Parallelize File loading using Multi-Threading

Ashwin Kumar Udayakumar (au2177)
Harsh Bansal (hb2709)
Karthikeyan Shanmugam (ks6964)
Sathvika Bhagyalakshmi Mahesh (sb8913)

Abstract

In the era of big data and real-time analytics, efficient file loading becomes pivotal for computational efficiency, especially in multicore processor environments. We explore parallel processing techniques in optimizing a fixed-size chunk-based file-loading process. We propose and evaluate four methods: parallel chunk reading, parallel chunk reading with RAID 1 like mirroring, parallel chunk reading with memory mapping, parallel chunk loading based on chunk priorities. The experimental results, derived from a series of benchmarks reveal significant insights into the efficiency of each approach. Our findings suggest that the optimal file-loading technique is influenced by file and chunk sizes, with memory mapping showing marked performance advantages when the file is large. This study underscores the importance of tailored approaches to file I/O in leveraging multicore processor capabilities for improved computational throughput. Our research also compares sequential and parallel processing models for loading prioritized chunks. It highlights the substantial efficiency gains of parallel processing in managing varied priority levels, ensuring controlled, race-condition-free execution.

1. Introduction

With the advent of multicore processors, the potential for concurrent data processing has substantially increased, unlocking new avenues for performance optimization in software applications. One critical operation that stands to benefit from parallel execution is file I/O, an essential yet often bottlenecked domain in system performance. As data sets grow in size, the ability to rapidly read and process file data becomes a crucial determinant of overall system throughput. Traditional single-threaded file loading methods tend to fall short in achieving a desirable throughput, and this project aims to address this limitation by exploring a parallelized approach to file-loading.

The conventional approach of sequential file reading in small chunks, while straightforward, does not exploit the parallelism inherent in modern CPUs, leading to suboptimal utilization of available resources. The repetitive nature of the loading process opens up significant room for improvement through parallelism. We conducted a literature review on file loading methods and parallelized approaches to gain a better understanding of this area. Drawing insights from our research, we improved the sequential single-thread file loading process incrementally. We explore and

benchmark three methods of parallel file loading: parallel chunk reading, parallel chunk reading with file mirroring, and parallel chunk reading with memory mapping. These methods are evaluated in terms of execution time, speedup and efficiency in a multicore environment across various thread counts, file sizes and chunk sizes. As an extension to this study, we also explore priority-based loading and compare sequential and parallelized implementations of it.

The subsequent sections will detail the background literature from which we drew insights and ideas, elaborate on our proposed methods, describe the experimental setup, and thoroughly analyze the observed outcomes.

In the research regarding priority loading, we delve into the dynamics of file processing, focusing on speed up, efficiency, and memory usage while examining the role of parallelization. Our study involves a comprehensive comparison between single-threaded and multi-threaded, and between a number of threads in the case of multi-thread processing, execution across various priority ranges, threads, and a number of chunks.

2. Literature Survey

In [1], the modular components of the applications are designed in such a way that if both the source and destination support multi-threading, the source component partitions the data into multiple parts depending on the number of CPUs, while the destination component reads the data using multiple threads and joins it. The partitioning design allows multiple threads to access different parts of the file. However, multiple threads operating upon the same file will lead to resource contention, leading to increased number of context switches, ultimately leading to a decrease in performance. We aim to solve this problem of increased context switches using different methods.

One method that we use to decrease the number of context switches for multithreading is by replicating the RAID architecture [2], since it will lead to individual threads having access to their own file copy. Implementing a RAID like architecture also improves data redundancy. However, choosing the right type of RAID architecture to base upon is integral. Based on the Performance evaluation conducted in [3], we follow RAID 1 architecture, since it’s faster to implement on runtime, and serves the purpose of providing separate file copy for each thread.

In order to improve scalability of the parallel application, the application must scale properly in terms of performance with the problem set (file size). In [4], in order to deal with scalability problems of structured data, memory mapping technique is used. Using memory mapping also allows you to cache the frequently accessed parts.

However, one of the biggest problems with large files of any format would be the response time, when dealing with interactive applications. To deal with this problem, we borrow a priority based scheduling methodology [5] for processors. This type of non preemptive scheduling allows the

application to access the important chunks first, load other chunks later. Some applications are navigation maps, lists, etc.

In [6] Global Fixed-Priority Scheduling is used, the research seems to be centered around the implementation of these protocols, for task execution priorities on multiprocessor systems. In our study we also include parameters like, Chunk-Based Parallel Processing, Priority-Based Chunk Processing, Impact of Chunk Size and Number of Threads. The concept of chunks along with priority gives a broader view of performance based on synchronization wait times.

In [7] model, handles dynamic distribution of work and data, does not explicitly focus on priority-based task management, in our study file processing is taken care. We plan to observe that increasing the number of threads can sometimes increase execution time due to synchronization and bottleneck issues, particularly at higher priority levels not generally covered in parallel processing models. Also to analyze the impact of range of priorities on efficiency for given chunks.

3. Proposed Idea

Based on ideas gained from research and our understanding of multicore architectures and programming, we propose a multifaceted parallelization strategy that transforms file loading into a concurrent operation, amplifying throughput and minimizing execution time. All approaches stem from the traditional chunk-based file loading method. Chunk-based loading would enable multiple threads to work in parallel and also ensure our solutions are practical as it is a common practice to process ingested data in chunks, especially when files are large. Below we expand on each approach, explaining details in implementation.

3.1 Parallel Chunk Reading

Our first solution is the implementation of parallel chunk reading. This solution would be the logical first step in parallelizing the traditional file reading approach. It makes use of the idea of reading the file as a byte stream. The file is broken into chunks and multiple threads read these chunks into memory in parallel.

Implementation

The process begins by dividing the file into segments, each determined by a predefined chunk size. This is achieved through a `for` loop, where in each iteration, the loop reads a single chunk. We assign an identical number of chunks to each thread to ensure equal workload distribution. Before loading, we initialize an array of file pointers, all referencing the input file. The index of each pointer in this array corresponds to a specific thread number. Consequently, when the loop initiates, each thread employs its designated file pointer to access the same file.

During every iteration, a thread positions its file pointer to the start of the chunk it is tasked with (calculated as `chunk_index * CHUNK_SIZE`). It then reads a fixed size bytes equal to CHUNK_SIZE from the file, except possibly for its final chunk. The read bytes are stored in a pre-allocated memory space.

In order to verify the correctness of the implementation, we first checked if the number of bytes read by all the threads matched the size of the input file. We then output the bytes in memory into an output file and checked if the output and input files were identical.

3.2 Parallel Chunk Reading using File Mirroring

Our second solution draws inspiration from the RAID-1 [2] setup. It introduces a software-based simulation of RAID 1 mirroring. This approach creates multiple copies of the data (mirrors) and distributes the file reading operations across these copies. We hypothesized that this redundancy could be advantageous, allowing for simultaneous reads from different storage locations using multiple threads, potentially increasing the read speed while at the same time achieving fault tolerance.

Implementation
We employ a strategy akin to our previous approach, beginning with dividing the file into chunks, each processed in a for loop iteration. Consistent with our aim of workload balance, we assign an equal number of chunks to each thread.

In the event that a file copy is not present for a thread, a dynamic reassignment mechanism is implemented. If a file pointer is found to be NULL (indicating an unavailable file), then the system iterates through the available file copies to find the next suitable copy. This ensures that threads are seamlessly redirected to alternative file copies, mitigating the impact of potential unavailability and enhancing the robustness of the system while maintaining an even distribution of threads amongst the available copies. In order to save time by not reopening unavailable files copies an array is used to maintain the status of the copies when they are first encountered.

The operations in each iteration are almost identical to the previous approach. The only difference is that each thread’s file pointer points to the specific copy it is assigned. The read bytes are stored in a pre-allocated memory space.

In order to verify the correctness of the implementation, checks were performed in the similar manner as the previous approach.

