

**Member 1:** Ajinkya Vjiay Sonawane  
**Net ID:** avs8687  
**NYU ID:** N10004179

**Member 2:** Roshni Sen  
**Net ID:** rs7633  
**NYU ID:** N15508945

# CS-GY 6233 Operating Systems (Final Project)

## 1. Basics

Write a program that can read and write a file from disk using the standard C/C++ library's **open**, **read**, **write**, and **close** functions.

- Add parameter for the file name;
- Add parameter for how big the file should be (for writing);
- Add a parameter to specify how much to read with a single call (blocksize);

Initially, we created a program to handle disk reads and writes. `Part_1.c` is the name of the source code and **run** is the executable object file, and it has been posted to our Github repository. This is not the same program that should be used in the raw program performance evaluation because it doesn't use multi-threading and uses block size specified by the user as CLI arguments. There is a shell script, `build.sh`, which can be used to compile both this application and the rest of the project's programs.

**Use the following instructions to run this program:**

`./run <filename> [-r|-w] <block_size> <block_count>`

```
ajinkyasonawane@Ajinkyas-Air Project % ./run ubuntu-21.04-desktop-amd64.iso -r 524288 10000
xor: a7eeb2d9
```

The xor of all 4-byte numbers in the input file will be printed by this application. The program **part\_6.c** or the **fast** executable object, which will be detailed in section 6, will improve the performance.

```
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems % ./run Part1/random.txt -w 4096 20
File of size : 81920 bytes created
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems % █
```

A file of the given name will be created (if it does not already exist) as per the given `BLOCK_SIZE` and `BLOCK_COUNT`. The file will contain random characters generated using the random function.

## 2. Measurement

Write a program to find a file size which can be read in "reasonable" time.

- Input: filename & block size
- Output: file size

Using the program from Part 1, we created two programs - part\_2.c (takes filename as an input and reads increasing chunks of it and returns the block count needed for the read operation to take between 5 seconds to 15 seconds based on the block(buffer) size given as the input ) and part\_2\_FileCreate.c (generates file with random data and doubles the size of the file till the read operation does not take reasonable time).

**Use the following instructions to run this program:**

`./run2 <filename> <block_size>`

```
ajinkysonawane@Ajinkyas-Air Project % gcc Part2/part_2.c -o run2
ajinkysonawane@Ajinkyas-Air Project % ./run2 ubuntu-21.04-desktop-amd64.iso 128

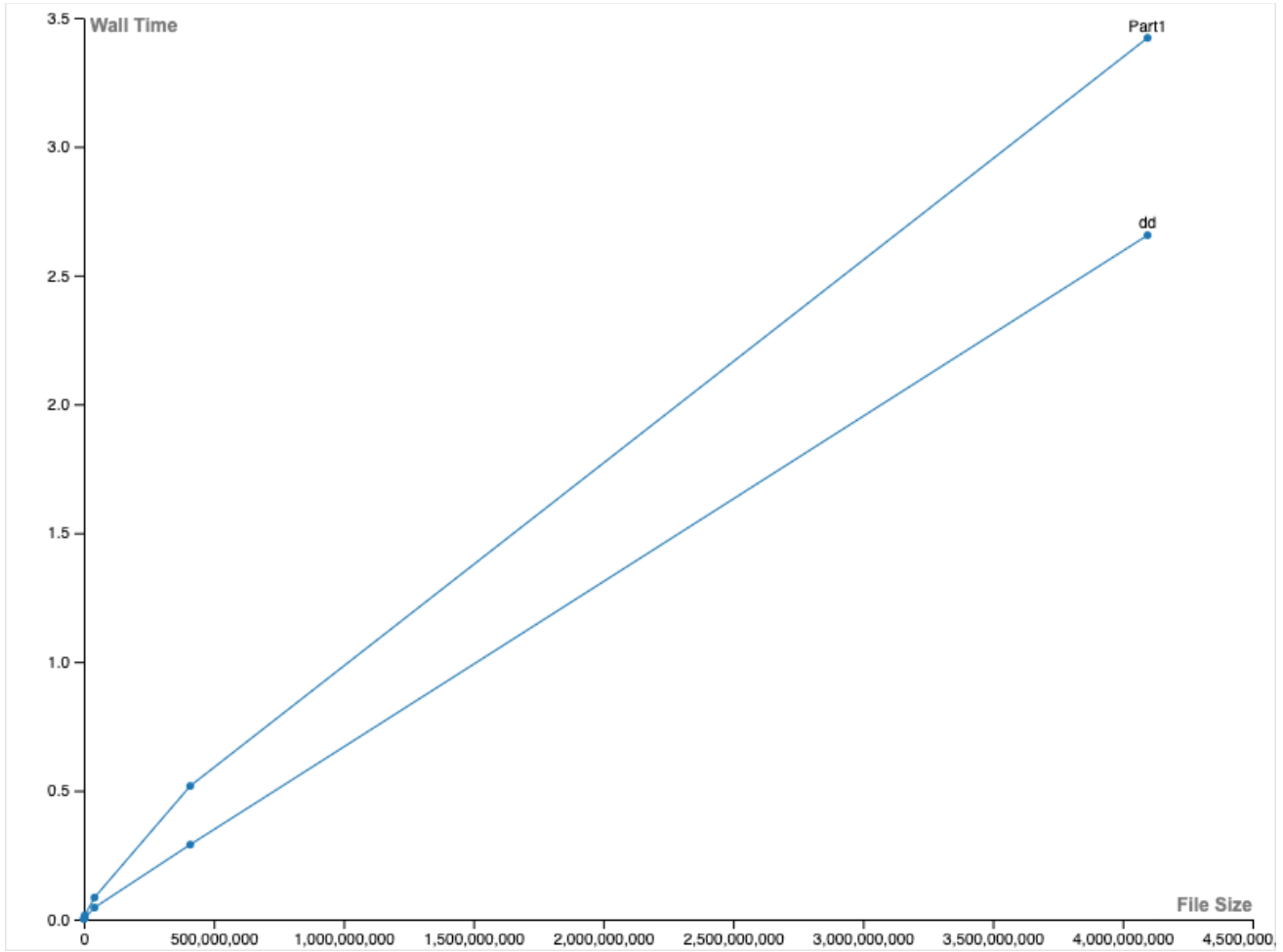
Max Filesize : 2818738176 bytes | Final Block Count : 33554432
ajinkysonawane@Ajinkyas-Air Project % █
```

