Optimised Multi-file Word Occurrence Counter.

Alex Purser

Falmouth University - 2202628

Introduction

Word occurrences are the number of times a specific word appears in a string or text. There are two main requirements it needs to do: take in the desired string or text, and compare each word to the desired word, incrementing a counter accordingly. These aspects can be optimised individually and compiled together. This project is a multi-file word occurrence counter that has been developed in C++ and optimised for minimal run-time using multithreading and memory management text-parsing.

Background

Word counters are implemented in every day software such as Word and Notepad using 'Ctrl + f' (find). While they are quick, they are typically used for smaller and constant file sizes.

Hongmei Xie and Chunli Chen display a good example of large file analysis within: *'Design and implementation of parallel word occurrence frequency based on message passing interface'*. This paper discusses the problems of word occurrence counters and an implementation of a serial word counting algorithm for parallel processing large text files [1].

Furthermore these shortcuts only apply to singular documents. One study from Nithin Kavi, implemented a *'MapReduce for Counting Word Frequencies with MPI and GPUs'* which utilised the Julia programming language and parallelisation to count across multiple documents [2].

A renowned text analysis software: the Linguistic Inquiry and Word Count (LIWC) is considered "the gold standard in software for analyzing word use" [3]. This software can delve into psychometric analysis, determining people's psychological states based on text analysis. This is shown in James W. Pennebaker et. al's paper: *'The Development and Psychometric Properties of LIWC-22'* [4]. Further functionality shows dream analysis: a study by K. Bulkeley et al on *'Using the LIWC program to study dreams'* [5].

This project aims to implement key aspects of these papers, being able to process multiple large text files, while overcoming the problems with word occurrence counters. By achieving this milestone, this project will work as a baseline for more text analysis features.

Optimisations

Text Parsing using Memory Management

The first optimisation is text parsing. Text parsing is the movement of text into a program and can be quite complex due to file size and type of data. This project needs to access and parse text across multiple text files within the directory "texts" to begin comparing them. This adds another layer of complexity due to the need for merging of the files. By correctly optimising this, the parsing time can be reduced, drastically impacting the total runtime.

Multithreading

The second optimisation is multithreading. Most programs execute programs using a single thread, and in-turn, a small percentage of the CPU. Multithreading utilises many threads within a CPU to run parallel when executing, increasing the CPU usage to reduce the runtime of a project [6]. Dean M. Tullsen et. al displayed this in their paper: *'Simultaneous multithreading: maximising on-chip parallelism'* [7].

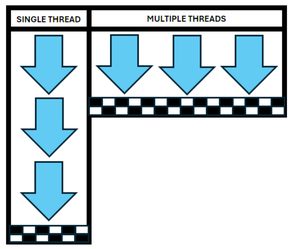

Figure 1. Multithreading Diagram

Methodology

Text parsing

Using the "getLine" function in the C++ 'string' header, the first version parsed each line into its own element within a vector. This had a time complexity of O(n^2) due to thread contention and having the threads scan through each line to get to their start point. Using the "file_size" within the C++ 'filesytem' header, the file size can be gathered in bytes, after which the file is transferred using the devices memory. In doing so, the variables storing the text are easily traversal and allow for efficient chunking for each thread. This reduces the time complexity from O(n^2) to O(n). Additionally, gathering the file size is useful for determining if the system has enough available memory to store the file in its entirety. This ensures the project is accessible for lower spec devices, and prevents the project from encountering any memory allocation errors [8].

In the first version, if the file size was bigger than memory, the program would process the input file in chunks using "buffer.write()", otherwise the file would be written in all at once using "inputFile.rdbuf()". After testing, "buffer.write()" proved faster than "inputFile.rdbuf()" and so was implemented as the next version of text parsing.

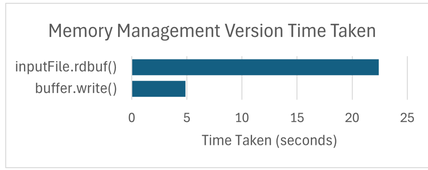

Figure 2. Memory Management Version Analysis 2 files 500MB

Multithreading

The next improvement was multithreading the counting process. First, the project gathers the number of available threads using the "hardware_concurrency()" function. Once gathered, the project uses the maximum number of available threads to count equal sized chunks of the text supplied, reducing total runtime. By gathering the number of available threads, this makes the project adaptable for devices across all specifications. Furthermore, if "hardware_concurrency()" does not work, the thread count is automatically set to one to ensure the program operates correctly.