3.3 Parallel Chunk Reading using Memory Mapping

Our third solution involves memory mapping, a technique that directly maps a file's contents into the process's virtual memory space. This mapping provides threads direct access to the file and

enables them to read different portions efficiently and simultaneously without the overhead of traditional file I/O system calls. Given this mechanism, for this solution, we do not need to write data to process memory. However, we still retain chunk processing as we want to maintain the same chunk-level processing capabilities across our approaches.

Implementation
The initial portion of the implementation where the number of chunks are calculated remains the same. Subsequently, the `mmap` system call is used where the file is opened in read-only mode, and its data is mapped into the process's memory space. Unlike previous implementations, we do not allocate any memory for the data nor do we create any file pointers.

In each iteration of the for loop, similar to before, threads compute the chunk start pointers and chunk size. At this point, the chunk data is directly accessible and hence, reading and storing of the data is not necessary.

In order to verify the correctness of the implementation, we checked if the number of bytes read by all the threads matched the size of the input file.

Benchmarks for Comparison

The comparison criteria for these methodologies are:

- **Execution Time**: The primary measure is the time taken from the initiation of the file loading process to its completion. It is further used to make the comparison between the models in terms of speed up and efficiency by varying different parameters.

- **CPU & Memory Analysis**: Understanding the CPU and memory performance matrix like number of context switches, cache misses and impact of memory size shows large insight in the program execution.

These metrics were selected to not only determine the fastest method but also to understand the trade-offs and practical implications of each approach when deployed in real-world scenarios.

3.4 Loading of Priority Chunks Sequentially and Parallelly by Processing Chunks of same priorities

In our research, we explore the concept of Loading, analogous to progressively loading image rows, where the highest priority (first row) is loaded first, followed by subsequent rows in descending order of priority. This method contrasts with the conventional approach of loading the entire image simultaneously. Applying this to file processing, we divide the file into chunks, grouping them into batches based on their priority levels. Our study conducts a comparative analysis of sequential and parallel loading of these prioritized chunks. In the sequential processing model, the chunks are processed using a single thread based on priority . In the parallel

processing model, a batch is processed using multiple threads, and the next priority batch is not initiated until the current one is completed. This approach ensures adherence to priority levels and prevents race conditions in accessing subsequent priority chunks, effectively maintaining the integrity of the Loading concept in a parallel processing environment.

Implementation

Our parallel file processing system efficiently handles large files by dividing them into manageable Chunks.Each Chunk consists of a unique identifier (chunkID), a data buffer (char *buffer), data size (bytesRead), and a priority level (priority). Chunks are grouped into Batch structures based on their priorities, ensuring orderly processing. The entire file is segmented into a given number of chunks, with the size of each chunk dynamically calculated based on the total file size. Each batch maintains an array of Chunk pointers (Chunk **chunks) and a count of chunks (numChucks) per batch.

For processing, we iterate through batches sequentially, and within each batch, we parallelize the execution of chunks using multi-threading, significantly enhancing the processing speed by concurrently handling chunks of the same priority. The groupChunksByPriority function organizes chunks into batches, while generate_priorities assigns priorities based on chunk position. This architecture optimizes performance, maintaining order while leveraging parallel computing benefits.

Benchmarks for Comparison

- **Execution Time**: Time taken starting from processing each chunk with the last chunk with last priority is completed.
- **Number of Threads**: Varying the number of threads to understand how work among the batches with the same priority is divided.
- **Number of chunks**: Study of how varying the number of chunks to process with a particular priority would affect performance.
- **Range of Priorities**: A comparative analysis of how the range of priorities assigned to chunks would affect execution time in case of multi-thread processing.

Justification of Choices

The rationale behind the proposed parallelization strategies stems from an understanding of multicore processors' architecture and the nature of file I/O bottlenecks. The chunk reading technique aligns with the concept of data parallelism, while file mirroring builds on the idea of I/O parallelism and fault tolerance. Memory mapping is chosen for its potential to minimize I/O overhead and capitalize on memory access speed. The benchmarking parameters are chosen to ensure a comprehensive evaluation of performance across different system states and workloads.

4. Experimental Setup

For carrying the below experiments, we used 2 different setups.

Setup A:

NYU CIMS Crunchy-1 Cluster:
CPU: Four AMD Opteron 6272 (2.1 GHz) (64 cores)
Memory: 256GB
OS: CentOS 7 (Linux)

(Cluster is shared among other users)

Setup B:

Custom Virtual Machine hosted on Windows
CPU: Intel Core i7 8650U (1.9 GHz) (2/4/8 cores)
Memory: 2/4/8GB
OS: Ubuntu 23.10 (Linux)

For the first three experiments, we also use a script to generate a random file of the desired size in GB. We generate a random file by using a buffer of size 1024 and filling it with random characters from A to Z. Once the buffer is full, it is dumped into the file, and processed until the file reaches the desired size.

For the last experiment, we use a C code that generates a file with the file size mentioned as required in CSV format, filled with random numbers. It initializes a random number generator, then writes numbers to the file. Each line in the file contains 1024 numbers. The file size is monitored to ensure it doesn't exceed the file size given. Once the desired size is achieved, the file is closed.

Time calculation

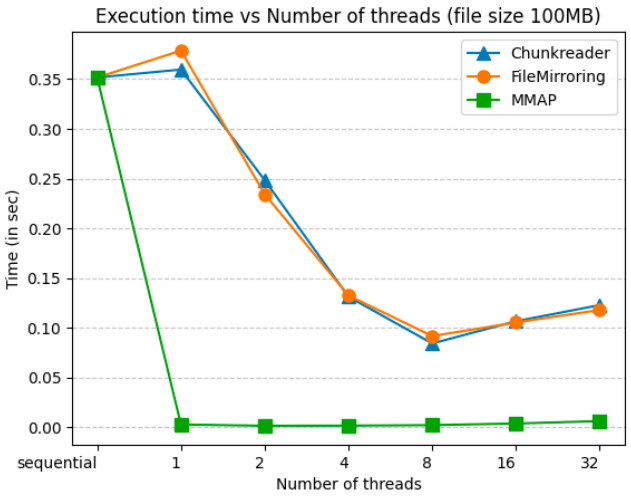
In order to calculate the execution time, we start measuring the wall time using omp_get_wtime() once we allocate the memory for file data. Once the parallel for-loop finishes executing, we once again measure the wall time to measure the end time. Execution time is then calculated using end_wall_time - start_wall_time.

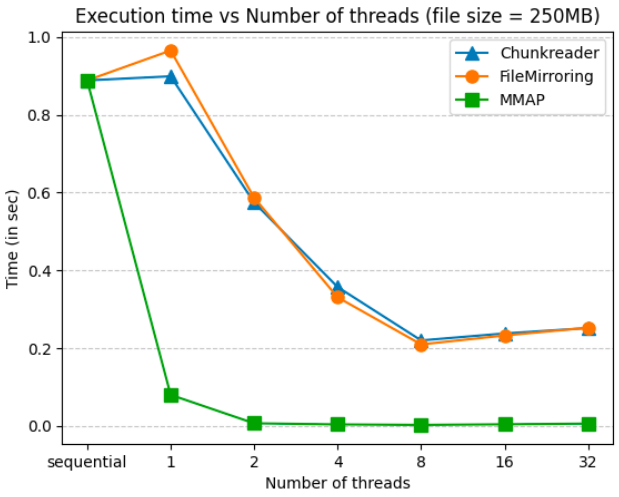
5. Results & Analysis

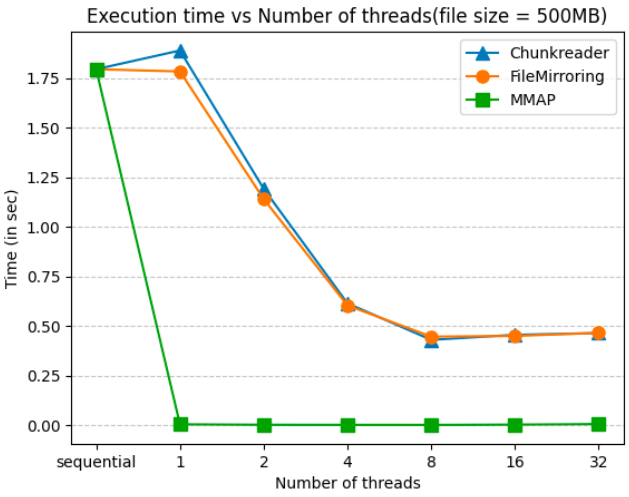
In this section we would go over the results obtained and draw conclusions from them.
Note: Unless otherwise mentioned, the number of copies used for FileMirroring is 4.

