Faculty of Engineering & Technology
Electrical & Computer Engineering Department

Operating Systems ENCS3390

Application on parallelism

Prepared By:
Mumen Anbar 1212297

Instructor: Dr. Bashar Tahayna
Section: 4
Date: Dec 5, 2023

Abstract:

The findings of this study aim to provide valuable insights into the selection and optimization of parallel computing strategies, assisting in determining the most suitable approach for specific computational tasks like matrix multiplication, balancing performance gains against resource costs.

Before proceeding further, it's essential to clarify certain definitions.

Multiprocessing: Multiprocessing refers to the use of multiple cores or processors within a computer system to execute multiple tasks or processes simultaneously. Each process runs independently and can perform its own computations.

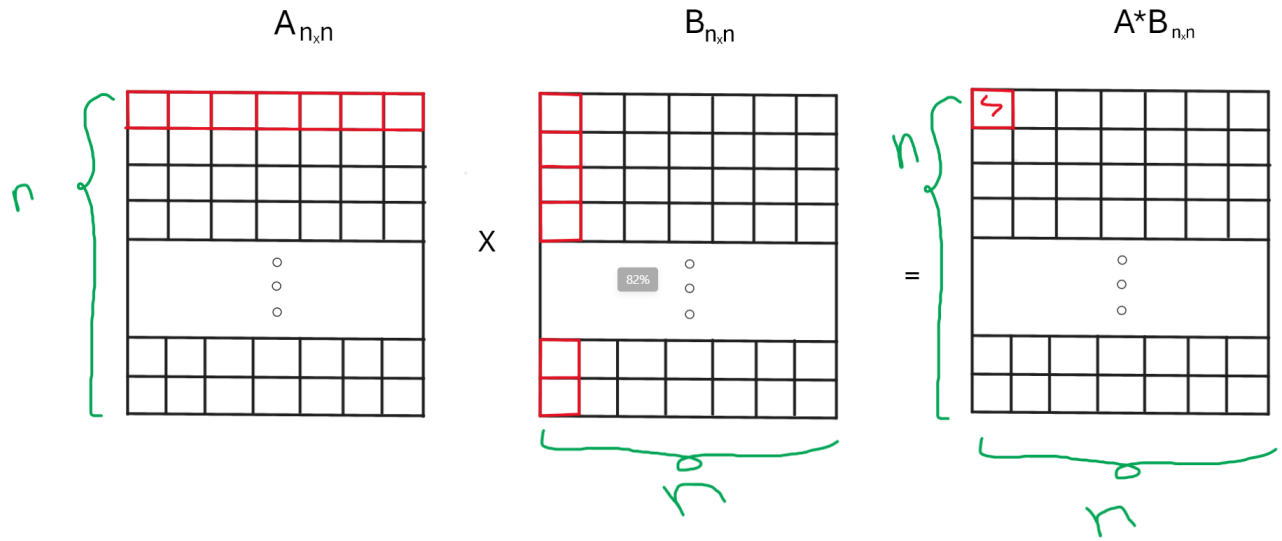
Multithreading: Multithreading, on the other hand, involves breaking a single process into multiple threads of execution. Threads within a process share the same memory space and resources, allowing them to execute concurrently. They can perform different parts of a task simultaneously, utilizing the advantage of the potential parallelism within a single process.

Context Switching: Context switching refers to the process performed by the operating system to store and restore the state of a CPU so that it can switch between different processes or threads. Where it saves the current state (context) of the running process or thread, including its registers, program counter, and other necessary information. It then loads the saved state of the next process or thread to be executed.

Threads: They are part of a process and share the same memory space, allowing multiple threads to exist within a single process. Threads can execute tasks concurrently, enhancing the efficiency of the program by dividing workloads and leveraging parallelism within the same memory space.

