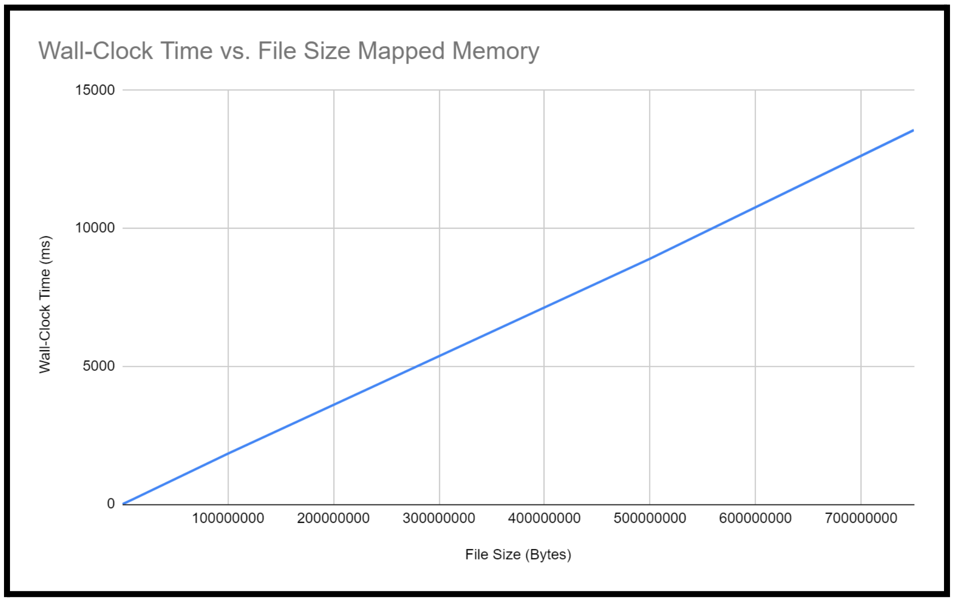
Operating Systems Analysis of Chartype Processing Methods

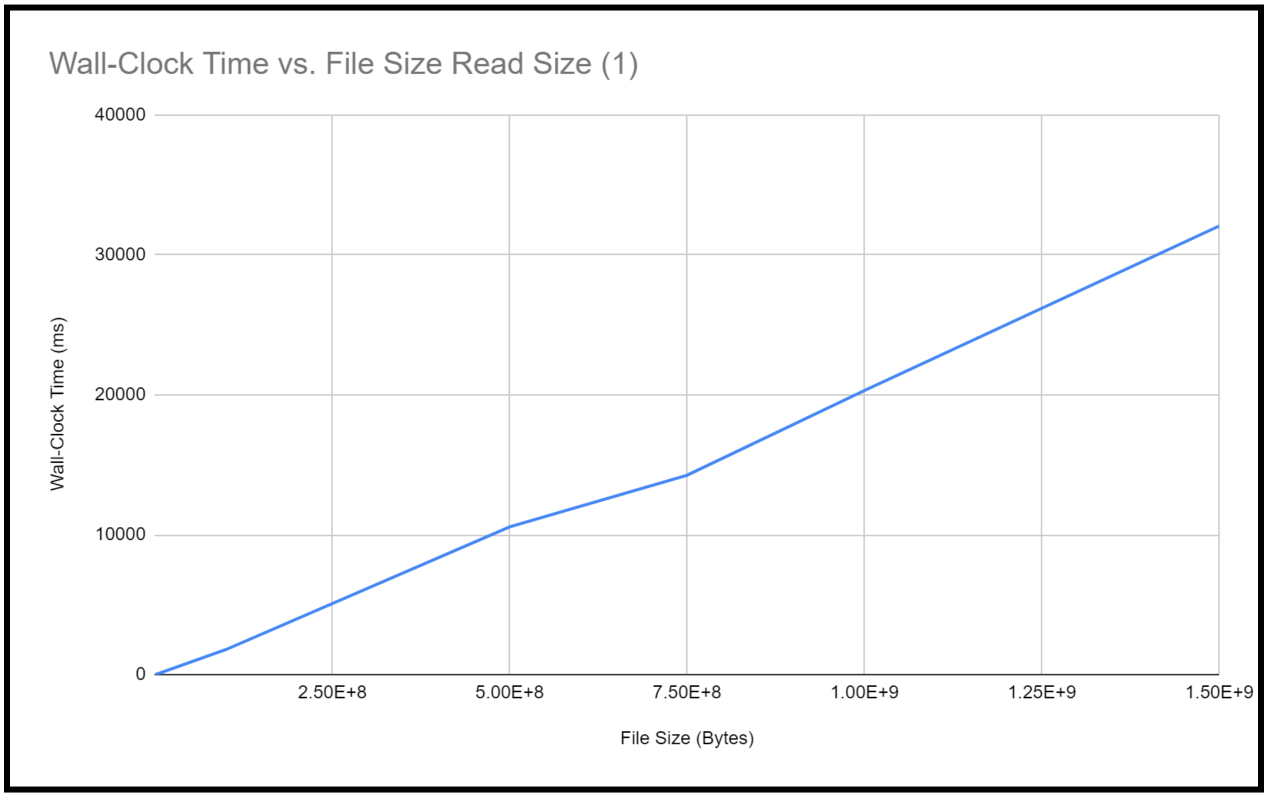
Mapped Memory:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	178
100000000	1833
500000000	8900
750000000	13564
1000000000	18408
1500000000	27023