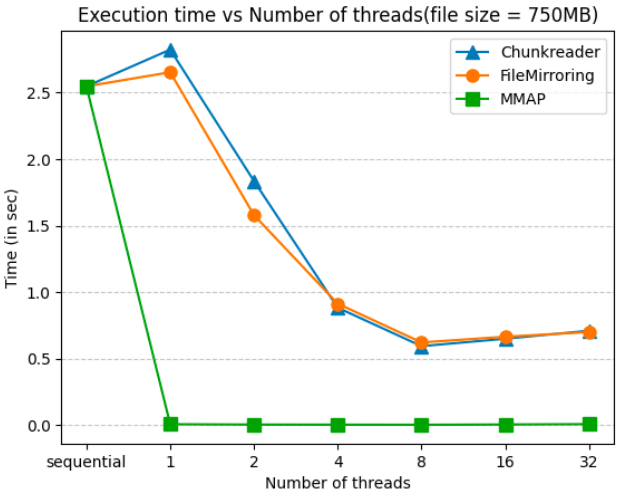
5.1 Execution time vs Number of Threads

We ran all three models for varying file sizes and different number of threads ranging from 1 to 32. The file size varies from 100MB to 750MB. Chunk size is 1024 bytes. It was tested in Setup A.









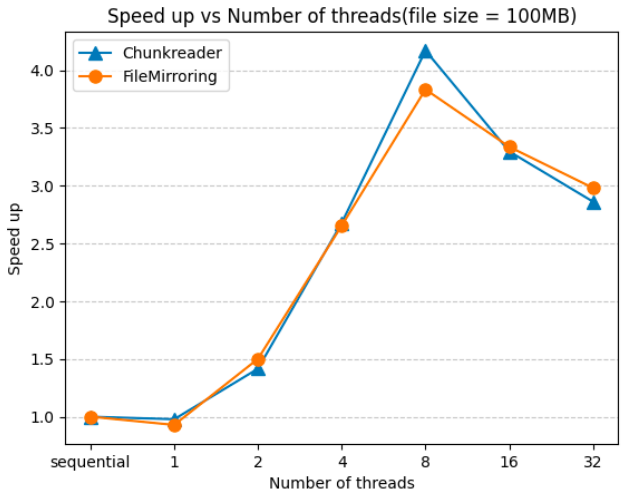
In both the Chunkreader and FileMirroring models, the execution time initially increases when run with a single thread, indicating the impact that parallel overhead has over execution time as

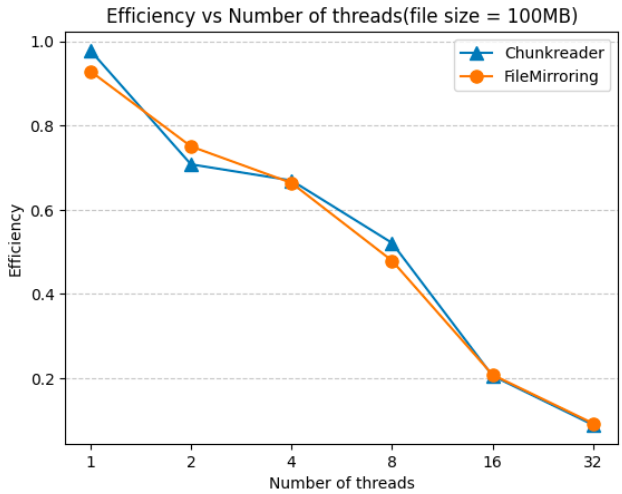
compared to the sequential version. However, with an increase in the number of threads, the execution time decreases, reaching a minimum when there are 8 threads. This reduction is attributed to the compensatory gain in performance from parallelizing the file reading process. Beyond 8 threads, a slight increase in execution time is observed, likely due to the growing overhead associated with parallelization, although this effect diminishes in larger file sizes.

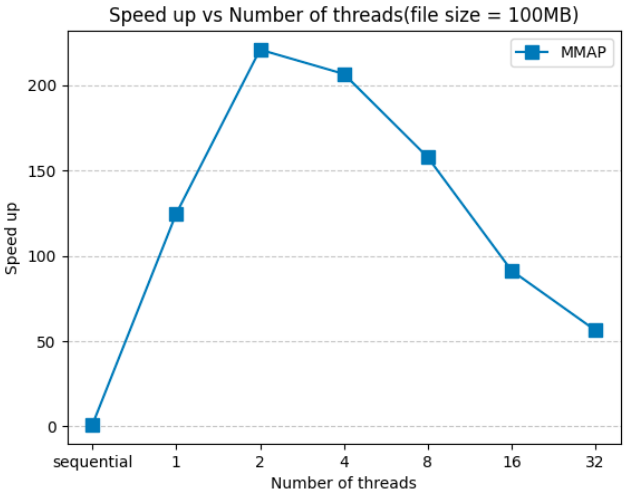
We find that the execution time for model using mmap is significantly smaller than the other two models. This can be explained by the fact that in mmap the file data is directly mapper to the process's address space without reading into the buffer reducing the read operation. As we increase the number of threads the execution time increases slightly due to the inrceased overhead generated by threads.

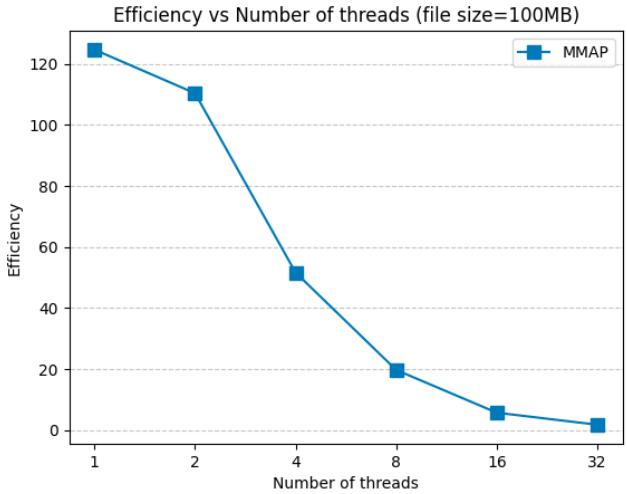
5.2 Speed up and efficiency vs number of threads

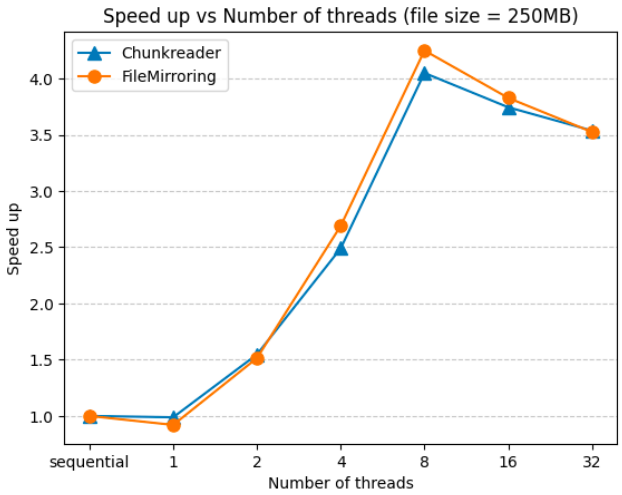
We ran all three models for varying number of threads ranging from 1 to 32. The file size varies from 100MB to 750MB. Chunk size is 1024 bytes. It was tested in Setup A.