The Max Filesize is the size required for the program to perform read operation with the input block size of 128 bytes to be completed in over 5 seconds.

**Extra credit idea:** learn about the dd program in Linux and see how your program's performance compares to it!

The dd program is used to copy a file, converting and formatting according to the operands. We compared it with our program that performs read and write operations as shown in Part 1. We write a file and then read it as our program does not copy from an input file and write to another one. For different given block size and given block count, we created files using dd as well as our program and compared the results. The dd performed better than our Part 1 program and following are the results from the comparison.

File Size (bytes)	dd - Wall Time (secs)	Part1 - Wall Time(secs)
409600	0.00171	0.002449
4096000	0.005807	0.016384
40960000	0.045815	0.084664
409600000	0.290108	0.517696
4096000000	2.655796	3.422016

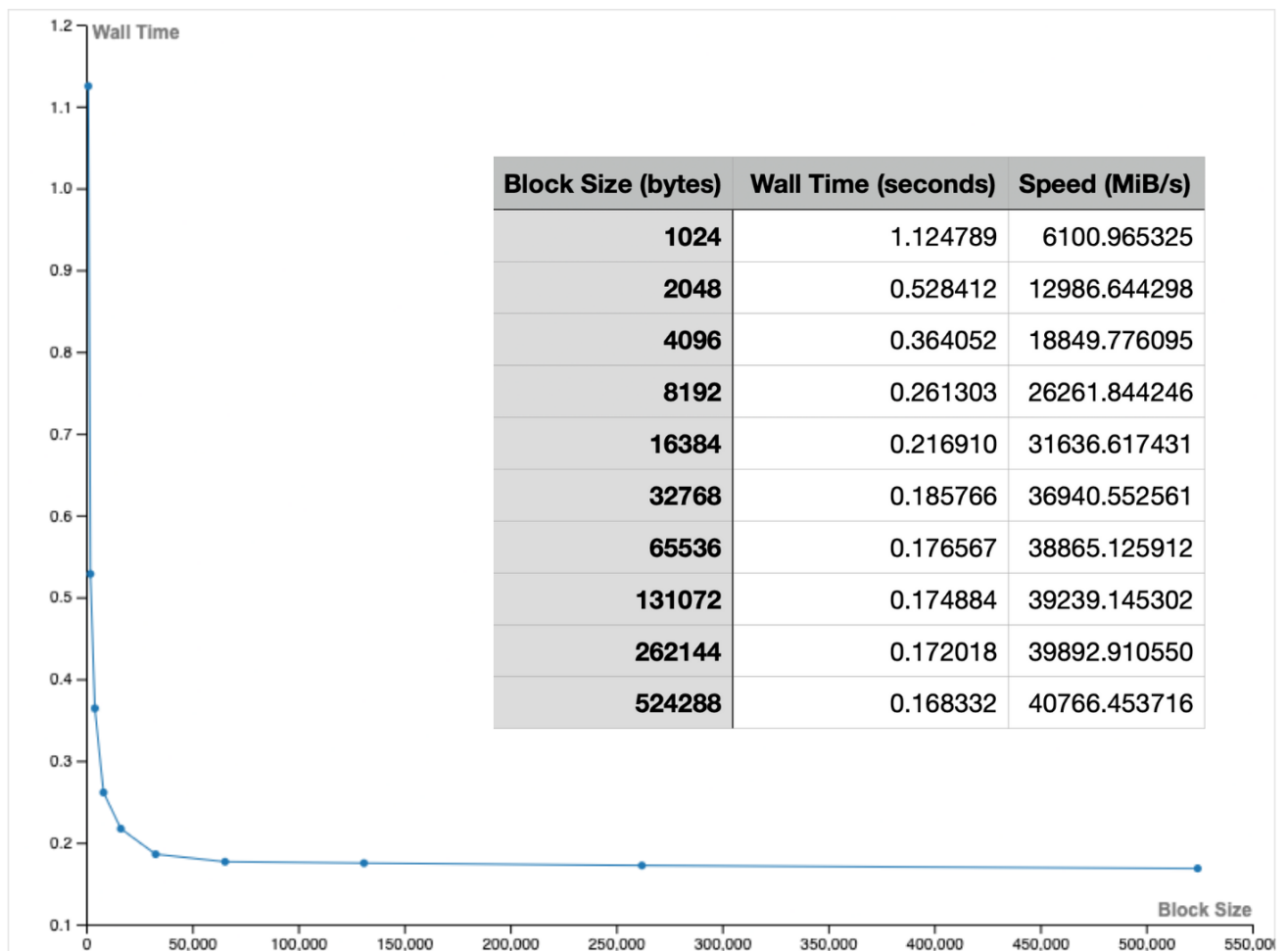


[X-axis : File Size (bytes), Y-axis: Wall Time (seconds)]

