	104,857,600	18.78	103840
1/4	52,428,800	19.65	52700
1/8	26,214,400	20.05	27108
1/16	13,107,200	19.66	14368
1/32	6,553,600	20.17	219072



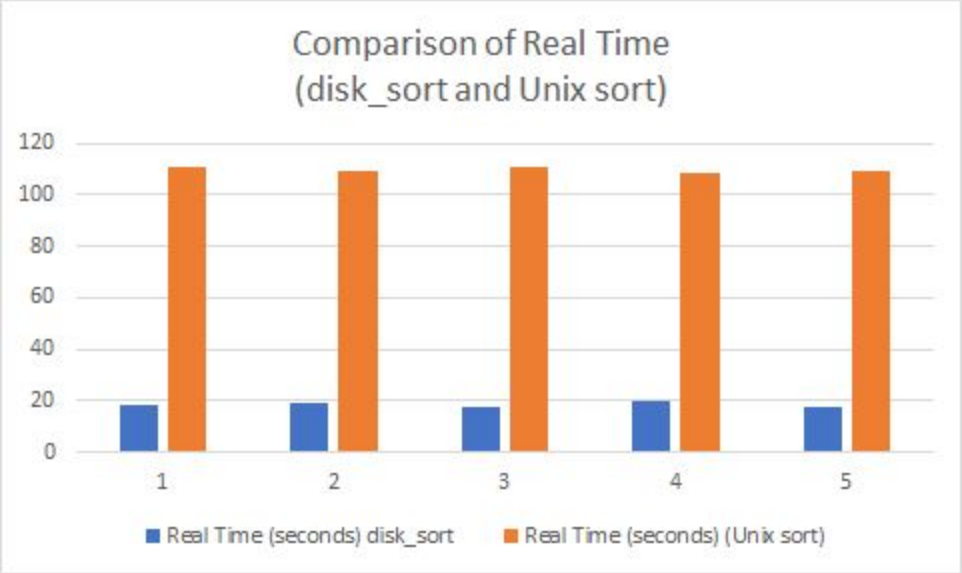
Based on the experiments we did using different amounts of main memory, run times remained fairly similar. The amount of memory given to run the program was reduced each iteration and by the most extreme case (reducing the available memory to 1/32 approximately 6 MB of the original 200MB) run times increased marginally. This is mostly consistent with the 2PMMS theory the runtime of the program did not heavily depend on the number of total runs and size of the input and output buffers. However as with many programs by limiting the resources available, ultimately will limit the speed at which the algorithm can perform at. So, by limiting the size of main memory, we are limiting the amount of blocks the size of our optimal block size that can fit in main memory.

Experiment 3: Comparing to Unix sort

As part of our experiment, we ran Unix sort to sort the data and compared it to

Run	Real time (seconds)	Maximum Resident Segment Size (kilobytes)
1	110.86	1836784
2	109.42	2269940
3	110.68	2799220
4	108.84	2997392
5	109.56	3093176
Average	109.872	2599302.4

In comparison to our disk_sort program, Unix sort is significantly slower and has a higher memory consumption. Unix sort also also uses the multi-way merge sort algorithm as well, so there are probably differences in values provided to our disk_sort and Unix sort.



The command that we were required to run, "sort -t ',' -n -k2 edges.csv" does not specify any limitation on total memory that the program is required to use. This explains the drastic increase in maximum resident set size when using Unix sort. Using the "-S" option, we could have specified a certain amount of main memory that would be used as a sorting buffer. To explain the time discrepancies, Unix sort could be slow because it needs to parse each line, find the key position to sort on ("k2" option) and convert this string to a numeric value to make a numeric conversion ("n" option). Our disk_sort program does not have to do parsing but is able to quickly read the binary data into a buffer and retrieve the automatically converted numeric value for the sort comparison.

Summary

Including Part 1 of this assignment, we have learnt many things. In Part 1, we learnt the benefits of sequential reading and writing over the random versions of these two functions. We also learned the benefits of finding a correct optimal block size.

In Part 2, we learnt how to implement the 2-Phase Multiway Merge Sort algorithm and the relation of our chosen optimal block size with the total memory restraint on this program. With less memory, we are able to hold a smaller amount of blocks in memory which has a marginal affect on the runtime.