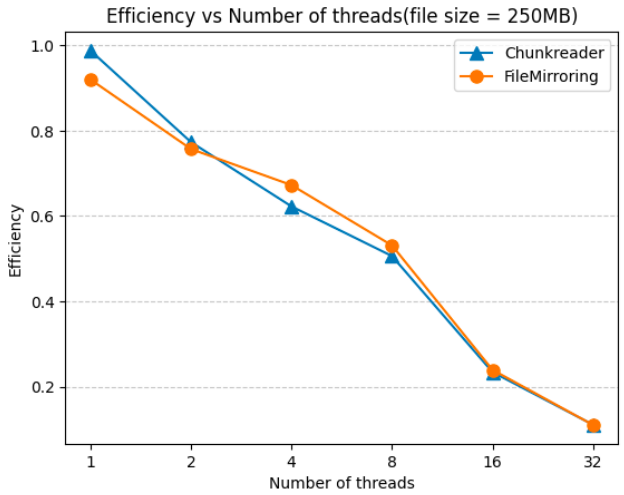


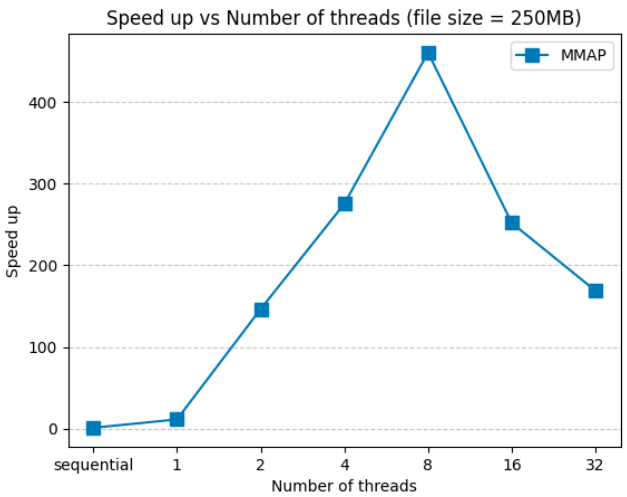


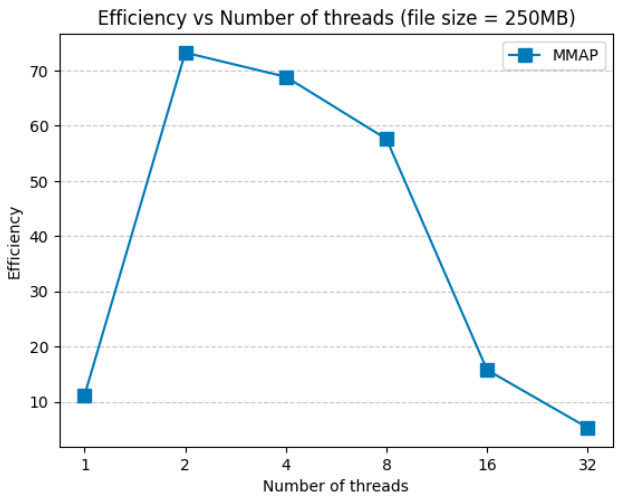


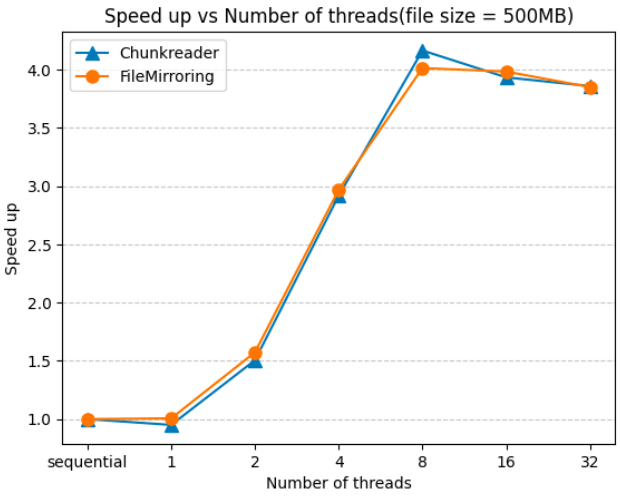


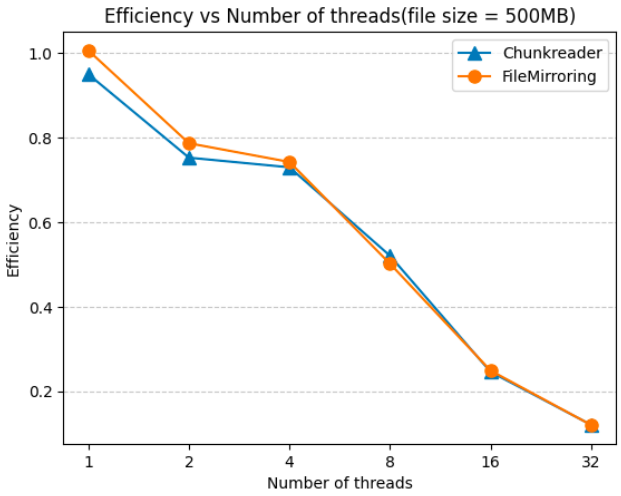

Speed up vs Number of threads (file size = 250MB)
Chunkreader
FileMirroring
Speed up
Number of threads
sequential
1
2
4
8
16
32

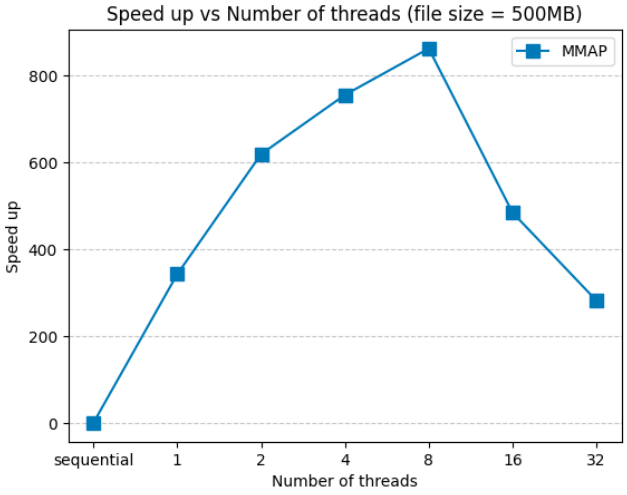

Efficiency vs Number of threads(file size = 250MB)
Chunkreader
FileMirroring
Efficiency
Number of threads

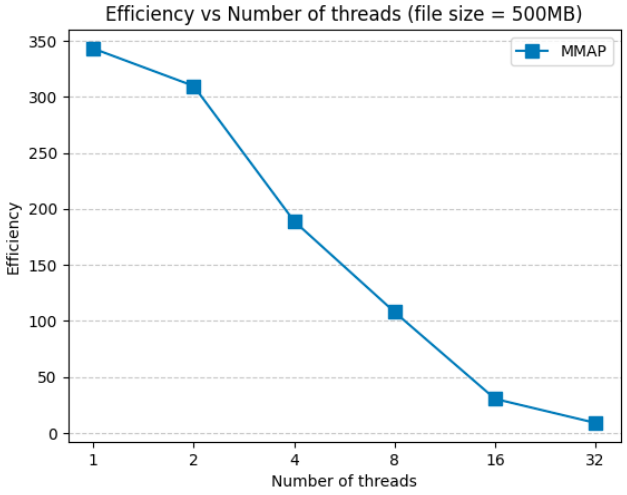

Speed up vs Number of threads (file size = 250MB)
MMAP
Speed up
Number of threads
sequential

