Birzeit University

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS3390, OPERATING SYSTEMS

project #1

multiprocessing & multithreading

Prepared by: Mohammad Manasrah

ID: 1211407

Section: 2

Instructor: Dr. Mohammad Khalil

Date: 7/12/2024

Abstract

This project analyzes three approaches—naive, multiprocessing, and multithreading—for finding the top 10 most frequent words in the enwik8 dataset. Performance is evaluated using varying processes/threads (2, 4, 6, 8) on a 6-core machine. Execution times are compared, and results are analyzed using Amdahl's law to identify the optimal configuration. Insights on performance trade-offs and parallelism effectiveness are provided.

Contents

List of Figures

List of Tables

1 Theory

1.1 multiprocessing

An operating system that uses many CPUs to boost performance is known as a multiprocessing operating system. In multi-processing operating systems, several processors collaborate to complete the task at hand. Every CPU that is accessible is linked to clocks, computer busses, peripheral devices, and physical memory. Increasing system execution speed is the primary goal of the multi-processing operating system. The system performs better overall when a multiprocessing operating system is used. For instance, the most popular multi-processing operating systems are Solaris, LINUX, and UNIX.[1]

1.2 multithreading

Operating systems have a capability called multithreading that enables programs to do many tasks simultaneously. Imagine it as several hands collaborating to finish various tasks more quickly. These ”hands,” which are referred to as threads, aid in improving the efficiency of programs. Applications like web browsers, games, and many more daily programs run faster and more smoothly when your computer is multithreaded because it makes greater use of its resources.[2]

2 Environment description

The development environment consists of a powerful computer with a 12-core processor running at a speed of 2933 MHz, supported by 32 GB of RAM and 1 TB of storage. Of the 12 cores, 6 are allocated to a virtual machine running Ubuntu Linux, with 22 GB of RAM dedicated to this setup. The host operating system is Windows, providing the base environment. The primary programming language used is C, developed using the Code::Blocks IDE, offering an efficient platform for compiling and debugging. The virtual machine setup ensures seamless multitasking and development across both operating systems.

3 Achieving the multiprocessing and multithreading

To meet the multiprocessing and multithreading requirements, various APIs and methods were employed. The `fork()` system function was utilized to create multiple processes, where each process was responsible for processing a subset of words.

In order to facilitate data sharing across processes, shared memory was managed using `mmap()` with the `MAP_SHARED` flag. The `wait()` function ensured that all processes were synchronized. The local word frequency computations performed by each process were subsequently combined into the shared memory.

This approach effectively leveraged multiprocessing techniques to ensure efficient parallel processing while maintaining consistency across shared resources.

For multithreading, threads were created and managed using the `pthread` library. Threads were initiated using `pthread_create()`, where custom parameters were passed through a `ThreadArgs` structure. This structure specified the data region each thread was responsible for.

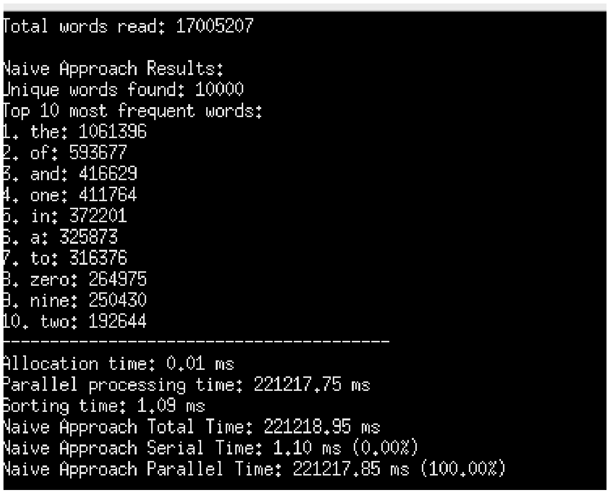
After execution, threads were synchronized using `pthread_join()` to ensure that all threads completed before proceeding. In the serial part of the program, the local word frequency computations performed by the threads were merged and arranged for further processing.

The program leveraged the `qsort()` function for sorting word frequencies in both the multiprocessing and multithreading approaches. To measure the execution times of both serial and parallel processes, efficient timing mechanisms were employed using `clock_gettime()`.

These methodologies ensured optimal utilization of CPU cores and memory resources, facilitating effective parallel processing and maximizing performance.

4 analysis according to Amdahl's law

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$



```
Total words read: 17005207

Naive Approach Results:
Unique words found: 10000
Top 10 most frequent words:
1. the: 1061396
2. of: 593677
3. and: 416629
4. one: 411764
5. in: 372201
6. a: 325873
7. to: 316376
8. zero: 264975
9. nine: 250430
10. two: 192644
----------------------------------------
Allocation time: 0.01 ms
Parallel processing time: 221217.75 ms
Sorting time: 1.09 ms
Naive Approach Total Time: 221218.95 ms
Naive Approach Serial Time: 1.10 ms (0.00%)
Naive Approach Parallel Time: 221217.85 ms (100.00%)
```

Figure 1: naive approach

according to the figure the parallel portion of the code is very high, ranging from 99.9975% to 99.9988% across different process counts. This indicates that nearly all of the task is parallelizable, with only a tiny fraction being serial.