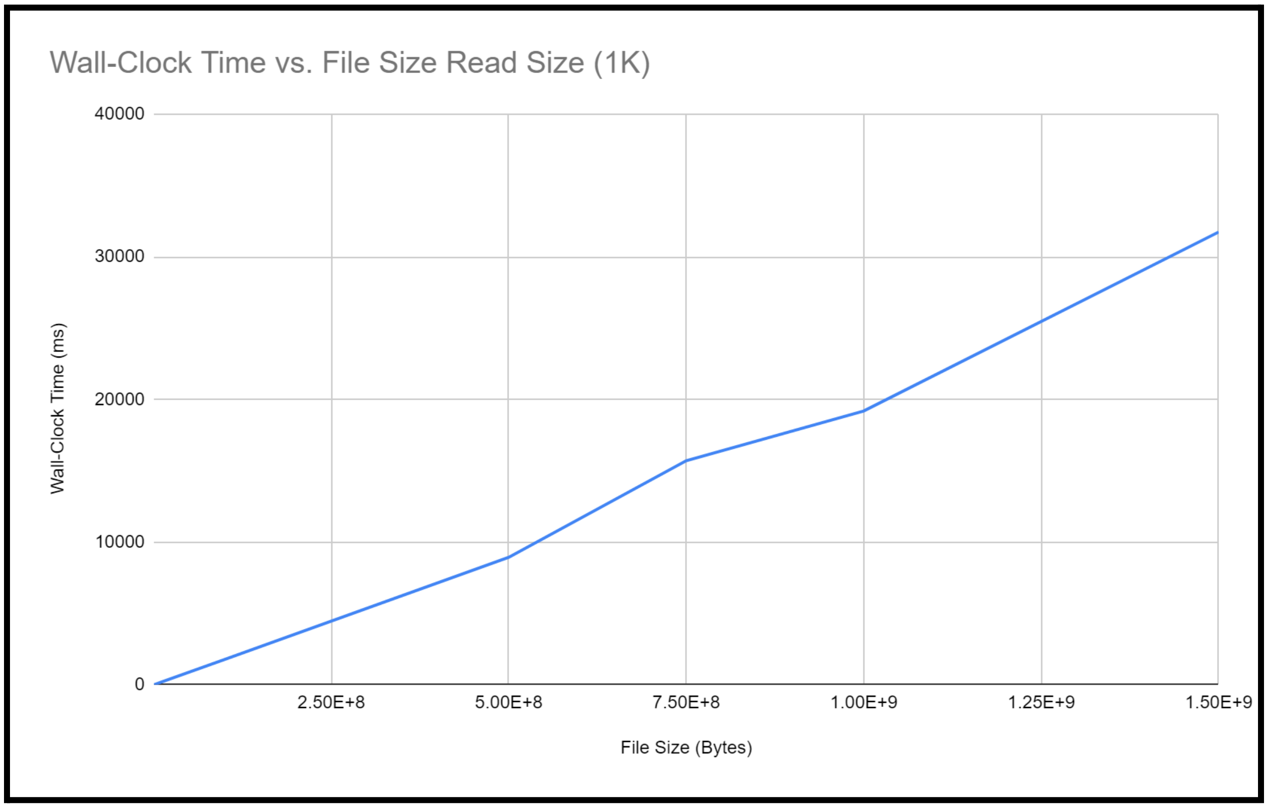
Read Size of 1:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	3
8000	3
10000000	183
100000000	1800
500000000	10572
750000000	14258
1000000000	20309
1500000000	32064