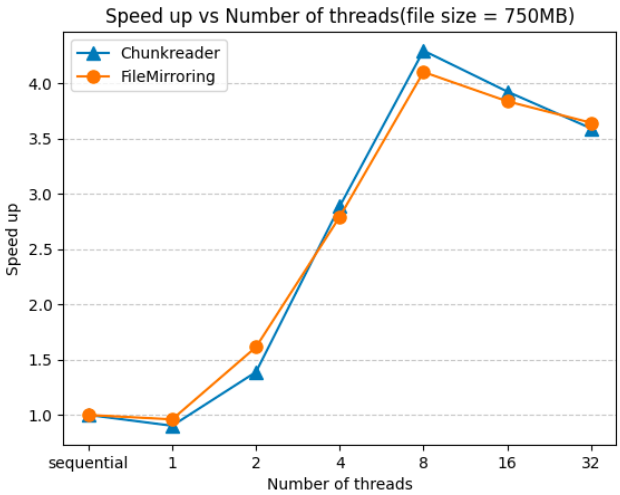

Efficiency vs Number of threads (file size = 250MB)
MMAP
Efficiency
Number of threads

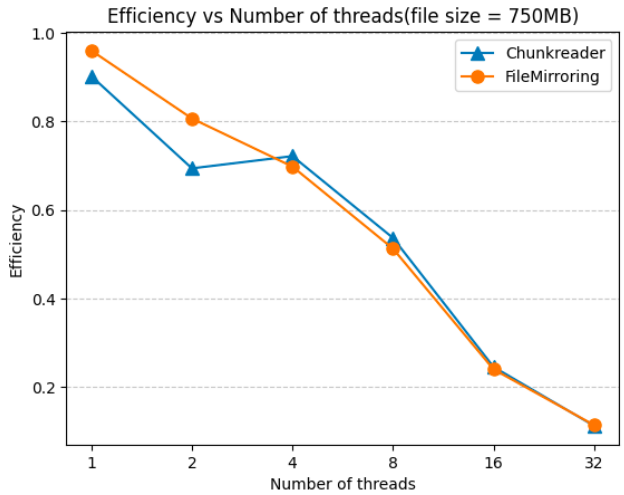

Speed up vs Number of threads(file size = 500MB)
Chunkreader
FileMirroring
Speed up
Number of threads
sequential

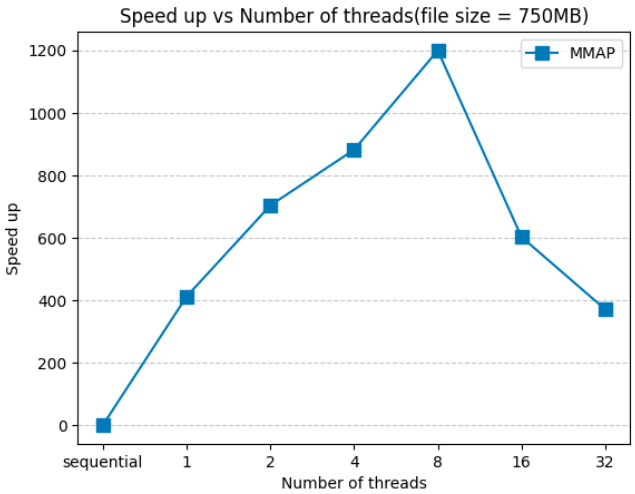

Efficiency vs Number of threads(file size = 500MB)
Chunkreader
FileMirroring
Efficiency
Number of threads

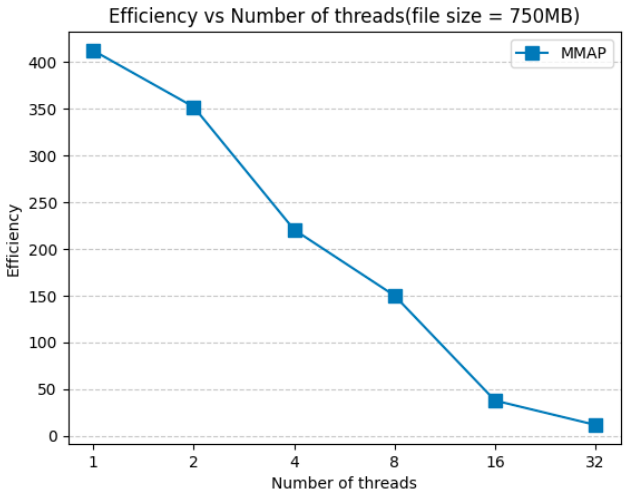

Speed up vs Number of threads (file size = 500MB)
MMAP
Speed up
0
200
400
600
800
sequential
1
2
4
8
16
32
Number of threads


Efficiency vs Number of threads (file size = 500MB)
MMAP
Efficiency
0
50
100
150
200
250
300
350
1
2
4
8
16
32
Number of threads


Speed up vs Number of threads(file size = 750MB)
Chunkreader
FileMirroring
Speed up
1.0
1.5
2.0
2.5
3.0
3.5
4.0
sequential
1
2
4
8
16
32
Number of threads


Efficiency vs Number of threads(file size = 750MB)
Chunkreader
FileMirroring
Efficiency
0.2
0.4
0.6
0.8
1.0
1
2
4
8
16
32
Number of threads


Speed up vs Number of threads(file size = 750MB)
MMAP
Speed up
0
200
400
600
800
1000
1200
sequential
1
2
4
8
16
32
Number of threads

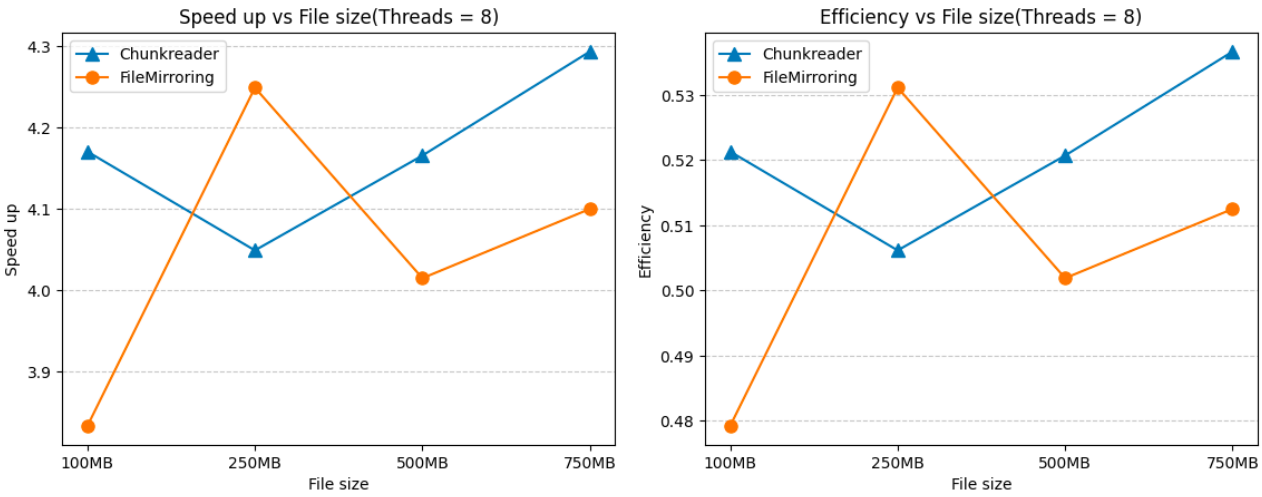

Efficiency vs Number of threads(file size = 750MB)
MMAP
Efficiency
0
50
100
150
200
250
300
350
400
1
2
4
8
16
32
Number of threads

For the speed up we find the trend conforming to the result that we expected. Aside from the slight dip when only one thread is running in parallel version, we find that it continues to increase as the number of threads increases, reaching a maximum when a number of threads = 8. After that, there is a drop in the speed-up which can be attributed to parallel overheads dominating the computational gain.

For the efficiency, we find that as the number of threads increases, the efficiency decreases consistently, indicating that the gain obtained by adding more threads is less than the overhead generated due to higher parallel threads running.