Introduction

In this task, matrix multiplication serves as a practical case study to enhance throughput and reduce execution time for a set of operations. Initially, the native approach is employed, highlighting that utilizing only one process results in a computational complexity of O(n^3) for calculating the multiplication answer. Given the independence of each operation group (e.g., rows of the resultant matrix), leveraging the concept of parallelism becomes viable for task execution. This approach allows the distribution of tasks across processes, enabling concurrent computation and potentially decreasing the overall computation time by exploiting parallelism. So, the solutions we require lie within the concepts of multiprocessing and multithreading. Indeed, both methodologies possess distinct advantages and drawbacks that define their respective characteristics.

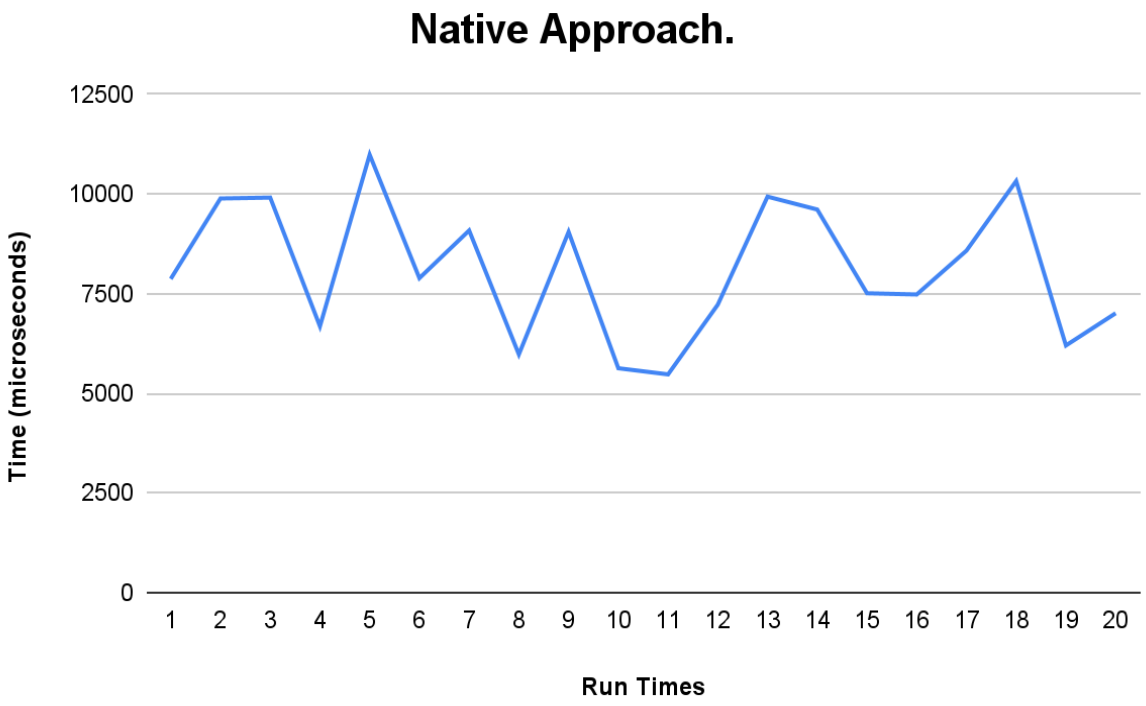


Approaches were used for reducing the execution time:

1- Native Approach: calculate the answer using 3 nested loops where

$$ans[i][j] = \sum\sum\sum A[i][k] * B[j][k]$$

This approach involves employing a single process to calculate the answer matrix, limiting the utilization of the CPU and not fully leveraging its potential for computational tasks.



$With\ average\ time\ execution\ =\ 7704.45\ \mu s$

From the native approach we can calculate the number of operations needed to find the answer matrix. This leads us to use it to find the throughput of each approach where,

$$Throughput\ =\ numOfOperations\ /\ Avg.\ Execution\ Time,$$
$$numOfOperations\ =\ 2\ *\ 100\ *\ 100\ *\ 100\ =\ 2\ *\ 10^{6}\ operations$$

$$Throughput\ for\ native\ approach\ =\ 2\ *\ 10^{6}\ /\ 7704.45\ *\ 10^{-6}$$
$$=\ 259590237\ operations/sec$$

2- Multiprocessing Approach: Leveraging the concept of parallelism optimizes CPU utilization by segmenting the answer matrix into **K** row blocks, allowing each process to compute a distinct block. Subsequently, reassembling these blocks yields the resultant matrix, maximizing the efficiency of CPU utilization.