While multithreading can improve performance, it is important to note the overhead cost of implementation. The process of building and calling multiple threads adds a setup cost, which for small projects, might prove slower than a single thread [9].

Debugging

The built-in Visual Studio 2022 profiler showcases information about memory and CPU usage. This example shows four threads being used on two 500MB files.

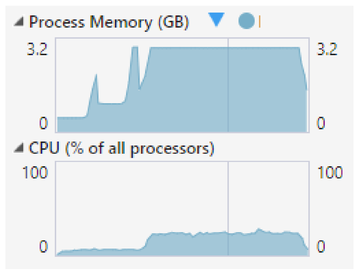

Figure 3. Visual Studio Profiler

Using the profiler, we can see these traits:

Memory

- The spikes showcase each file being stored in memory, then quickly moved and cleared.
- The constant line at the top shows the memory allocation per thread during counting.

CPU Usage

- The flat line at the start indicates the text parsing.
- The sudden increase shows the threads counting.

Analysis

To maintain a fair and controlled test, each optimisation has been measured using the same setup code with minimal background applications.

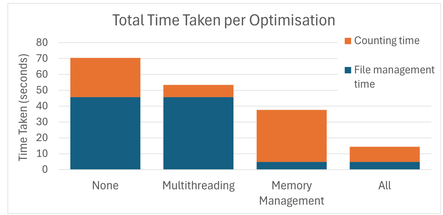

Figure 4. Optimisation Analysis Graph of Two 500MB Files.

Figure 3 showcases the results of each optimisation and their effect on runtime. Each optimisation has been tested with two files of 500MB file size. "None" is the first version of the project, having a time complexity of O(n^2) as the program would run through each line in the vector every time before reaching the desired line. Each optimisation shows a substantial change in their specific aspect's time taken, and overall runtime.

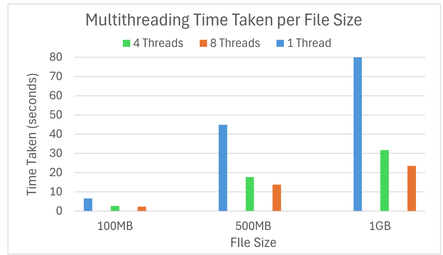

Figure 5. File Size Effect on Time Taken

Figure 4 displays the effect file size has on run time. It is evident that file size increases run time linearly at O(n), with minor fluctuation. This trend is reflected across all threads but each displays a slightly different steepness for time taken: 1 showing the steepest gradient and 8 showing the shallowest. This is likely due to the reduction in counting time, illustrated in the difference between 4 threads and 8 threads across the various files.

Future Work

Future enhancements would be to improve the user friendliness via the use of a User Interface (UI) and control the number of running threads to account for smaller files. For long-term implementations, this project works as the baseline for text analysis, similar to LIWC [3]. This level of functionality provides many opportunities in the field of linguistics analysis, being able to analyse text in many scenarios like social media or the news.

References

[1] H. Xie and C. Chen, "Design and implementation of parallel word occurrence frequency based on message passing interface," in *2011 Second International Conference on Mechanic Automation and Control Engineering*, Jul. 2011, pp. 3985–3988. [Online]. Available: https://ieeexplore.ieee.org/document/5987874

[2] N. Kavi, "MapReduce for Counting Word Frequencies with MPI and GPUs," May 2022. [Online]. Available: https://arxiv.org/abs/2206.05269v1

[3] "Welcome to LIWC-22." [Online]. Available: https://www.liwc.app/

[4] R. Boyd, A. Ashokkumar, S. Seraj, and J. Pennebaker, *The Development and Psychometric Properties of LIWC-22*, Feb. 2022.

[5] K. Bulkeley and M. Graves, "Using the LIWC program to study dreams," *Dreaming*, vol. 28, no. 1, pp. 43–58, 2018, place: US Publisher: Educational Publishing Foundation.

[6] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Computing Surveys*, vol. 35, no. 1, pp. 29–63, Mar. 2003. [Online]. Available: https://doi.org/10.1145/641865.641867

[7] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, May 1995, pp. 392–403. [Online]. Available: https://dl.acm.org/doi/10.1145/223982.224449

[8] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong, and J. Wei, "Experience report: A characteristic study on out of memory errors in distributed data-parallel applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2015, pp. 518–529. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7381844

[9] H. Kwak, B. Lee, A. Hurson, S.-H. Yoon, and W.-J. Hahn, "Effects of multithreading on cache performance," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 176–184, Feb. 1999, conference Name: IEEE Transactions on Computers. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/752659