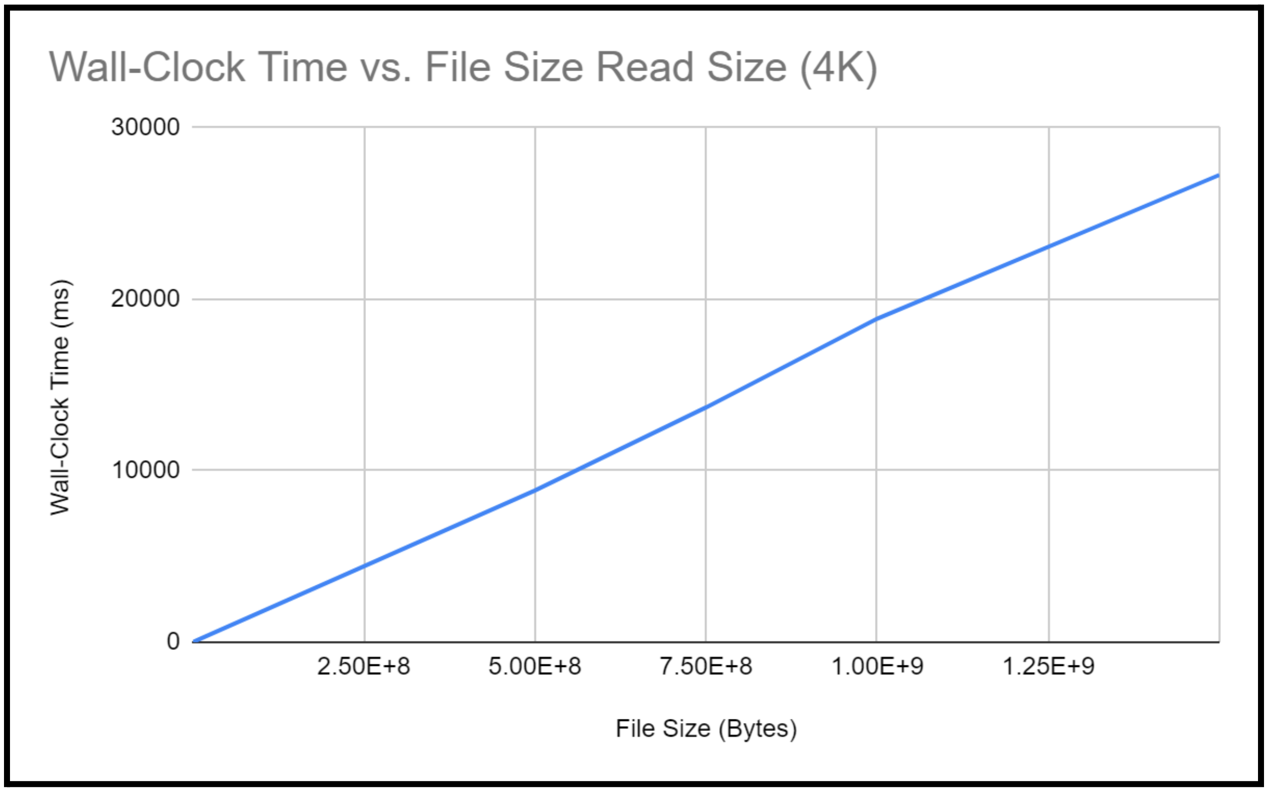
Read Size of 1K:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	183
100000000	1781
500000000	8938
750000000	15713
1000000000	19197
1500000000	31744