Procedure:

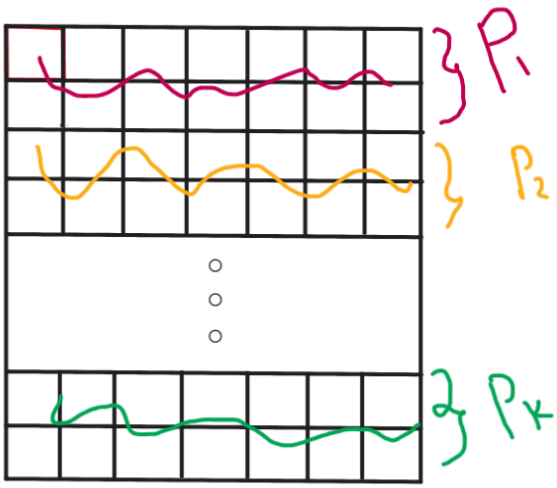
1) Creating **K** * 2 pipes since we have **K** processes to use, K pipes to transfer data from parent to child and the other **K** pipes to transfer data in opposite directions.
2) Determination of the length of the block where it equals:

$$blockLength = 100 / K$$

3) Initialize the array to be sent to the child process and contains the current block limits.
4) Create the pipes to be used to communicate between parent and child.
5) Send block limits to the child process and let it start calculating.
6) Repeat blocks one after another until reach the K-th block
7) When reaching the K-th block the block limit of the current block may not equal the previous ones since the number K couldn't be divisible by the number of rows (n = 100). So, with no doubt,

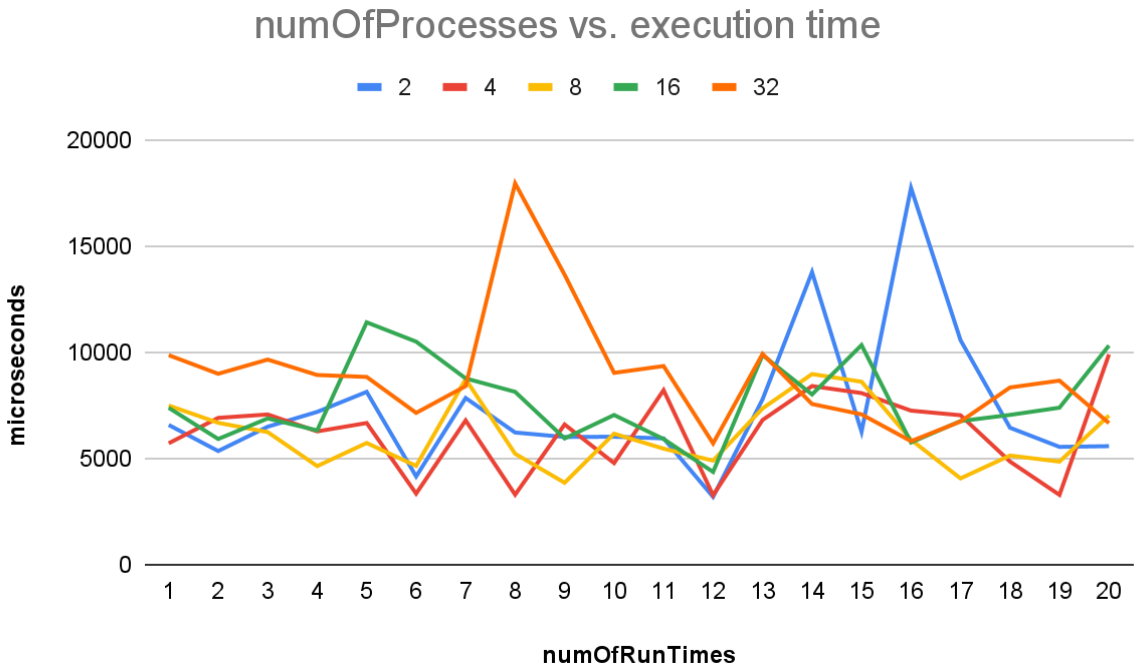
$$(Kth\ blockLength = 100 - lastBlockEnd) >= blockLength$$

8) Wait for the processes to finish calculating.
9) Regather the blocks of the answer matrix.
10) Measure the execution time.



