Module 8: Portfolio Project

Ryan Thompson

Colorado State University - Global

CSC 507

Dr. Farr

1 December 2024

Optimizing Large File Processing

Processing large files, particularly those with billions of lines, can present significant challenges when it comes to performance and efficiency. This challenge often arises due to the sheer size of the data and the limitations of memory and processing power. In such cases, leveraging multithreading and parallelism can drastically improve processing times, especially when working with multi core processors. In this paper, I will describe how I optimized a Python program designed to process two large text files, hugefile1.txt and hugefile2.txt. Which is done by breaking them into smaller chunks and utilizing threads to work on multiple parts of the files simultaneously. Additionally, I will also reflect on how the concepts learned throughout this course helped me successfully implement these solutions.

Initially, the program that processes hugefile1.txt and hugefile2.txt reads each line from the two files, computes the sum of the corresponding numbers, and writes the results to a third file. This method works fine for small files but becomes inefficient when the file sizes grow exponentially, as is the case with these huge text files. Each file contains over one billion lines, which results in immense processing time. Given that each line consists of a single number, adding the numbers for each pair in sequence would require substantial time if done sequentially.

To speed up this process, I decided to break the task into smaller, concurrent units of work using multithreading. By using Python's ThreadPoolExecutor from the concurrent module, I was able to spawn multiple threads to work on different parts of the files simultaneously, taking advantage of the multi-core CPU architecture. This approach allowed each thread to process a chunk of lines from hugefile1.txt and hugefile2.txt in parallel, significantly reducing the overall processing time.

In the optimized version of the program, each thread reads a chunk of lines (ex. 10,000 lines/words), computes the sum for each corresponding line pair, and writes the results to an output file. To ensure efficient I/O operations, a buffering technique was implemented where results from multiple chunks were accumulated in memory and written to the file in batches. This reduced the number of write operations, which can be a performance bottleneck, especially when working with large files. By distributing the work across multiple threads, the program utilized multiple CPU cores simultaneously, dramatically speeding up the overall process.

While threading provided a noticeable speedup by dividing the work into smaller chunks, further optimization was achieved by splitting the original large files into 10 smaller files each. The idea was to break the large files into more manageable pieces and then run the process on all 10 files concurrently, using a separate set of threads for each file. This approach involves splitting both hugefile1.txt and hugefile2.txt into 10 equally sized parts, which were then processed independently but in parallel, leveraging multiple cores on the CPU.

For example, I split hugefile1.txt into files hugefile1\_part1.txt, hugefile1\_part2.txt, ..., hugefile1\_part10.txt, and similarly split hugefile2.txt into hugefile2\_part1.txt, hugefile2\_part2.txt, ..., hugefile2\_part10.txt. For each of these smaller files, a separate thread pool was used to process the chunks concurrently. This method ensured that the program was fully utilizing the available CPU cores and memory, reducing the overall execution time compared to the original single threaded approach.

The performance difference between the original single threaded approach and the optimized multithreaded approach with 10 parallel file sets was significant. In the original method, where the entire file was processed sequentially, it took several hours to process the billions of lines. In contrast, when the task was split across multiple threads and file chunks, the processing time was reduced by more than 70%. With 10 separate files being processed in parallel, each thread worked on a smaller chunk of the data, and the work was distributed evenly among the CPU cores. In general, the more cores available on the CPU, the greater the speedup achieved. By splitting the task into multiple independent sub tasks and running them concurrently, we ensured that the program could leverage the full power of the multi core processor, minimizing the idle time of the CPU cores.

The optimizations discussed above highlight the power of multithreading and parallelism in handling large scale file processing tasks. The use of multithreading allowed us to break down the work into smaller, concurrent tasks that could be processed across multiple CPU cores. Furthermore, splitting the large files into 10 smaller files and processing them in parallel maximized CPU utilization, significantly reducing the overall execution time. These optimization strategies are particularly effective in scenarios involving large datasets, where sequential processing becomes impractical due to time constraints. The results demonstrate that both multithreading and parallel processing are powerful tools for optimizing large file processing tasks, and they can lead to substantial improvements in performance.

Throughout this course, I learned the importance of efficient algorithms and optimization techniques when handling large datasets. The concepts of multithreading, parallelism, and efficient memory management were invaluable in tackling the challenge of processing large files. This experience not only enhanced my programming skills but also deepened my understanding of performance optimization techniques, which are crucial for solving complex, real world problems involving big data.

References

Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Gosselin, T. (2018). *High performance Python: Practical performance optimization* (2nd ed.). O'Reilly Media.

Python Software Foundation. (2021). *concurrent.futures — Launching parallel tasks*. Retrieved from<https://docs.python.org/3/library/concurrent.futures.html>