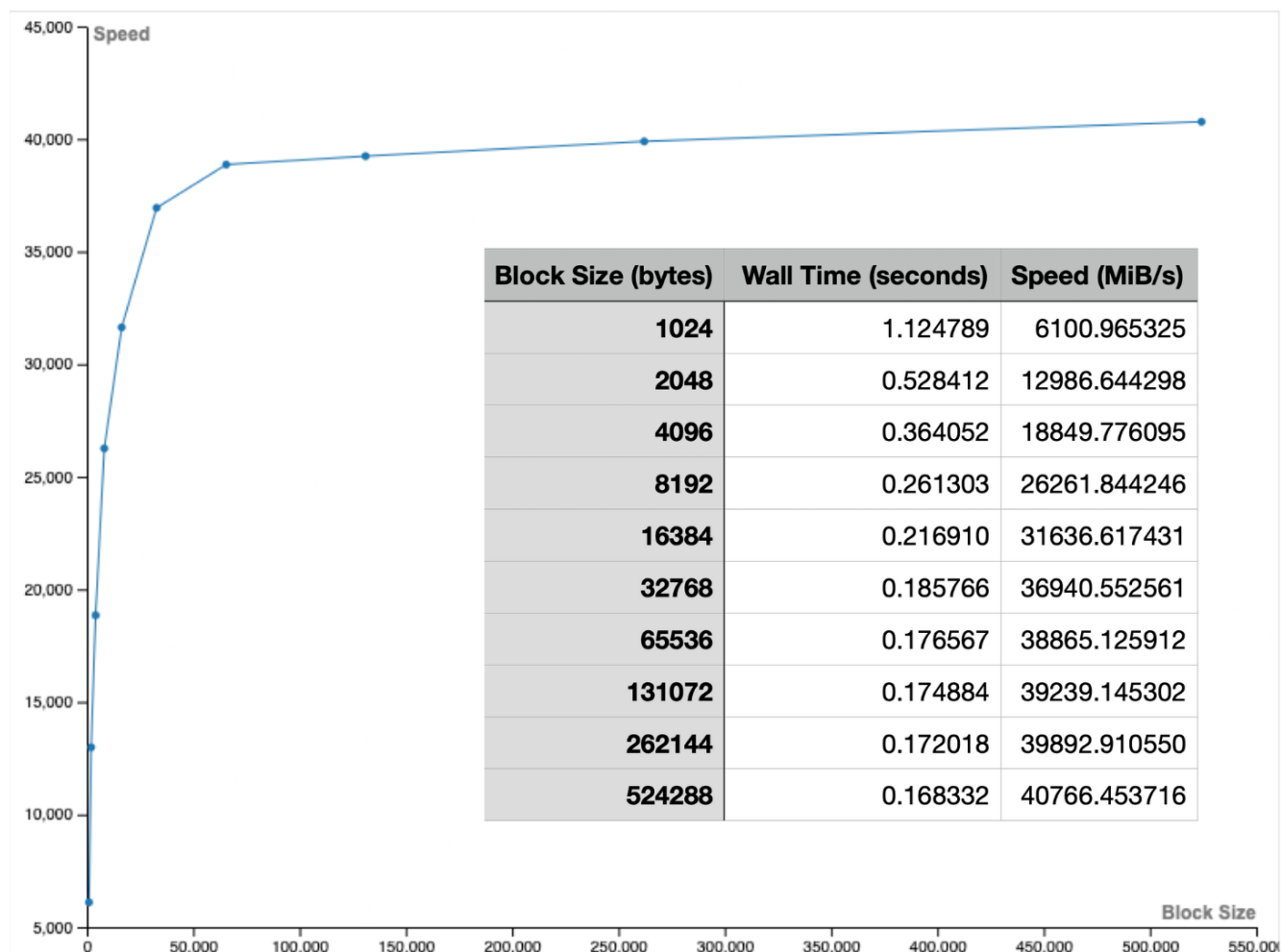
### 3. Raw Performance

- Make your program output the performance it achieved in MiB/s.
- Make a graph that shows its performance as you change the block size.

To measure the raw performance of our read system call, we ran the program for different block sizes and plotted the graph below. We can see that bigger block size is faster but after a point the speed plateaus and the program cannot read any faster. The line chart shows the same, the time taken to read reduces as the block size increases significantly and then plateaus.