Relationship Between Number of Processes K and Execution Time

The relationship between the number of processes and execution time might initially show improvements as more processes are added, but it might reach a point where adding more processes doesn't proportionally reduce execution time due to increased overhead. This pertains to various factors, including the count of internal CPUs, the number of cores, the frequency of each CPU, among other considerations.

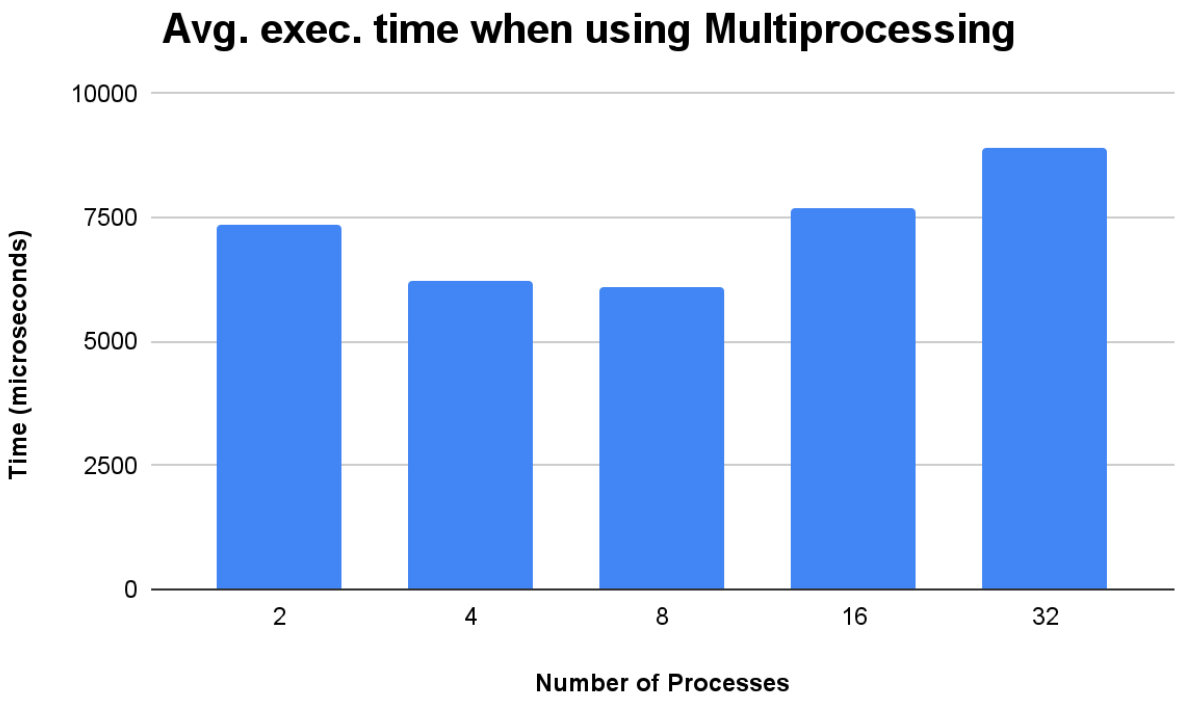


Must take in account that the variation in execution times across multiple runs is influenced by the instantaneous state of the CPU, which significantly impacts the outcome. A crucial observation: Before each program run, the cached memory was **flushed** using an internal command to flush the cache memory.

```
sudo sh -c "sync; echo 1 > /proc/sys/vm/drop_caches"
```

Measuring the execution time was using the function:

```
gettimeofday()
```



Analyzing the average runtime values in relation to execution time highlights the efficiency of utilizing 8 processes. Several factors contribute to this observation:

1) The increase in process count doesn't necessarily directly decrease execution time. External operations, such as context switching, are incurred by the CPU, contributing to an overall increase in time.