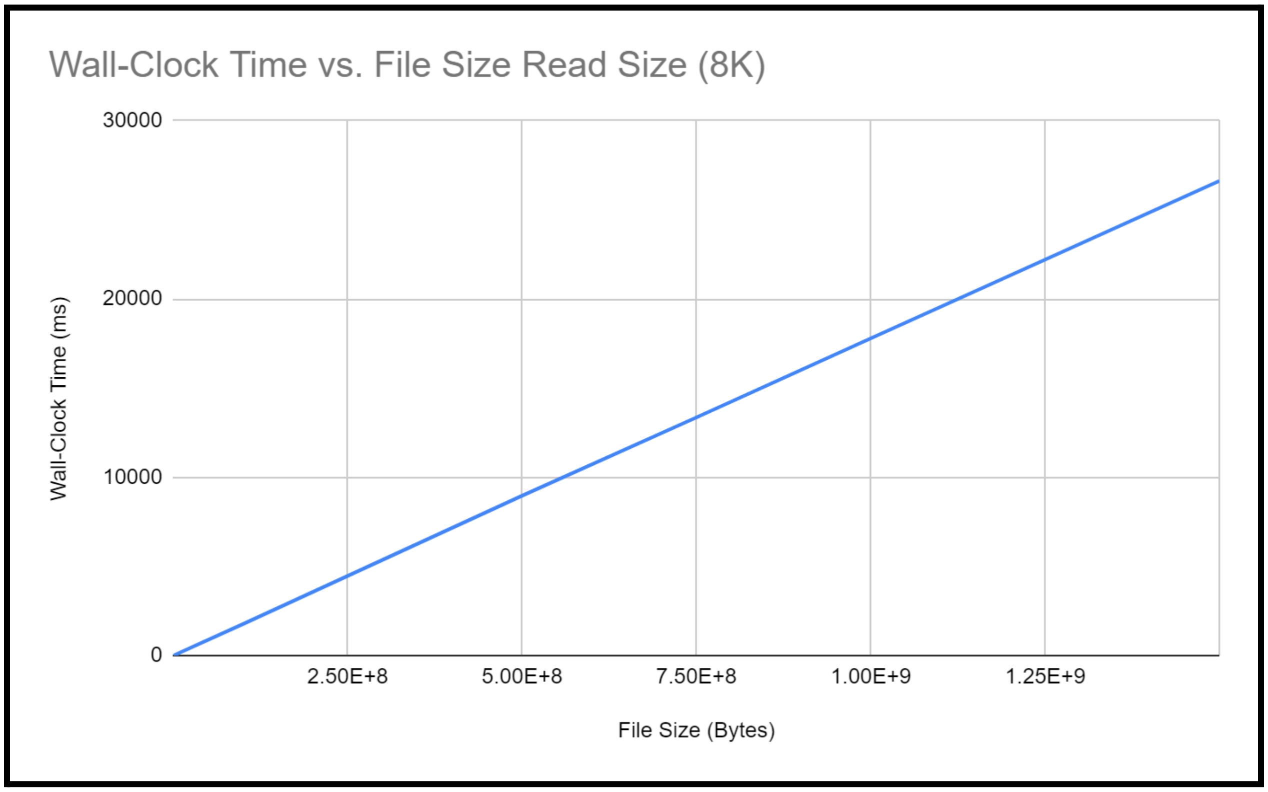
Read Size of 4K:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	180
100000000	1777
500000000	8850
750000000	13695
1000000000	18868
1500000000	27248



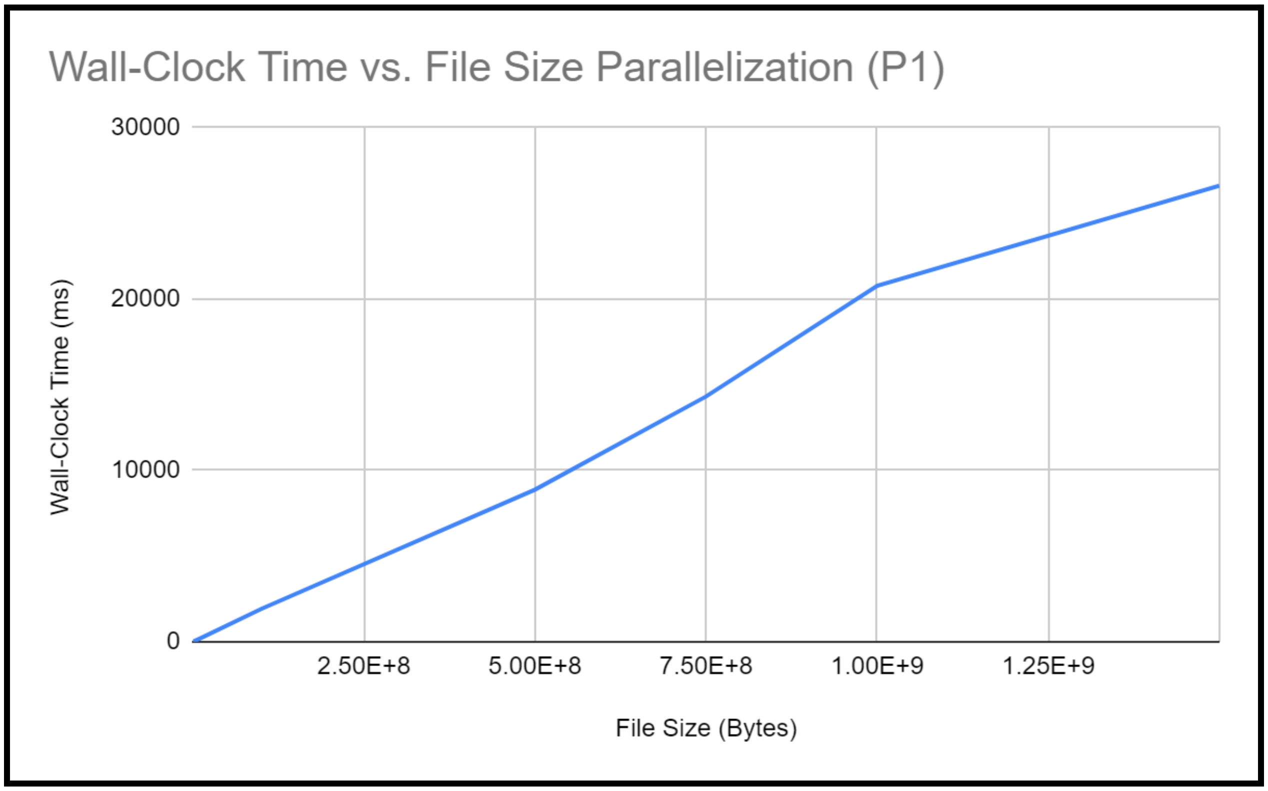
Read Size of 8K:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	180
100000000	1776
500000000	8964
750000000	13354
1000000000	17788
1500000000	26615



