Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Operating System Concepts

ENCS3390

Project 1

Prepared by:

Mohammad Khdour 1212517

Instructor: Mohammad Khalil

Section: 2

Data: 2/12/2024

Abstract

The aim of the project is to be able to use parallel programming using multi-processes and multithreading to utilize the execution time for counting the most frequent words from the cleaned file by noticing the time difference in each method and comparing it with the native one.

Environment description

in this project, the development is carried out on **macOS**, a Unix-based operating system powered by an Intel **Core i9 processor** with **8 cores**, Paired with **32 GB of RAM.** The primary programming language is **Java**, chosen for its platform independence and extensive libraries, and development is carried out in **IntelliJ IDEA**, This setup runs natively on macOS **without a virtual machine**, thus the hardware is utilized to its full extent and the result is optimal performance.

Procedure:

Naïve, Multi-processing, and multi-threading achievement

1. **Naive Approach**

The naive approach was implemented without utilizing any parallelism, making it a clear method for processing the data. The entire wordlist was read and processed sequentially within the main thread that iterated over the list of words, using a HashMap to count their occurrences. After counting, the results were sorted by frequency, and the top 10 most frequent words were identified. This approach ensured simplicity but was not optimized for larger datasets as it does not leverage multi-core processing capabilities. Its execution time served as a baseline for comparison against the multiprocessing and multithreading methods.

2. **Multiprocessing Approach**

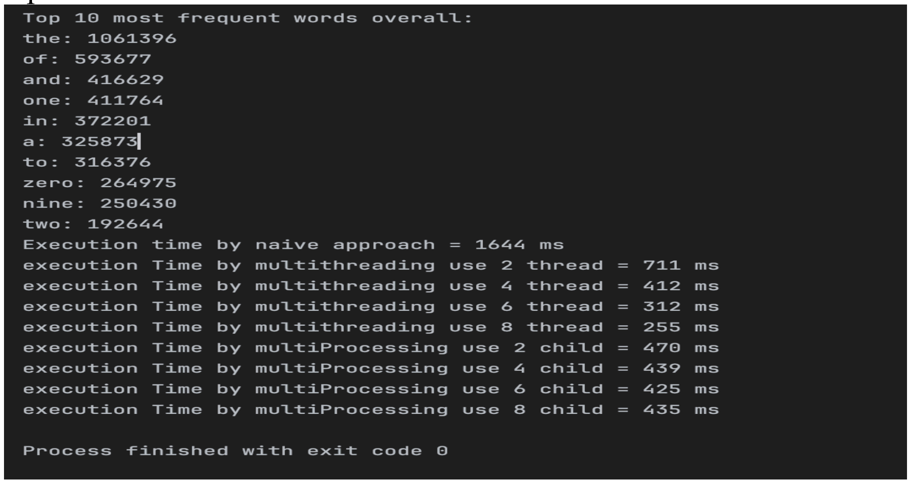
The multiprocessing approach was implemented using Java's Fork/Join Framework, which is part of the **java.util.concurrent** package. The **MyProcess** class inherits another class called **RecursiveTask**, allowing it to perform parallel computations by putting the code that responds to count the most frequent words in the **compute()** method. The input dataset was divided into parts, each part assigned to a separate **MyProcess** instance. These tasks were submitted to a **ForkJoinPool**, which managed their execution across multiple cores. The **fork()** method distributed the tasks in a separate process, while **join()** was used to retrieve the results from the child processes. After all the results were combined, the top 10 most frequent words were determined. This approach was efficient for larger datasets, as it took full advantage of multiple cores.

3. **Multithreading Approach**

The multithreading approach utilized the **ExecutorService** and **Callable** interfaces from Java's **java.util.concurrent** package. The dataset was split into parts, and each part was assigned to a **MyThread** instance, which implemented the **Callable** interface. A thread pool was created using **Executors.newFixedThreadPool(),** where the number of threads could be varied to analyze performance under different levels of concurrency. Each thread independently processed its assigned portion of the dataset, calculated word frequencies, and returned the top results via **Future** objects. The results from all threads were then combined to compute the overall top 10 most frequent words. This method provided flexibility and reduced overhead compared to processes, making it ideal for tasks that share memory.