2) The performance of the running machine significantly influences this analysis. In this instance, the computer used has 4 CPUs, each containing 2 internal cores. This information can be retrieved using the command:

```
sudo lscpu
```

3) The execution time at each process count is strongly influenced by the longest process duration, as the main process must wait for all others to finish. When the number of processes is not evenly divisible into 100, the last process becomes the longest one, dictating the overall

execution time.

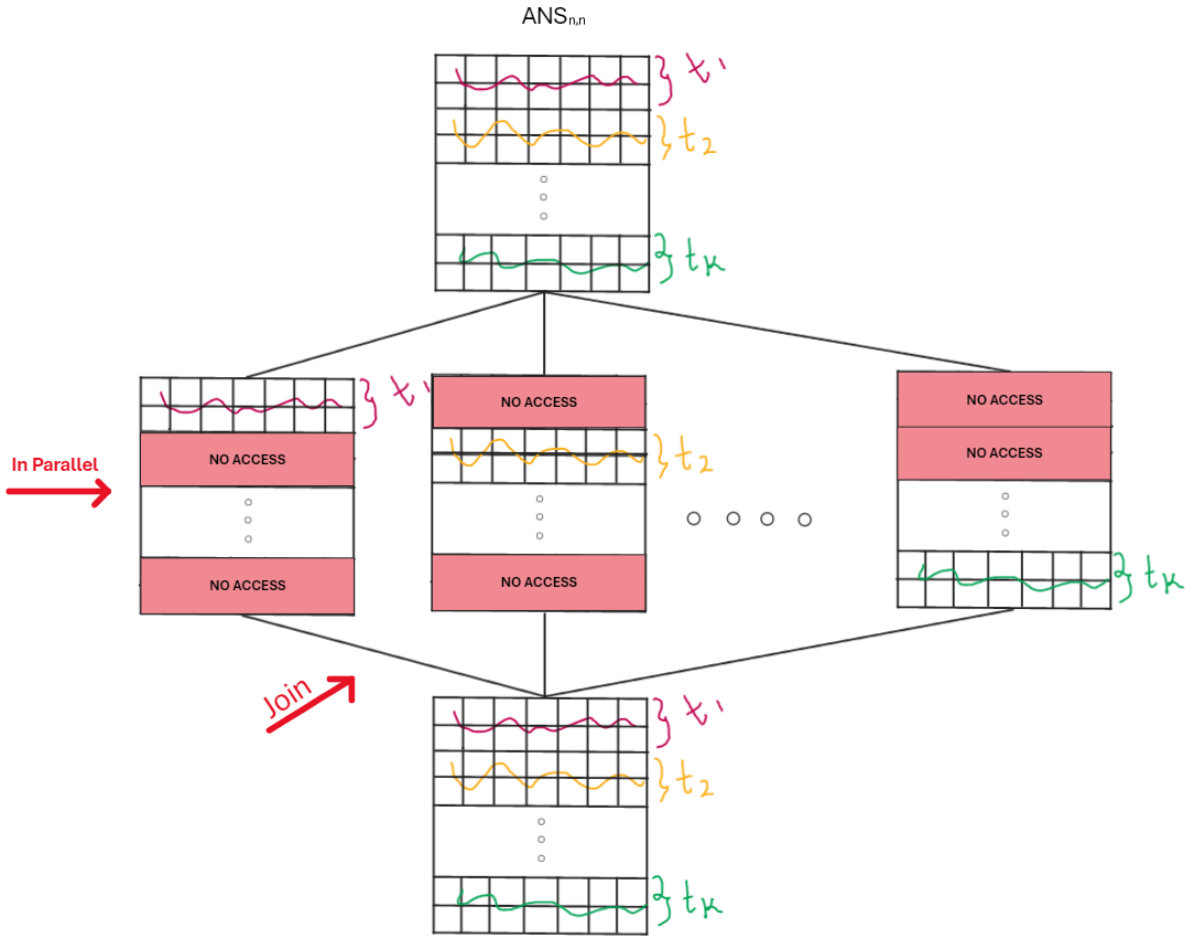
$$Throughput\ for\ MultiProcessing\ approach\ when\ using\ 8\ processes\ =$$
$$2\ *\ 10^{6}\ /\ 6228.85\ *\ 10^{-6}\ =\ 321086557\ operations/sec$$