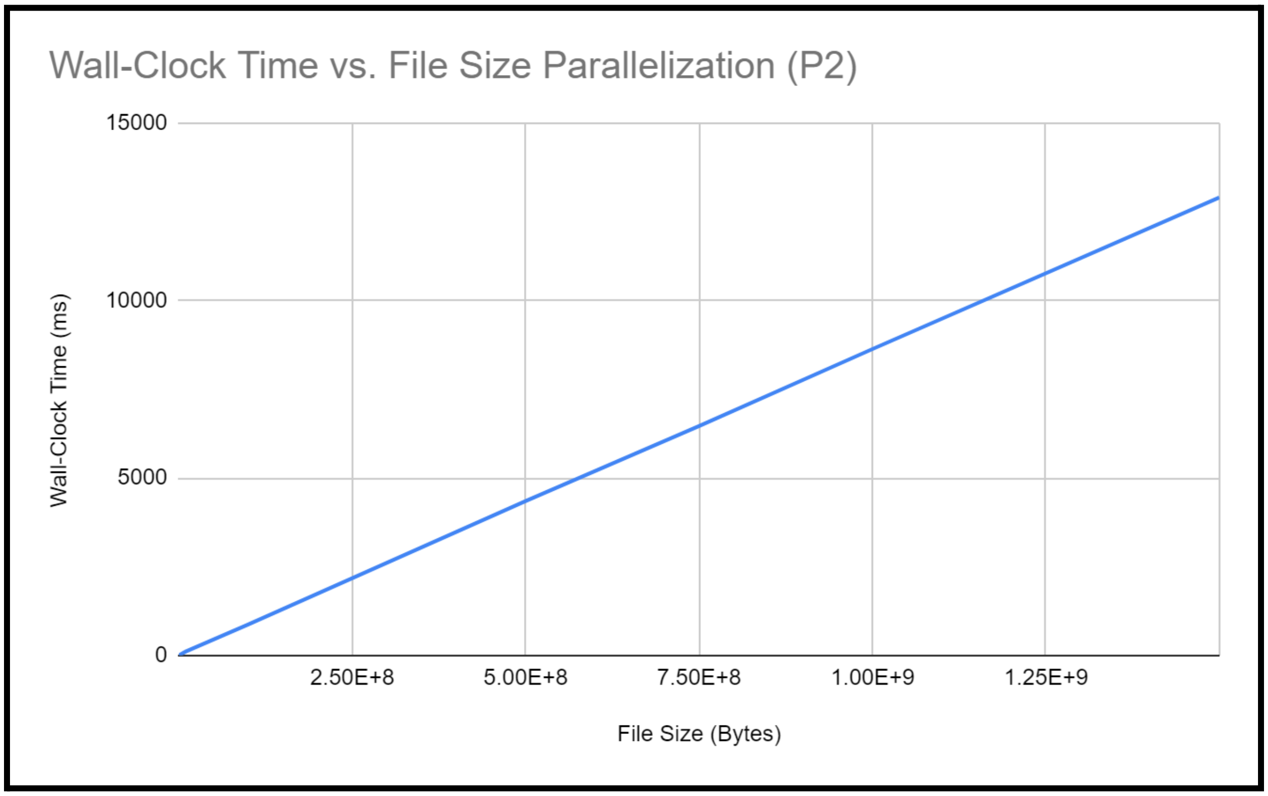
Parallelization P1:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	187
100000000	1919
500000000	8880
750000000	14317
1000000000	20769
1500000000	26609