Result analysis

The system output:



```
Top 10 most frequent words overall:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time by naive approach = 1644 ms
execution Time by multithreading use 2 thread = 711 ms
execution Time by multithreading use 4 thread = 412 ms
execution Time by multithreading use 6 thread = 312 ms
execution Time by multithreading use 8 thread = 255 ms
execution Time by multiProcessing use 2 child = 470 ms
execution Time by multiProcessing use 4 child = 439 ms
execution Time by multiProcessing use 6 child = 425 ms
execution Time by multiProcessing use 8 child = 435 ms

Process finished with exit code 0
```

Identifies performance gains from adding additional cores to an application that has both serial and parallel components

Amdahl's Law states:

$$Speedup = \frac{1}{S + \frac{1-S}{N}}$$

S is the serial portion

N num of processing cores

Serial partition calculation

Naive execution time T_{naive} = 1644 ms

Best multithreaded execution time T_{best} = 255 ms (using 8 threads)

To find the serial partition, first, we must find the speed-up

$$speedup = \frac{execution\ time\ old}{execution\ time\ new}$$

$$speedup = \frac{1644\ ms}{255\ ms} = 6.45$$

Now, by applying Amdahl's Law to compute serial partition with a speedup of 6.45 at 8-thread

$$Speedup = \ 6.45 = \frac{1}{S + \frac{1-S}{8}}$$

$$6.45\left(S + \frac{1-S}{8}\right) = 1$$

$$\frac{6.45}{8}(8S + 1 - S) = 1$$

$$0.801(7S + 1) = 1$$

$$7S + 1 = 1.25$$

$$S = 3.5\%$$

Maximum Speedup

maximum speedup according to the available number of cores which is 8-core N→ 8:

$$Speedup = \frac{1}{0.035 + \frac{1-S}{8}} = 6.61$$

The **maximum speedup** achievable with 8 cores is approximately **6.61x**. This is the theoretical upper limit under ideal conditions with perfect load balancing and no additional overhead.

Optimal number of child processes and threads

The optimal number of child processes or threads depends on the relationship between the computational workload, system architecture, and the overhead of managing parallelism. Based on the given data and analysis:

For Multithreading:

Execution times decrease consistently with increasing threads up to **8 threads**, With 8 threads, the execution time is the lowest, indicating that the system's performance scales well with the number of threads. Given that the device has 8 cores**,** 8 threads is the optimal choice as it fully utilizes all available cores.

For Multiprocessing:

The best performance is achieved with **6 child processes**, as adding more processes introduces overhead without further improvement. This is likely due to increased communication and context-switching costs in multiprocessing.

Performance table

# of processor and threads	**2 P/T**	**4 P/T**	**6 P/T**	**8 P/T**
Naïve Approach	**1644ms**			
M-processing	470ms	439ms	425ms	435ms
M-Threading	711ms	412ms	312ms	255ms

difference in performance

The results demonstrate clear differences in performance across the three approaches. The naive approach is the slowest (1644 ms), as it processes the dataset sequentially without leveraging parallelism. Both multithreading and multiprocessing significantly improve execution time by utilizing multiple cores, with multithreading achieving the best performance (255 ms with 8 threads). Multithreading benefits from a shared memory model, reducing overhead and context switching, which allows efficient workload distribution, while multiprocessing, though faster than the naive approach, shows slightly higher overhead due to process creation, process communication, and context switching. Interestingly, multiprocessing performance peaks at 6 processes (425 ms) and slightly declines with 8 processes (435 ms), likely due to increased overhead. Overall, multithreading proves to be the most efficient for this task, offering the fastest execution time as the number of threads scales.

Conclusion

In conclusion, the implementation and analysis of the three approaches, naive, multithreading, and multiprocessing, highlight the advantages of leveraging parallelism for performance improvement. The naive approach serves as normal levels but is significantly slower due to its sequential nature. Both multithreading and multiprocessing demonstrate substantial performance gains by utilizing multiple cores, with multithreading exceeding multiprocessing in this case due to its lower overhead and efficient memory sharing. While multiprocessing provides better scalability for processes requiring isolation, It results in higher costs for inter-process communication and process management. The optimal approach depends on the specific task and hardware; however, for this program, multithreading with 8 threads provided the fastest execution time, demonstrating its effectiveness for tasks involving shared data and fine-grained parallelism. This comparison emphasizes the importance of understanding the characteristics of each parallelization technique to achieve optimal performance.