3- The Multithreads Approach: Exploits parallelism to maximize CPU usage. It divides the answer matrix into **K** row blocks, enabling individual threads of only 1 main process to compute specific blocks concurrently. These computed blocks are then recombined to form the resultant matrix, optimizing CPU utilization efficiency.

Procedure:

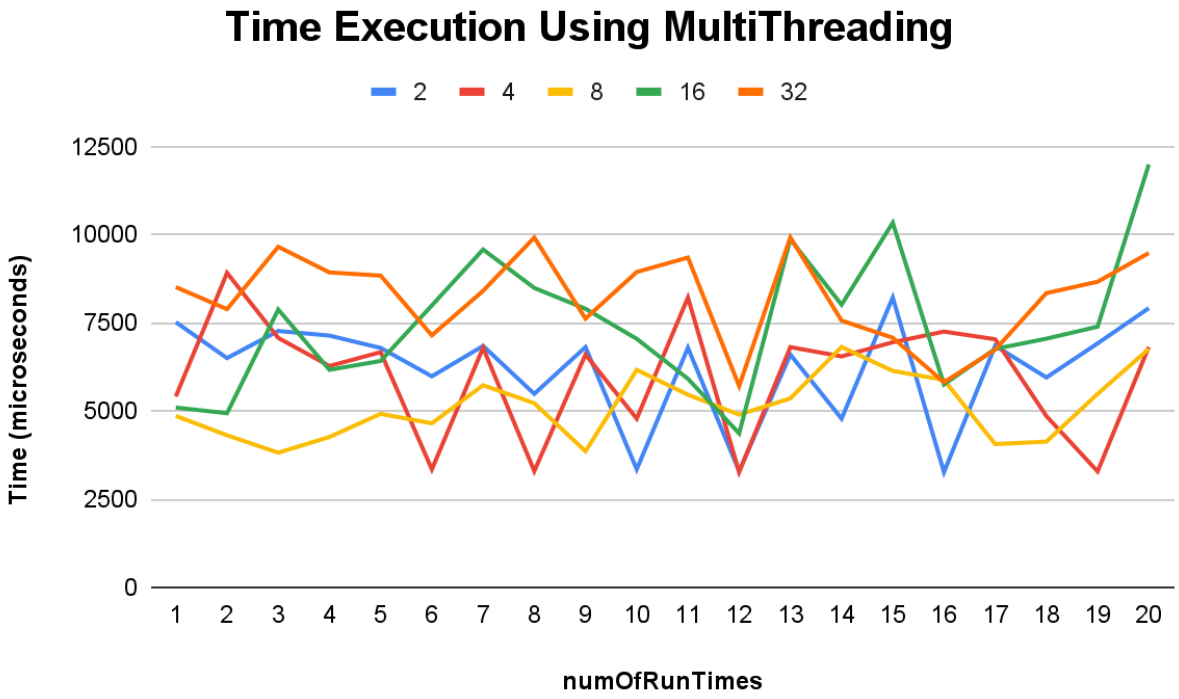
1) Declare two global 2D arrays that are intended for multiplication, with an array to hold the resulting values.
2) Determination of the length of the block where it equals:
$$blockLength\ =\ 100\ /\ K$$
3) Identify an array of threads using `pthread_t`.
4) Initiate a loop that assigns a distinct array to each thread, containing the current interval of the **K**-th block, to be passed into the function executed by the threads. These arrays will be deallocated using the `free` function after execution.
5) Sequentially create threads using `pthread_create`, specifying attributes and the function each thread will execute.
6) Repeat blocks one after another until reach the K-th block.
7) When reaching the K-th block, steps previously done in the multiprocessing approach will be repeated. Where the block limit of the current block may not equal the previous ones since the number K couldn't be divisible by the number of rows (n = 100). So, with no doubt,
$$(Kth\ blockLength\ =\ 100\ -\ lastBlockEnd)\ >=\ blockLength$$
8) Invoke the `pthread_join` function for each thread to ensure it completes its calculation before continuing.
9) Measure the execution time.