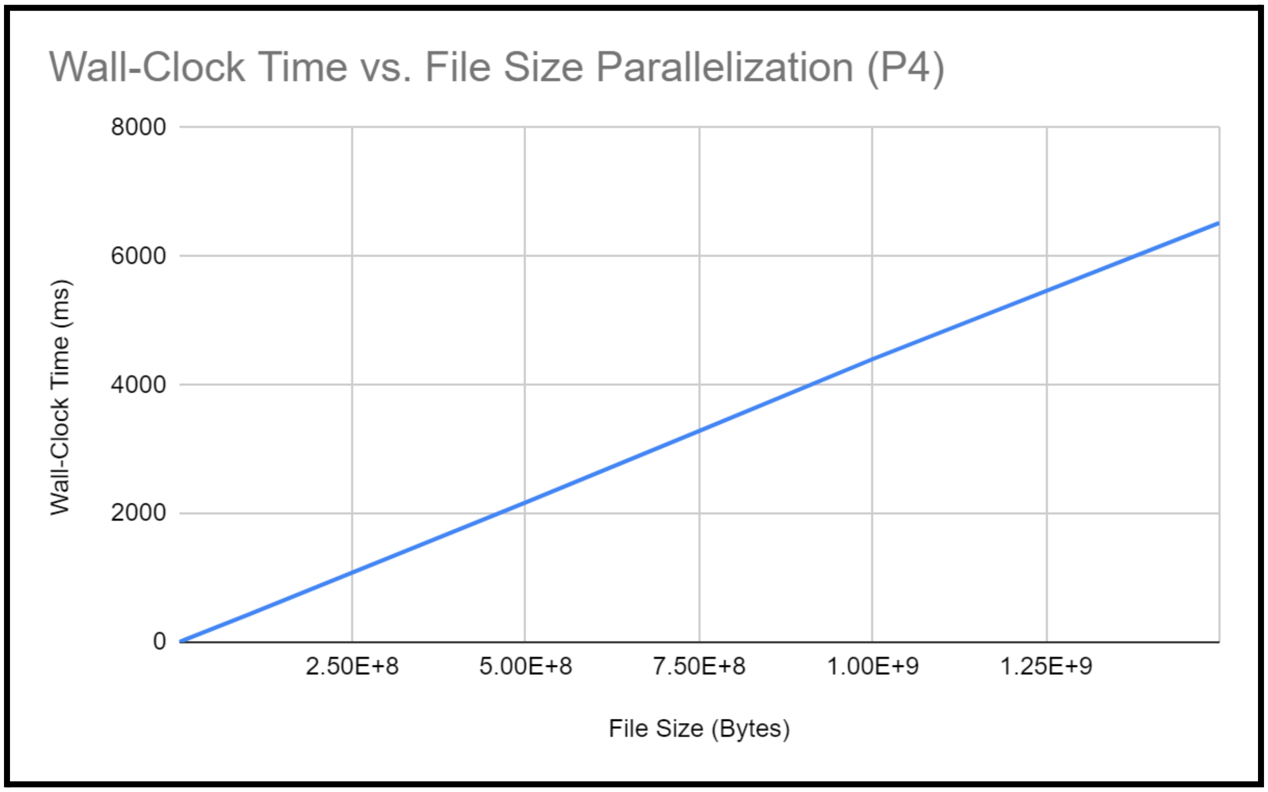
Parallelization P2:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	110
100000000	873
500000000	4350
750000000	6465
1000000000	8633
1500000000	12903