**[X-axis : Block Size (bytes), Y-axis: Wall Time (seconds)]**

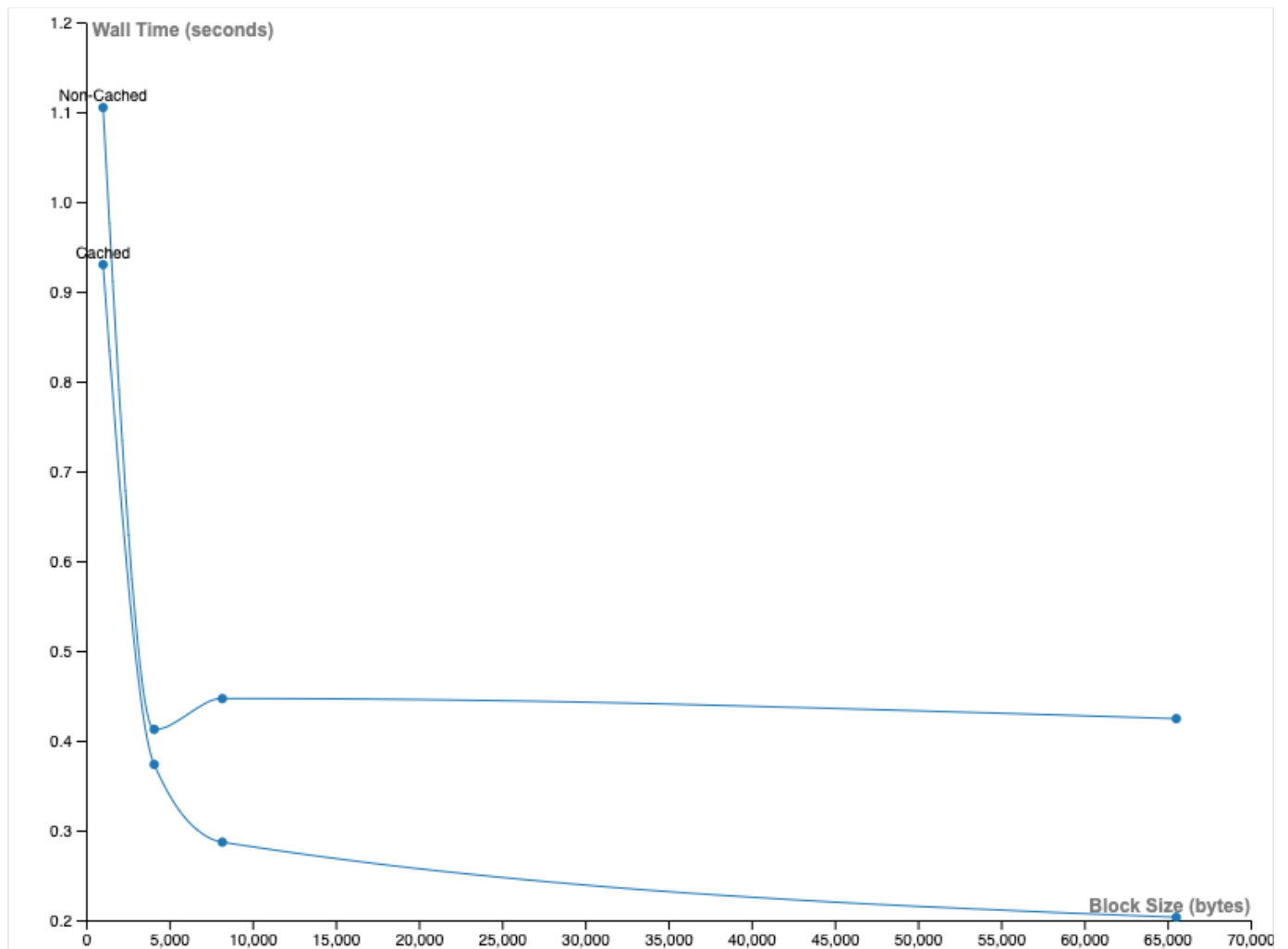


**[X-axis : Block Size (bytes), Y-axis: Speed (MiB/s)]**

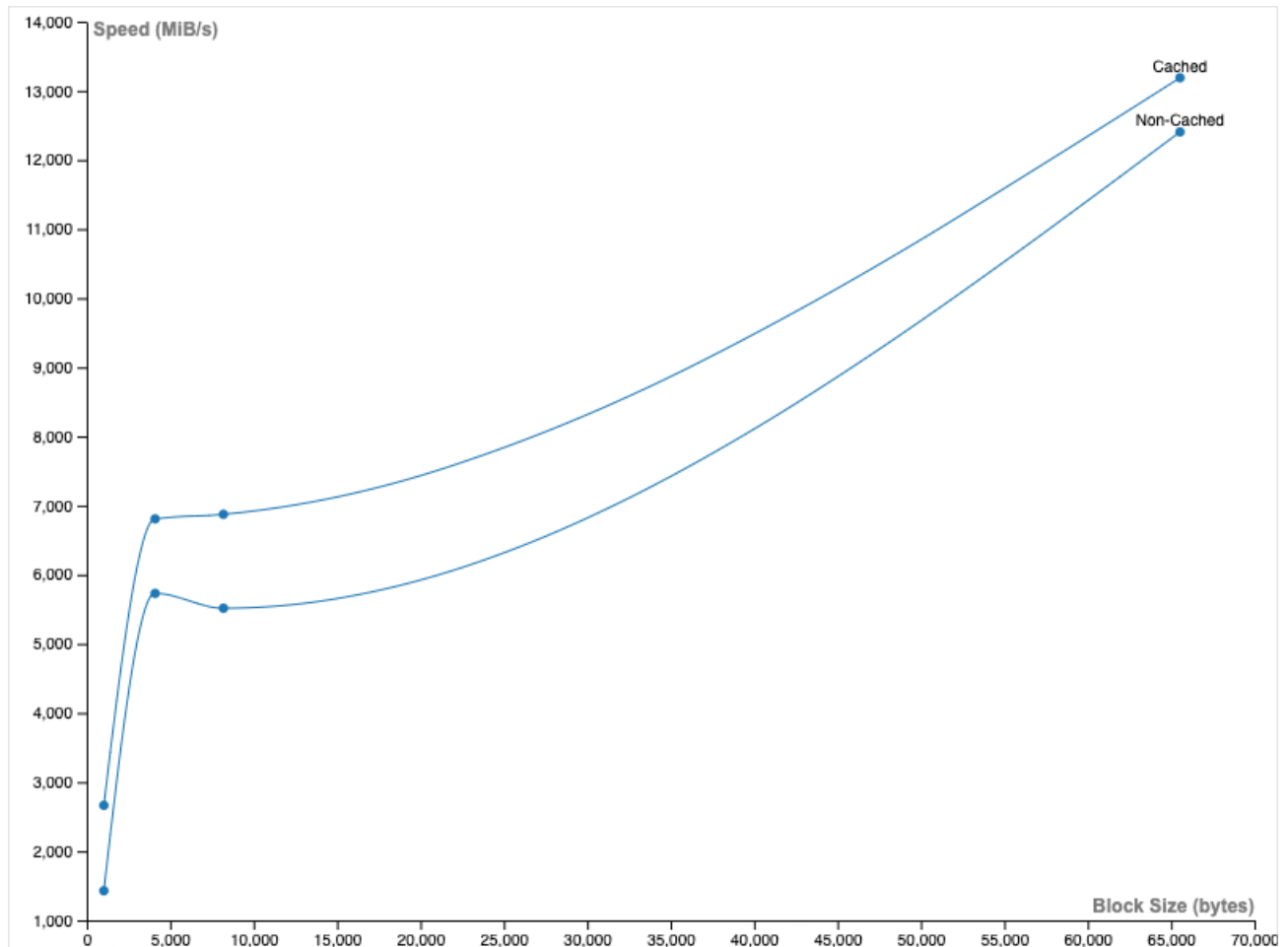
**Note:** This program DOES NOT make use of threads and the read operation is carried out by the main process. We will use the BLOCK\_SIZE of 524288 bytes in Part 6 as it gives us the fastest reading time from above.

## 4. Caching

This part focuses on reading the file and comparing how reading time differs based on cached data. The following line chart shows the difference in read time with different block sizes. Cache was cleared by restarting the system and we recorded the wall time for each execution. We can see that read is faster obviously with cached data as the pages are loaded into the cache recently and can be fetched easily as compared to non-cached where the data has to be brought from the disk.



[X-axis: Block Size (bytes) | Y-axis: Wall Time (seconds)]