A graphical representation illustrating the functionality of joinable threads.

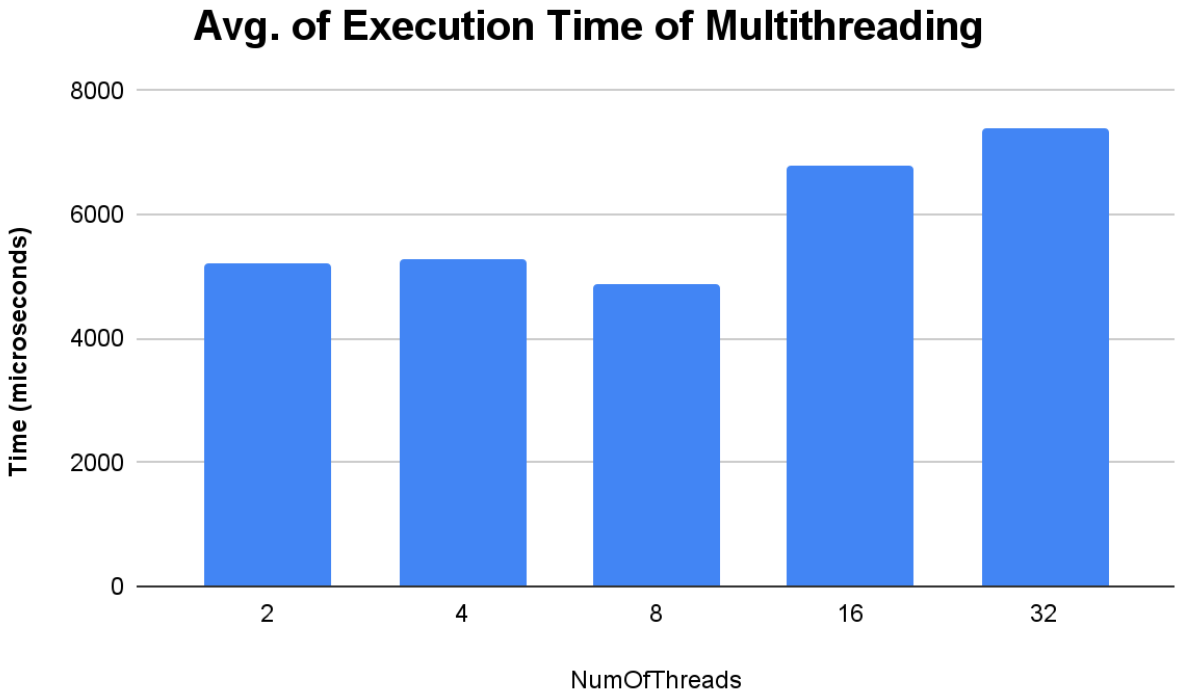


Initially, increasing the number of threads might lead to improved execution time. However, there comes a point where adding more threads doesn't consistently reduce execution time, mainly due to increased overhead and needing more time to keep switching between contexts of the threads. This is influenced by several factors, such as the count of internal CPUs, the number of cores, and the frequency of each CPU, among other considerations.

To calculate the execution time of using multithreading we use almost the same steps we used in the multiprocessing section.



The average duration required using 8 *threads to obtain the solution is* $= 4868.025\ \mu s$



Throughput for MultiThreading approach when using 8 *threads* $=$

$2 * 10^{6} / 4868.025 * 10^{-6} =$ 410844234 *operations/sec*

But, why do we use threads when they seem to offer similar functionalities as processes? Threads and processes are both fundamental units of execution in computing, but they serve different purposes and have distinct advantages.