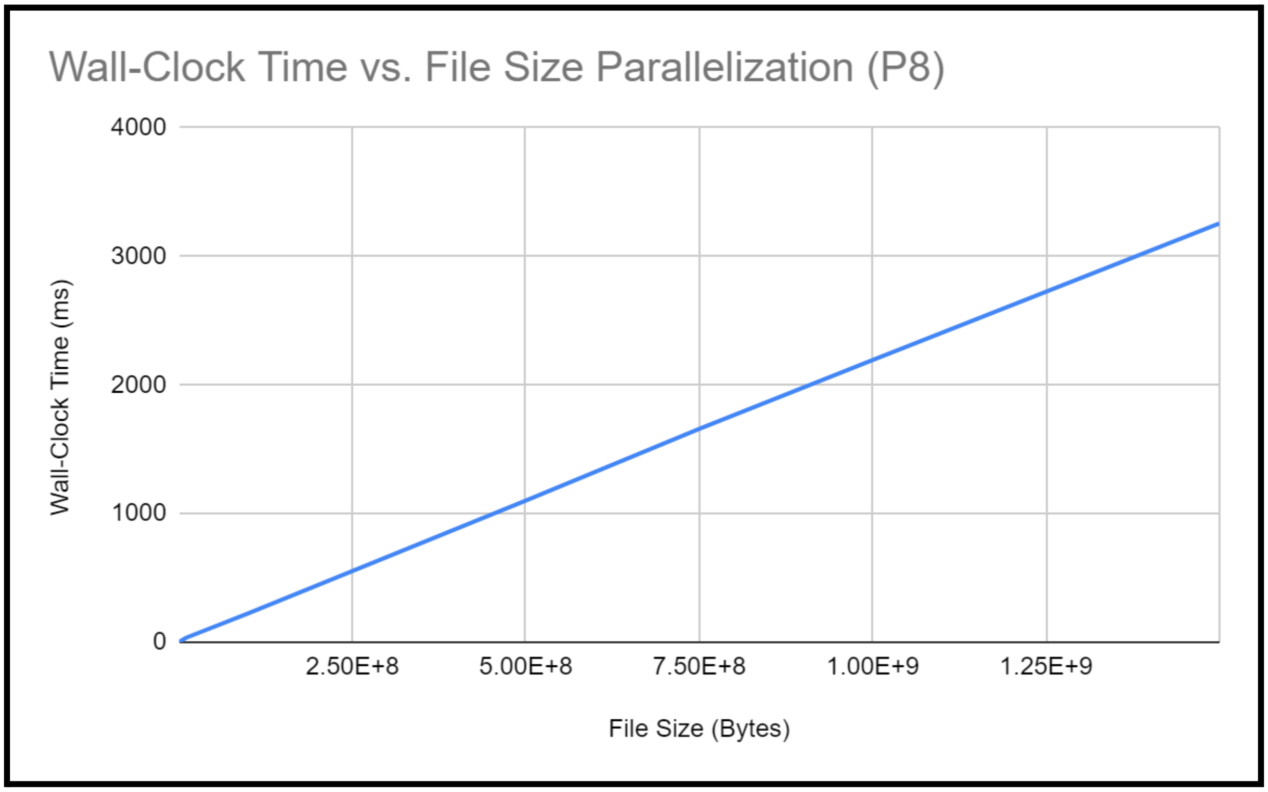
Parallelization P4:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	45
100000000	430
500000000	2173
750000000	3284
1000000000	4402
1500000000	6519



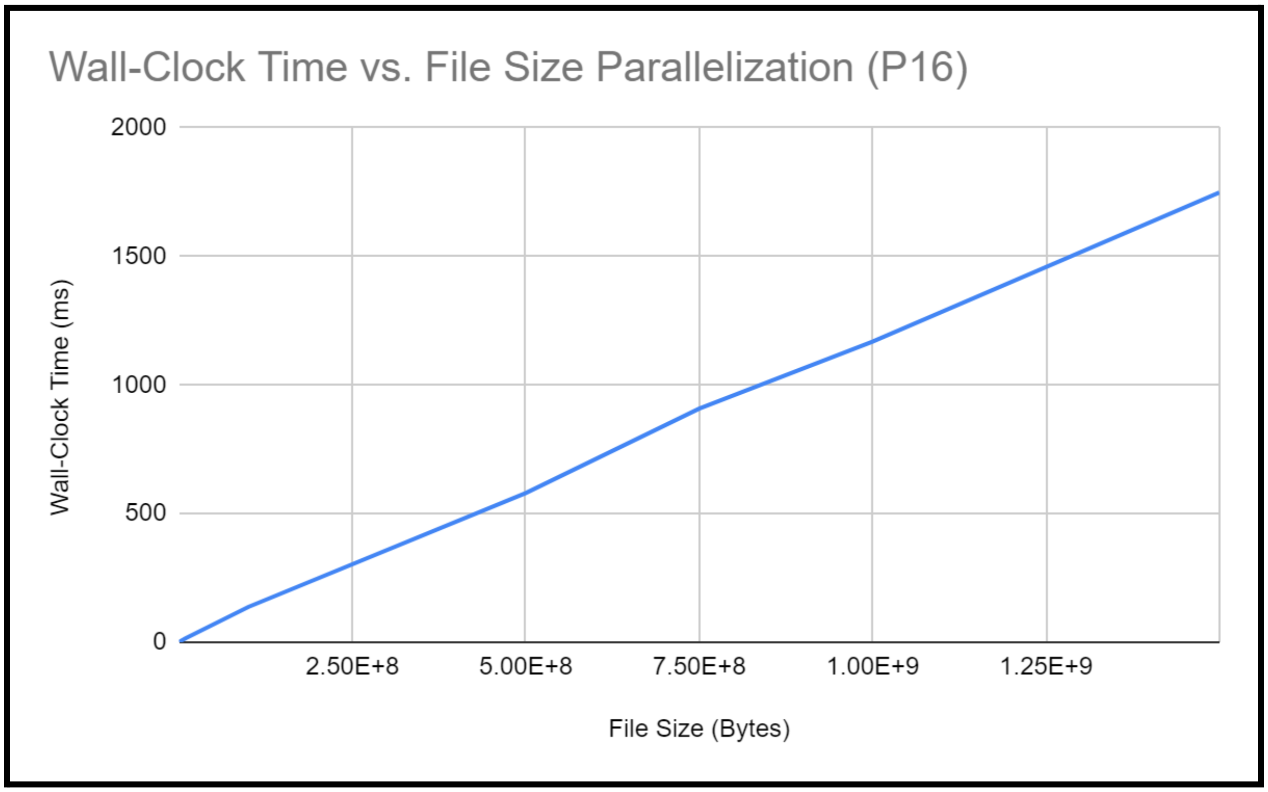
Parallelization P8:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	34
100000000	225
500000000	1101
750000000	1658
1000000000	2194
1500000000	3256