[X-axis: Block Size (bytes) | Y-axis: Speed (MiB/s)]

On Linux there is a way to clear the disk caches without rebooting your machine. E.g. `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"`.

**Extra credit:** Why "3"? Read up on it and explain.

→ There are three ways to clean cache on a Linux system without disrupting any processes or services. Only the Page Cache is cleared when the value 1 is used. Dentries and inodes are cleared at value 2. Value 3 clears the page cache, dentries, and inodes, among other things. ([Reference](#))

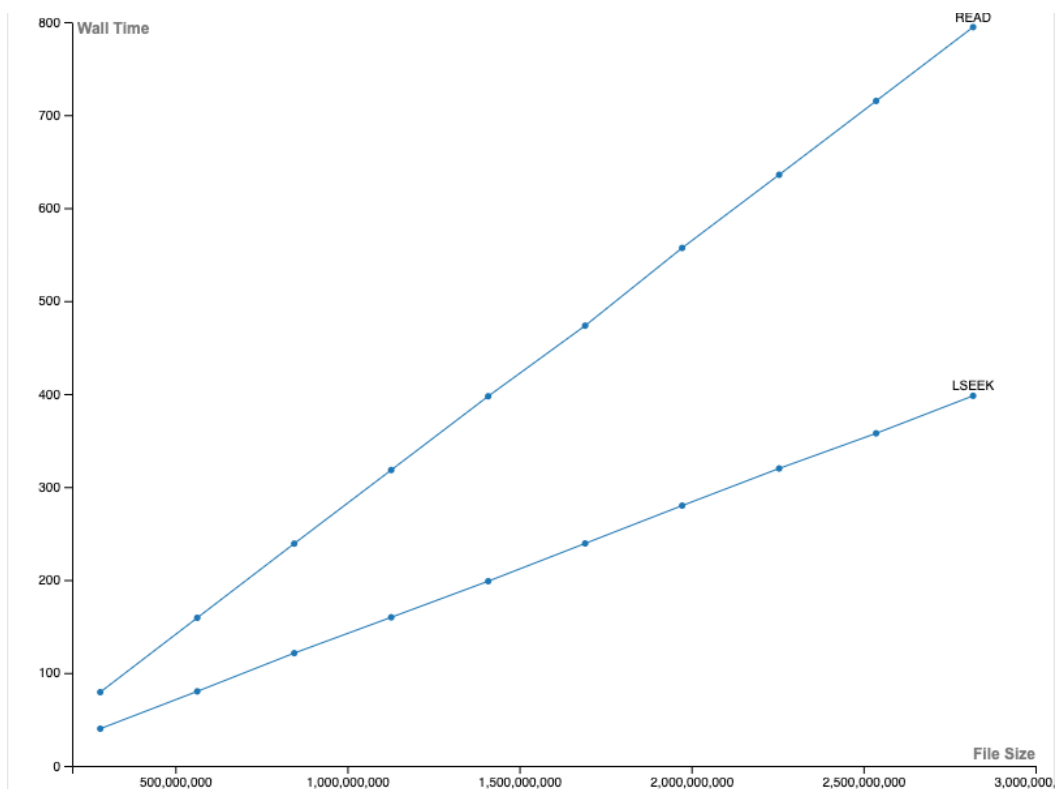
## 5. System Calls

In this part we focus on how the system calls perform and how much time they take for execution. We compare our **read** system call with **lseek** and see how much time it takes for both of them to finish execution on a given file for a given block size. Although, lseek does not actually do any work, it just sets the offset of the file descriptor by a specified number of bytes (in our case the block size = 1).