5.3 Speed up and efficiency vs File size

We run the experiment on setup A.
Number of threads were fixed to 8.



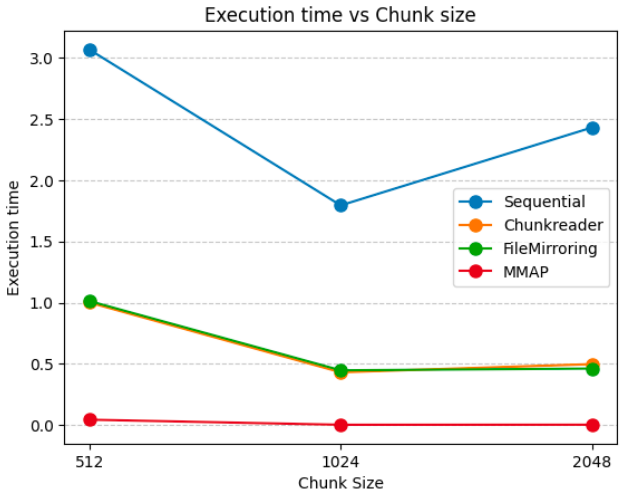
We see that for a particular thread as we increase the problem size, the efficiency of the program increases. Efficiency measures how effectively the resources are being utilized in parallel versions. The reason behind this flow is that as the file size increases each thread generates a more substantial gain in the computational time as compared to smaller file sizes. This compensates for the higher overhead leading to a better speed up as well as better efficiency.

The FileMirroring model exhibits a generally lower speedup and efficiency compared to the Chunkreader model. This difference can be attributed to a couple of factors. In FileMirroring, threads need to open files stored at different locations, while the Chunkreader threads only open a file stored at a fixed location. This distinction introduces cache considerations, reducing the time taken to open files by the Chunkreader threads. Additionally, the FileMirroring model involves extra checks for missing or corrupted copies to provide fault tolerance, contributing to increased

processing time. The observed results align with our expectations, although an unexpected spike in the graph for a file size of 250MB may be influenced by caching effects.

5.4 Execution time vs Chunk size

We perform this experiment on setup A.
We calculated the execution time by varying chunk sizes.

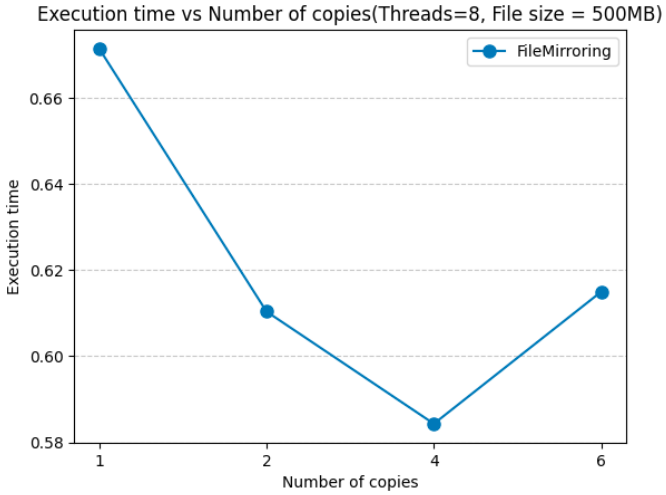


In our analysis, we focused on a specific configuration with 8 threads and a file size of 500MB, as these settings yielded optimal performance for our models. The observed trend aligns with expectations.

Considering our system's L1-D cache size of 16KB, we conducted experiments by varying the chunk size (512, 1024, and 2048). As the chunk size increased, we noted a corresponding decrease in the number of *L1-dcache-load-miss* events from 69,503,281 (4.24% of total hits) to 62,672,945 (3.91% of total hits). However, this trend reversed when the chunk size further increased to 2048, resulting in 63,969,552 *L1-dcache-load-miss events* (4.02% of total hits). This change in performance due to the number of cache misses is depicted in the chart. A higher number of cache misses indicates that data is now stored in the L2 cache, which takes more time, consequently impacting the overall execution time.

5.5 Execution time vs number of copies

For the FileMirroring model, we calculated the impact the number of copies have on the execution time.



In our analysis, we observe a noteworthy trend in the FileMirroring model concerning the number of copies. The execution time decreases as the number of copies increases, reaching a minimum with 4 copies, but then rises again with 6 copies.

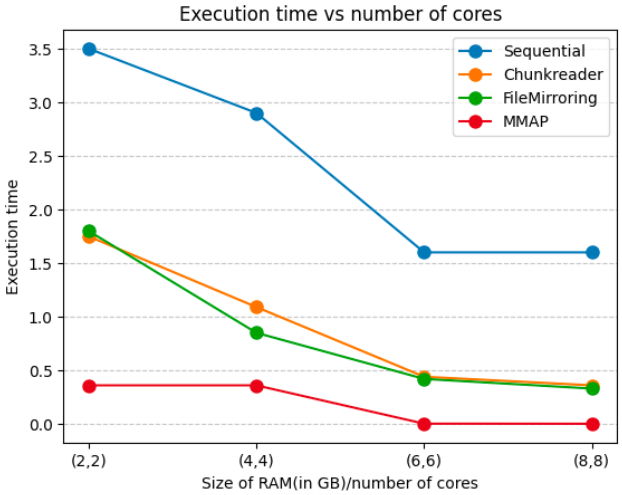
Initially, the reduction in execution time can be attributed to an optimized distribution of threads when the number of copies is increased. This leads to a more efficient operation as fewer threads are contending for access to the same copy, resulting in faster execution. However, when the number of threads is further increased to 6, we witness an increase in execution time.

Several factors contribute to this phenomenon. Firstly, the distribution of threads becomes uneven among the copies, with some copies having 2 threads while others operate with a single thread, creating an imbalance that impacts execution time. Additionally, as the number of copies continues to increase, the time required to check and open the file for each thread also increases, further contributing to the rise in execution time in this scenario.

5.6 CPU & Memory Analysis

Execution time vs Number of Cores/Memory

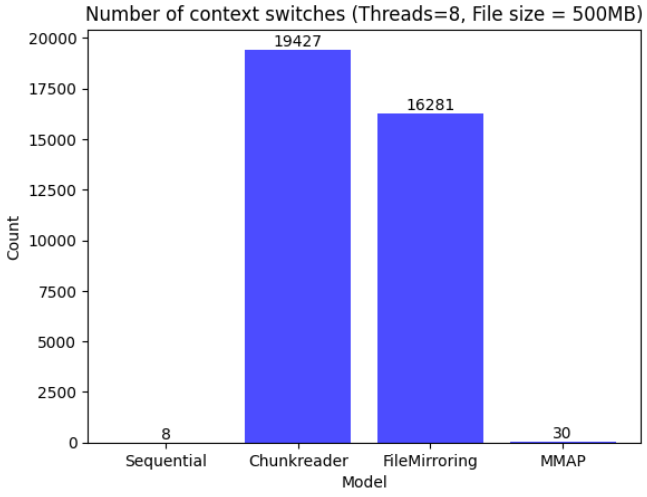
We performed this experiment on setup B.
It was performed on file size 500MB with 8 threads.



The chart above illustrates a notable influence of augmenting the number of cores or enlarging the RAM size on the program's execution time. As these factors rise, a corresponding decrease in execution time is evident. This outcome stems from the ability of an increased number of cores to facilitate parallel execution of multiple threads simultaneously. Additionally, expanding the RAM size contributes to a reduction in page swaps, subsequently lowering the overall time taken by the program.