Parallelization P16:

File Size (Bytes)	Wall-Clock Time (ms)
1	1
1000	1
4000	1
8000	1
10000000	16
100000000	137
500000000	580
750000000	908
1000000000	1169
1500000000	1748



Analysis:

All Data:
https://docs.google.com/spreadsheets/d/1h5S36K4ix4YP5LOASmanRIIMHu8JNZ5E8L8phns-8Yg/edit?usp=sharing

For this analysis I utilized 10 files that I randomly generated with specific byte sizes. The file sizes included: 1 byte, 1KB, 4KB, 8KB, 10MB, 100MB, 1GB, and 1.5GB. I tested all of the file sizes on various different read methods.

The first read method simply memory mapped all of the file data at once and then processed the entire file in memory. The Read method takes chunks of different sizes and processes the chunks one at a time. I used chunks of 1 byte, 1KB, 4KB, and 8KB for the reading chunks. The final method of parallelization splits the problem into multiple processes. It divides the file size by the given number of processes and allots each process a specific portion of the file to process. Each process will map the given file chunk to memory and process that section. This will occur in the multiple parallel processes. I tested 1, 2, 4, 8, and 16 parallel processes.

Mapping to memory seems to consistently have a faster time compared to reading in chunks. Reading in chunks of 8000 bytes seems to be most comparable to mapping to memory. This may be because as the chunks get bigger the closer we get to mapping the entire file. Surprisingly, reading in chunks of 8000 bytes ended up being slightly faster than memory mapping as the time in milliseconds for reading 8000 bytes was 26615, and the time for memory mapping was 2702. The times for these processes tended to vary from every run. There were times when memory mapping was faster and there were times when reading in chunks of 8000 bytes was faster. The reason for this inconsistency was most likely the file size. I used file sizes up to 1.5GB, however, I believe in order to truly see what is the most efficient method of processing data, we must use much larger files of up to 10 or more GB. Downloading such large files became very difficult on the remote linux server, so I settled on 1.5GB as my greatest file size.

The time for parallelization with one process is very similar to the time of memory mapping. This is because parallelization maps the given file data to memory. In this case the entire file was given to one process so it was the same as memory mapping as done before. However, the greatest changes are observed when using 2 or more parallel processes. The times were just about cut in half every time we doubled the number of parallel processes. For example the time in milliseconds for 4 parallel processes, processing a file of 1.5 GB is 6519 ms, the time for 8 parallel processes is 3256 ms, and the time for 16 parallel processes is 1748 ms. These values are not exact, however, it is clear that there is a very significant decrease close to half of the previous time. This is clearly the fastest and most efficient method of data processing. Unlike the simple memory mapping of all the data or reading chunks of data—in this case there was a substantial decrease in processing time.

A question is if additional processes provide better performance even though the machines have a smaller number of cores:

In this testing environment, the results show that using multiple processes has a clear impact on the time it takes to process data. The machine I used for this testing has two cores and I ran tests using 4, 8, and 16 processes. Each of these tests resulted in half the time of the previous, so it seems in this case additional processes provide better performance even when the machine has fewer cores. However, this is only relevant to the processes we are testing and the number cores on the machine used. If we were doing more complex tasks than simply processing ascii characters in a file or if the machine used for testing had more cores, parallel processes may not have as much or an impact or it may even increase the time it takes for these tasks. We simply cannot assume so without having further testing. So, more testing would need to be done in order to say that overall, additional processes provide better performance regardless of the number of cores.