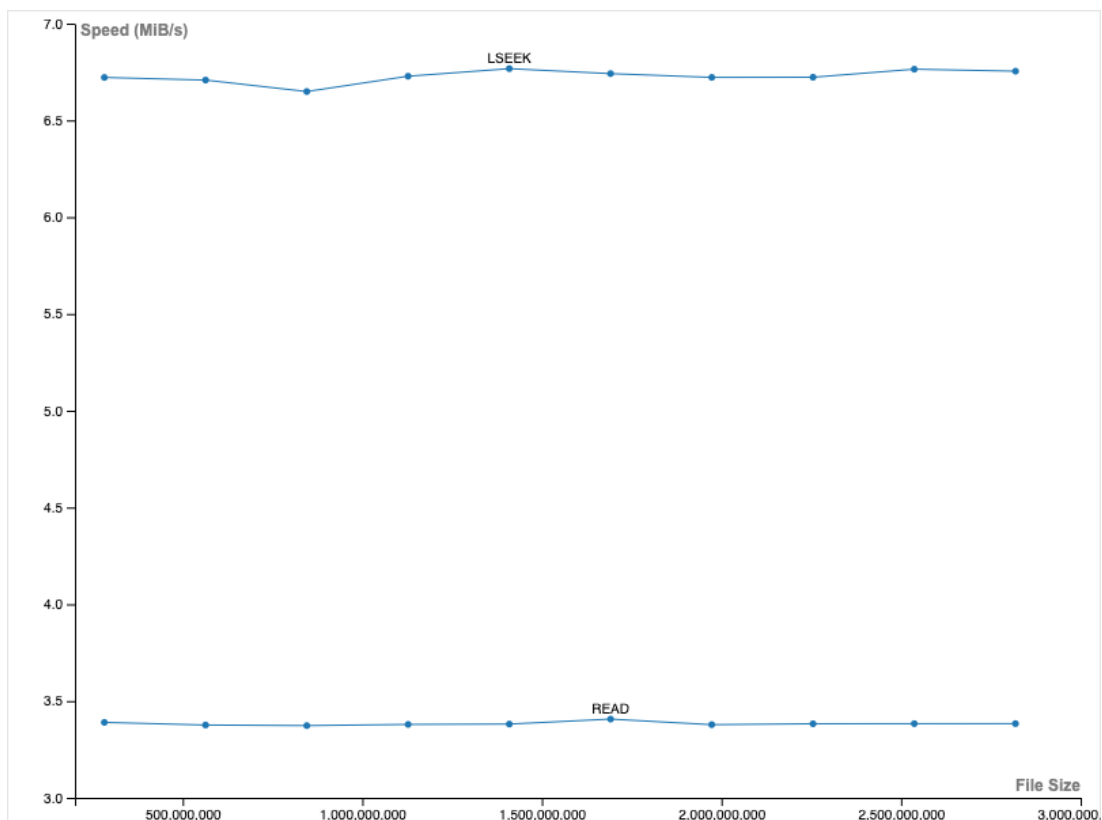
The following snip shows the execution time for READ as well as LSEEK system call for the given file and block size of 1 byte. We can see the difference in execution time as the work done by both these system calls is different and the speed MiB/s.

Type	File Size	Wall Time	Speed (B/s)	Speed (MiB/s)
READ	281873817	79.302903	3554394.685904	3.389735
LSEEK	281873817	39.992212	7048217.712989	6.721704
READ	563747634	159.244219	3540145.052299	3.376145
LSEEK	563747634	80.146826	7033935.866656	6.708084
READ	845621451	239.079816	3536983.862327	3.373131
LSEEK	845621451	121.288424	6971988.118174	6.649006
READ	1127495268	318.199525	3543359.368623	3.379211
LSEEK	1127495268	159.818650	7054841.647079	6.728021
READ	1409369085	397.529714	3545317.583480	3.381078
LSEEK	1409369085	198.619846	7095811.991517	6.767094
READ	1691242902	473.444248	3572211.319800	3.406726
LSEEK	1691242902	239.251300	7068897.439638	6.741426
READ	1973116719	557.055597	3542046.304940	3.377959
LSEEK	1973116719	279.929652	7048616.339508	6.722084
READ	2254990536	635.757722	3546933.773618	3.382620
LSEEK	2254990536	319.885830	7049360.504653	6.722794
READ	2536864353	715.130166	3547416.223804	3.383080
LSEEK	2536864353	357.659686	7092955.824493	6.764370
READ	2818738170	794.535839	3547653.902620	3.383306
LSEEK	2818738170	398.003321	7082197.613120	6.754110





[X-axis: File Size (bytes) | Y-axis: Wall Time (seconds)]