Impact of Context Switches

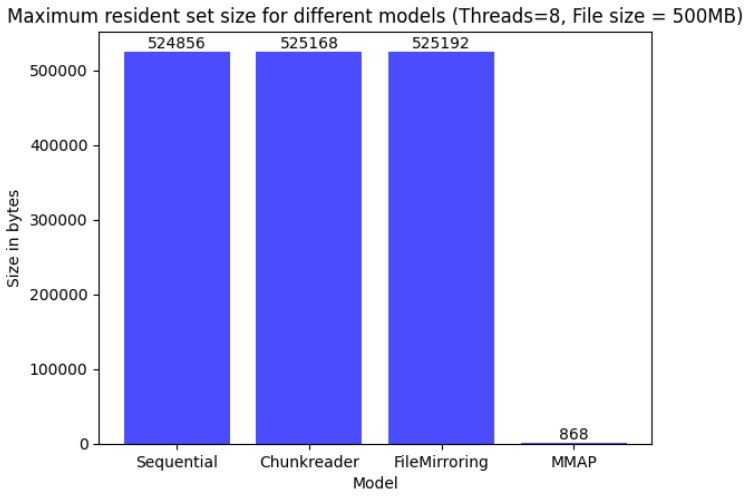
This experiment was conducted on setup A.



Here we observe that the number of context switches for FileMirroring is only slightly less than the Chunkreader. This is because in Chunkreader, all the threads point to the same file location, leading to more context switches between them. Whereas in FileMirroring, the threads operating on different files lead to a lesser contention amongst them for the file, leading to a lesser context switch. For mmap the memory is directly mapped to the process's virtual space so whenever the new chunk is required it directly can access it. It does not have to go through MMU to load that chunk in the memory which leads to very few context switching.

Impact on Resident Set Size (RSS)

This experiment was conducted on setup A

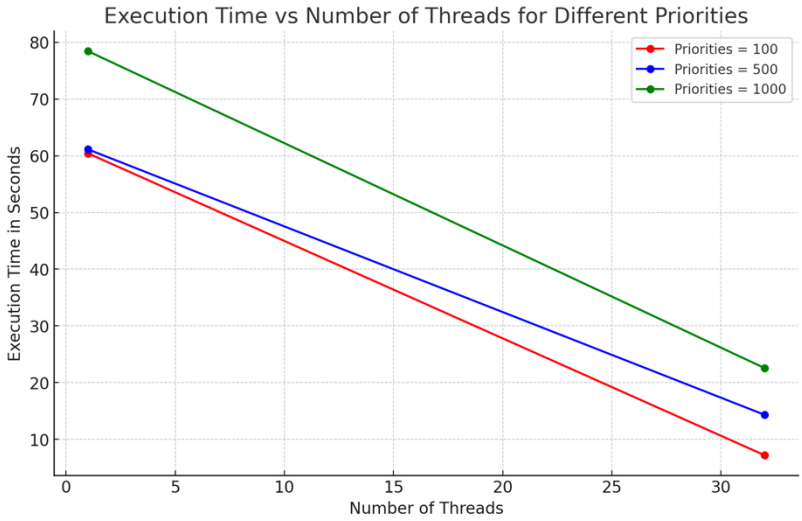


For the sequential, Chunkreader and FileMirroring model we find that they have nearly identical RSS which is because they load the whole file into the memory irrespective of the required chunk. The resident set size (RSS) for mmap is low. This is because mmap directly maps the section of the file to the process's user space, and only the required sections are loaded on demand.

5.7 Analysis of Priority Chunk Loading

Execution Time vs Number of Threads

Chunks =100000, File size = 1.3GB



The results depicted in the graph provide valuable insights into how execution time varies between single-threaded and multi-threaded processing, particularly under the conditions of a fixed file size (1.3GB) and a set number of chunks (100,000).

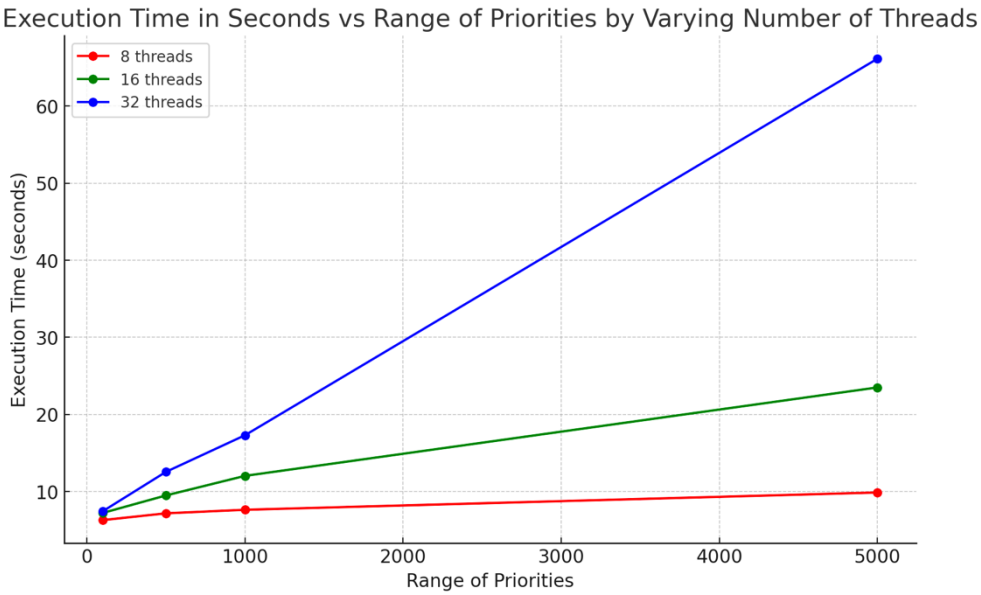
In the single-threaded scenario, where the number of threads is 1, we observe significantly higher execution times across all priority ranges. This is expected as single-threaded processing handles tasks sequentially, leading to longer completion times. In contrast, multi-threading demonstrates a substantial decrease in execution time. This decrease is consistent across all priority levels but varies in magnitude depending on the priority range. Multi-threading allows concurrent processing of multiple chunks of the file, leading to more efficient utilization of computational resources and faster overall execution.

For lower priority ranges the reduction in execution time is approximately 8 times when compared to the single-threaded approach. This significant decrease can be attributed to the effective distribution and parallel processing of tasks among multiple threads.

For higher priority ranges, the reduction in execution time is less pronounced but still notable, at about 4 times The diminished effect at higher priority ranges could be due to increased task complexity or other factors that limit the benefits of parallel processing.

Execution Time vs Range of Priorities

Chunks = 100000, File size = 1.3GB



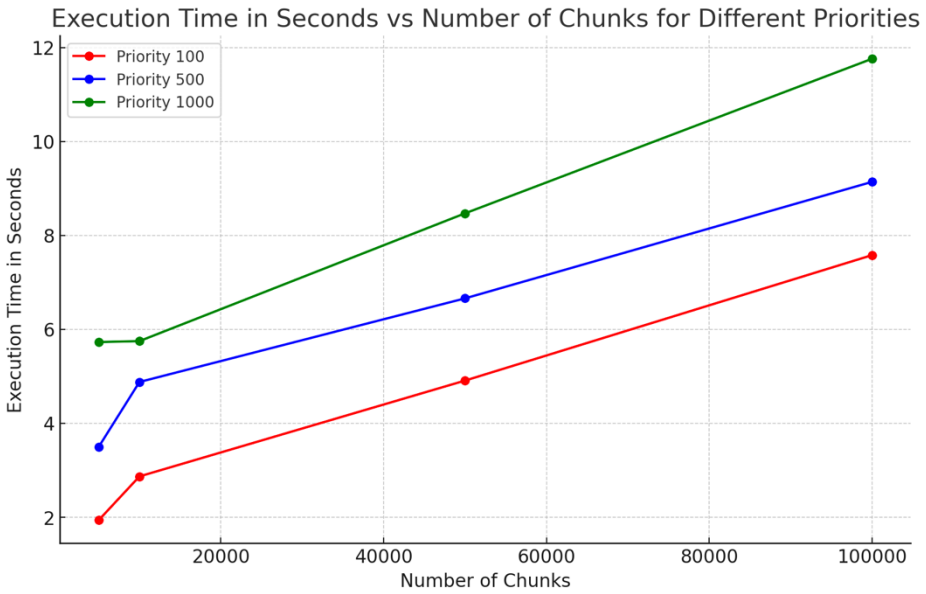
In the analysis of execution time for processing a file of constant size (1.3 GB), constant chunk size 100000, the impact of varying range of priorities was investigated across a number of threads. Linear dependency of range of priorities with time.