For 2 processes:

$$S = (1 - 0.999975) + (2 \times 0.999975)$$

$$1 - 0.999975 = 0.000025$$

$$2 \times 0.999975 = 1.99995$$

$$0.000025 + 1.99995 = 1.999975$$

So,

$$S_{theoretical} = 1.999975$$

The actual is:

$$S_{actual} = \frac{221218.95}{117165.43} \approx 1.89$$

For 4 processes:

$$S_{theoretical} = 3.999925$$

The actual is:

$$S_{actual} = \frac{221218.95}{73554.51} \approx 3.01$$

For 6 processes:

$$S_{theoretical} = 5.999875$$

The actual is:

$$S_{actual} = \frac{221218.95}{54553.62} \approx 4.06$$

For 8 processes:

$$S_{theoretical} = 7.999825$$

The actual is:

$$S_{actual} = \frac{221218.95}{58303.44} \approx 3.79$$

The relationship between theoretical and actual speedups shows that while Amdahl's law predicts increasing speedups with more processes, real-world factors limit the achievable performance. The gap between theoretical and actual speedups widens as the number of processes increases, demonstrating the practical limitations of parallelization.

for the 6 cores the maximum speedup was 4.06 with 6 processes, The drop in performance beyond 6 cores suggests that adding more cores leads to diminishing returns. This happens because the overhead of managing additional cores, resource bottlenecks like memory bandwidth, or a problem size that isn't large enough to fully utilize extra cores all contribute to decreased efficiency. Therefore, finding the optimal number of cores for a specific workload is crucial to maximize performance.

5 compare the performance

Number of Cores	Multiprocessing (ms)	Multithreading (ms)	Speedup (Threads)
Serial (1)	221,218.95	221,218.95	1.00x
2	117,165.43	119,057.07	1.86x
4	73,554.51	70,395.30	3.14x
6	**54,553.62**	**56,767.98**	**3.90x**
8	58,303.44	58,199.08	3.80x

Table 1: Performance Comparison Across Different Approaches

Note: The best performance (6 cores) is highlighted in bold.

6 overall discussion

6.1 Sequential vs Parallel Performance

221,218.95 ms was the baseline sequential execution time, which serves as a benchmark for parallel execution comparisons. When compared to sequential execution, multiprocessing and multithreading both exhibit notable gains. The potential of parallelization is demonstrated by the about a double speedup that occurs when switching from sequential to employing only two cores.

6.2 Scaling Pattern

Both multiprocessing and multithreading show encouraging gains when growing from two to four cores. The speedup goes from 1.89x to 3.01x for processes and from 1.86x to 3.14x for threads. In both situations, this suggests good scalability. Processes show a performance jump from 3.01x to 4.06x (the greatest performance) and threads improve from 3.14x to 3.90x as the number of cores increases from 4 to 6. However, performance starts to suffer when increasing from 6 to 8 cores. Threads decline from 3.90x to 3.80x, while the process-based method reduces from 4.06x to 3.79x, indicating decreasing results beyond 6 cores.

6.3 Optimal Performance Point

The optimal performance point appears to be at 6 cores for both multiprocessing and multithreading. At this point, multiprocessing slightly outperforms multithreading. The best performance for processes is achieved with a time of 54,553.62 ms, which corresponds to a 4.06x speedup, while the best performance for threads is 56,767.98 ms, which corresponds to a 3.90x speedup.

6.4 Performance Degradation

Adding more cores beyond 6 results in performance degradation. This suggests several contributing factors, including increased overhead from managing additional processes or threads, resource contention (such as CPU, memory, and cache), and communication overhead between processes or threads. These factors become more pronounced as the number of cores increases, ultimately diminishing the potential performance gains from parallelization.

6.5 Processes vs Threads Comparison

Overall, both processes and threads exhibit similar performance patterns. Processes tend to outperform threads slightly at 6 cores, but the performance converges at 8 cores, with both approaches reaching around 58,200 ms. The distinction between processes and threads lies in their advantages: processes provide better isolation but incur more overhead, while threads offer shared memory and less overhead but may experience contention between them. This comparison underscores the trade-offs involved in choosing between multiprocessing and multithreading for parallel tasks.

conclusion

In conclusion, this analysis highlights the importance of identifying the optimal number of cores for parallel applications. While increasing the number of cores can initially improve performance, more cores do not always lead to better results. Factors such as overhead, resource contention, and communication requirements can limit the benefits of additional cores. Therefore, achieving maximum efficiency in parallel computing requires careful consideration of these elements to determine the ideal number of cores for a given task.

References

[1] GeeksforGeeks, “Multi-processing in Operating System,” `https://www.geeksforgeeks.org/multi-processing-operating-system/` (accessed December 8, 2024).

[2] GeeksforGeeks, “Multithreading in Operating System,” `https://www.geeksforgeeks.org/multithreading-in-operating-system/` (accessed December 8, 2024).