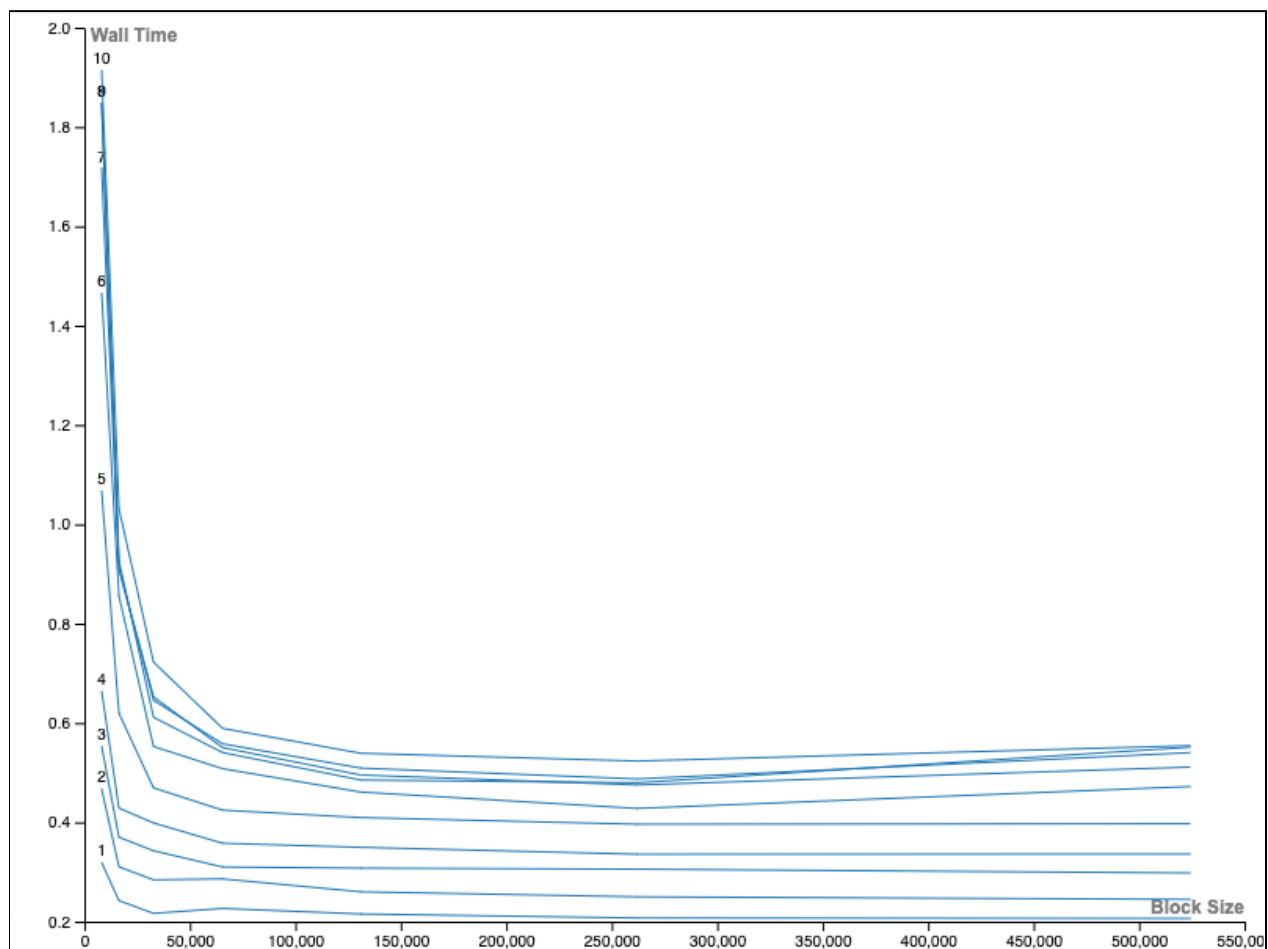
[X-axis: File Size (bytes) | Y-axis: Speed (MiB/s)]

## 6. Raw Performance

Try to optimize your program as much as you can to run as fast as it could.

- Find a good enough block size?
- Use multiple threads?
- Report both cached and non-cached performance numbers.