Increase in Execution Time with higher threads, Contrary to what one might expect, increasing the number of threads leads to an increase in execution time, due to the presence of batches and priority handling. When threads are assigned, chunks based on priority, a thread has to wait if it finishes its current chunk and the next chunk of its assigned priority isn't available yet because it's being processed by another thread, bottleneck caused by waiting for available chunks of specific priorities becomes more significant

In conclusion, the graph demonstrates a scenario where increased parallelism (more threads) does not equate to increased efficiency, likely due to the added complexity of managing and coordinating threads when tasks are prioritized and batched.

Execution Time vs Number of Chunks

File size = 1.3GB, Threads = 16



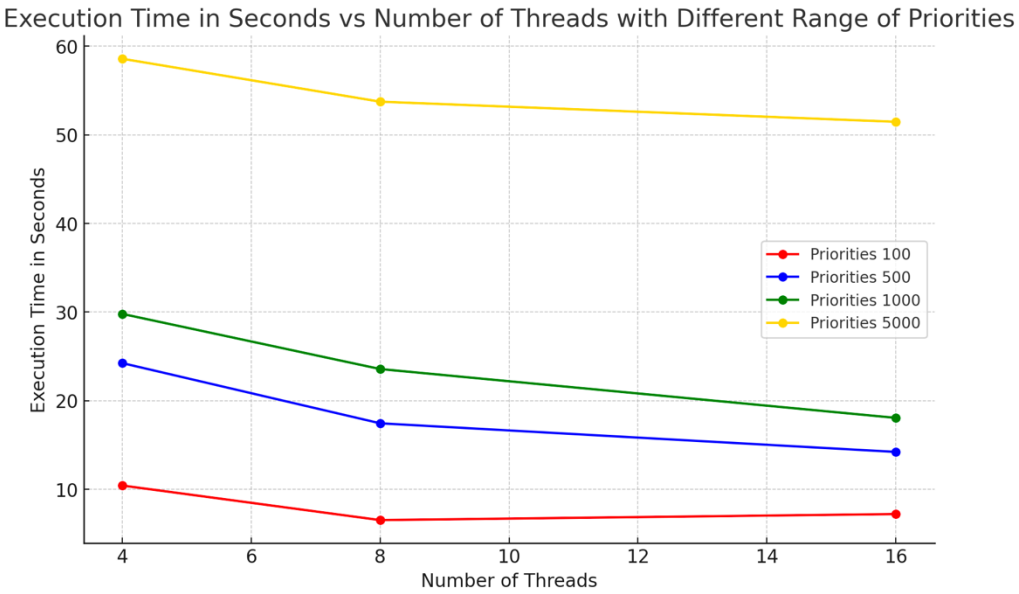
In the analysis of execution time for processing a file of constant size (1.3 GB), the impact of varying number of chunks (5000, 10000, 50000, 100000) was investigated across different priority levels (100, 500, 1000). The graph illustrates a noteworthy trend in execution time as a function of both chunk size and priority level.

As the priority level increases from lower (100) to higher (1000), there is a diminishing rate of improvement in execution time when the chunk size is increased. This suggests that while increasing chunk size does reduce execution time for all priorities, the extent of improvement becomes less pronounced at higher priorities.

The graph reveals a linear relationship between chunk size and execution time across all priorities. This linearity indicates that the execution time increases at a consistent rate with the increase in chunk size, regardless of the priority level. However, this rate of increase is steeper for lower priorities compared to higher ones. Increasing chunk sizes generally leads to faster execution times, and the degree of improvement is more substantial at lower priority levels.

Execution Time vs Number of Chunks

File size = 1.3GB, Chunks = 100000



In our analysis, we examined the impact of thread count on execution time for processing a file of size 1.3GB, divided into 100,000 chunks, across varying priority levels. The results, as depicted in the graph, reveal a notable trend: at lower priority levels, the difference in execution times across different thread counts is relatively small. This suggests that with fewer priority levels, threads are less likely to experience delays due to synchronization needs, as there are fewer priority levels to manage.

Conversely, at higher priority levels, a significant increase in execution time is observed as the number of threads increases. This is attributed to the complexity introduced by managing a larger number of priority levels, which results in increased synchronization wait times for threads. Specifically, with higher priorities, threads must frequently wait to synchronize until all chunks in a batch with the same priority level are processed, leading to longer overall execution times.

This pattern underscores the impact of priority management in multi-threaded processing, where balancing the number of threads and priority levels becomes crucial for optimizing execution time.

5.8 Challenges and Limitations

Upon analyzing the Parallel Chunk Reading and File Mirroring algorithms, we found out that the performance suffered a lot due to voluntary context switches, leading to the OS constantly switching to kernel space. This was solved by implementing memory-mapping which dramatically reduced the number of switches.

During the design of the file mirroring algorithm, replicating the RAID 1 architecture for difficult without the use of hardware RAID cards and disks. This involves running into disk quota issues and decreased performance.

In scenarios where the file size was not an exact multiple of the number of chunks, special handling was necessary for the last chunk. This adjustment ensured that all remaining bytes were accurately accounted for, compensating for any discrepancies or rounding errors arising from the initial division of the file into chunks.

6. Conclusions

Impact of L1-dcache miss: L1-dcache miss played a very crucial role in determining the performance of our programs. It degraded the performance of Chunkreader and improved the performance of FileMirroring so much so that despite additional checks in FileMirroring it was able to perform on par with the Chunk Reader model with the added benefit of providing fault tolerance to the system.

Efficiency of Memory Mapping: Memory mapping emerged as a powerful solution, particularly when dealing with large file sizes under constrained memory conditions. Notably, it demonstrated a significantly lower number of context switches and resident set size (RSS), leading to faster read operations on files. This efficiency makes memory mapping a valuable tool for optimizing performance with limited available memory.

Thread management with a prioritized and batched system: It doesn't always lead to faster execution, especially when threads idle waiting for specific priority chunks to finish up. It is influenced by chunk size optimization, as larger chunk sizes generally enhance processing speed, demonstrating a direct correlation between number of threads and execution efficiency. These factors converge to illustrate a delicate balance in parallel processing: the efficiency gained through parallelism must be carefully weighed against the complexities of thread coordination and priority management to achieve optimal performance.

7. References

[1] “Multi-Thread Processing.” HULFT, www.hulft.com/help/en-us/DataSpider/dss44sp2/help/en/servista/multi_stream_processing.html. Accessed 28 Nov. 2023.

[2] “Common Raid Disk Data Format (DDF).” SNIA, www.snia.org/tech_activities/standards/curr_standards/ddf. Accessed 28 Nov. 2023.

[3] Shenze Chen, and D. Towsley. “A performance evaluation of Raid Architectures.” IEEE Transactions on Computers, vol. 45, no. 10, 1996, pp. 1116–1130, https://doi.org/10.1109/12.543706.

[4] Privé, Florian, et al. “Efficient analysis of large-scale genome-wide data with two R packages: Bigstatsr and bigsnpr.” Bioinformatics, vol. 34, no. 16, 2018, pp. 2781–2787, https://doi.org/10.1093/bioinformatics/bty185.

[5] George, Laurent, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Diss. Inria, 1996.

[6] Zewei Chen, Hang Lei, Maolin Yang, Yong Liao and Lei Qiao. Blocking analysis of suspension-based protocols for parallel real-time tasks under global fixed-priority scheduling , 2021

[7] Emanuel H. Rubensson, Elias Rudberg, Chunks and Tasks: a programming model for parallelization of dynamic algorithms, https://ar5iv.labs.arxiv.org/html/1210.7427, October 2012.