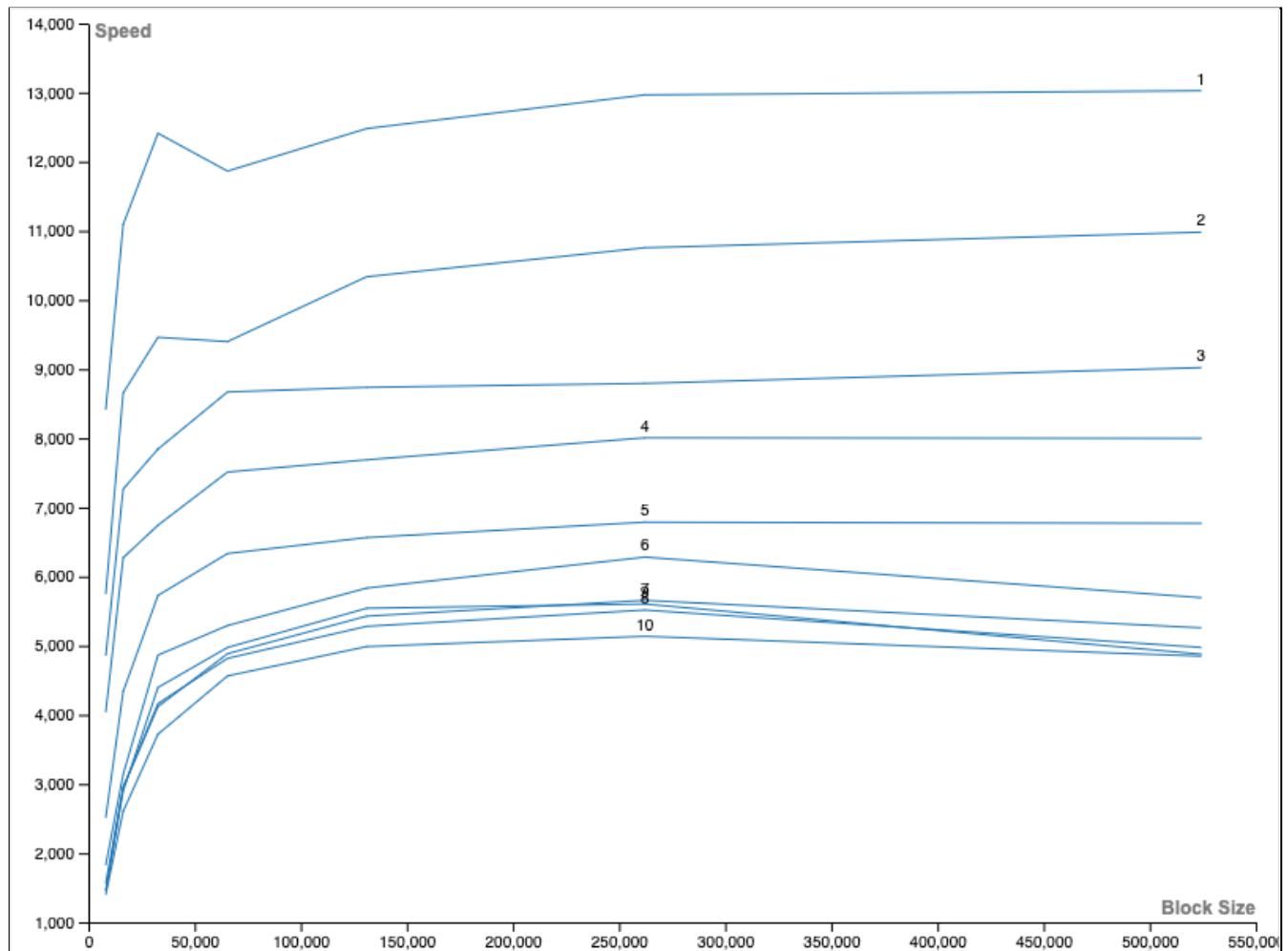
This part focuses on optimizing the read operation using a good enough block size and by checking if threads help make the read execute faster. We looped over different block sizes (buffer) and used a different number of threads with each block size. The following line chart shows how the performance degrades with the use of threads and from *PART 3*, we know that the performance plateaus after a point and increasing the block size does not make a difference.



[X-axis: Block Size (bytes) | Y-axis: Wall Time (seconds)]

Lines: Number of threads]

**Comparison between number of threads and block sizes vs performance(speed) wall time reading & XORing time:**



**[X-axis: Block Size (bytes) | Y-axis: Speed (MiB/s)**  
**Lines: Number of Threads]**

The above multi-set bar chart shows how the execution time differs and eventually plateaus as we increase block size and also how the execution time is high with higher number of threads, especially without a smaller block size

Our basic assumption is that multi-threading speeds up the program significantly. But as we see from the above graphs, introducing threads does not really improve the performance. As we are reading from the disk, the threads will try to access the disk simultaneously which can lead to heavier

resource sharing and the overhead of creating and managing threads also adds up to the overall execution time.

**Following is the comparison between threaded and non-threaded version of fast.c and we can see that non-threaded program runs faster:**

```
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems % gcc -O3 fast.c -o fast
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems % ./fast ubuntu-21.04-desktop-amd64.iso

Without multi-threading
Block Size      : 524288
xor             : a7eeb2d9
Wall Time (seconds) : 0.234221
Speed (MiB/s)    : 11477.016165
Speed (B/s)     : 12034523701.973776

With multi-threading
Block Size      : 524288
Number of threads : 3
xor             : a7eeb2d9
Wall Time (seconds) : 0.285919
Speed (MiB/s)    : 9401.817309
Speed (B/s)     : 9858519986.429724
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems %
```

**Following is the output of fast.c (Final Code as submitted):**

The fast.c program is compiled using the flag -O3 for optimized compilation. With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. -O3 turns on all optimizations specified by -O2 and additional optimization flags. ([Reference](#))

```
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems % ./fast ubuntu-21.04-desktop-amd64.iso

Without multi-threading
Block Size      : 524288
xor             : a7eeb2d9
Wall Time (seconds) : 0.227792
Speed (MiB/s)    : 11800.933321
Speed (B/s)     : 12374175458.312847
ajinkyasonawane@Ajinkyas-Air CS-GY_6233_OperatingSystems %
```

**Note:** All the above experiments were conducted on MacBook Air M1 with 16 GB RAM and 512 GB SSD.

**Github Repository:**

[https://github.com/Ajinkya-Sonawane/CS-GY\\_6233\\_OperatingSystems](https://github.com/Ajinkya-Sonawane/CS-GY_6233_OperatingSystems)