MultiProcessing vs. Multithreading

Standards	Multiprocessing	Multithreading
Execution Unit	Each process has its own memory space and resources.	Threads share the same memory space and resources of the parent process.
Resource Utilization	requires more system resources as each process has its own memory allocation and resources.	More lightweight compared to multiprocessing as threads share resources and memory within the same process, reducing overhead.
Communication and Synchronization	Inter-process communication mechanisms (like pipes or shared memory) are needed for communication between different processes.	Threads within the same process can communicate and share data directly.
Scalability and Performance	Scales well with multiple cores or processors. Ideal for CPU-bound tasks that can benefit from parallel processing.	Suitable for I/O-bound tasks where threads can perform other operations while waiting for I/O. Might not scale as efficiently with multiple cores due to various reasons.

Complexity and Overhead	Higher start-up and memory overhead due to the creation of separate processes.	Lower overhead compared to multiprocessing but requires careful synchronization and coordination to manage shared resources effectively to avoid race conditions.
Programming Ease	Generally, it involves more complex programming due to the need for inter-process communication and management.	Easier to implement and manage within a single process, but requires careful consideration for synchronization to avoid race conditions.

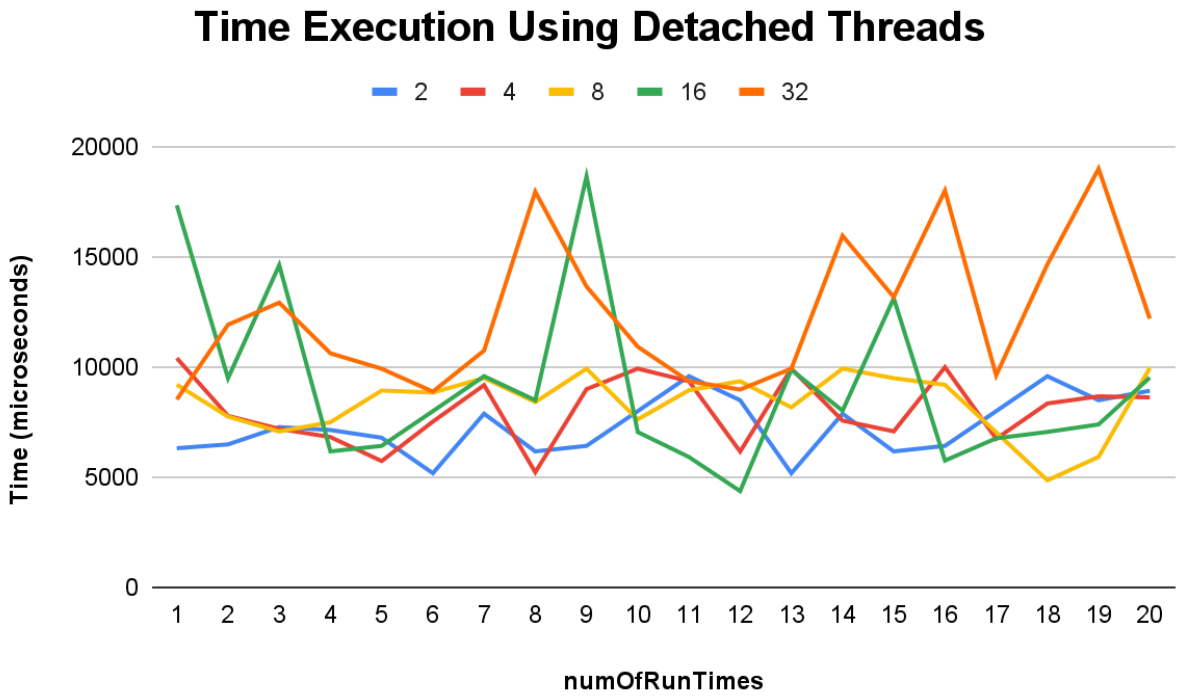
Detached Threads

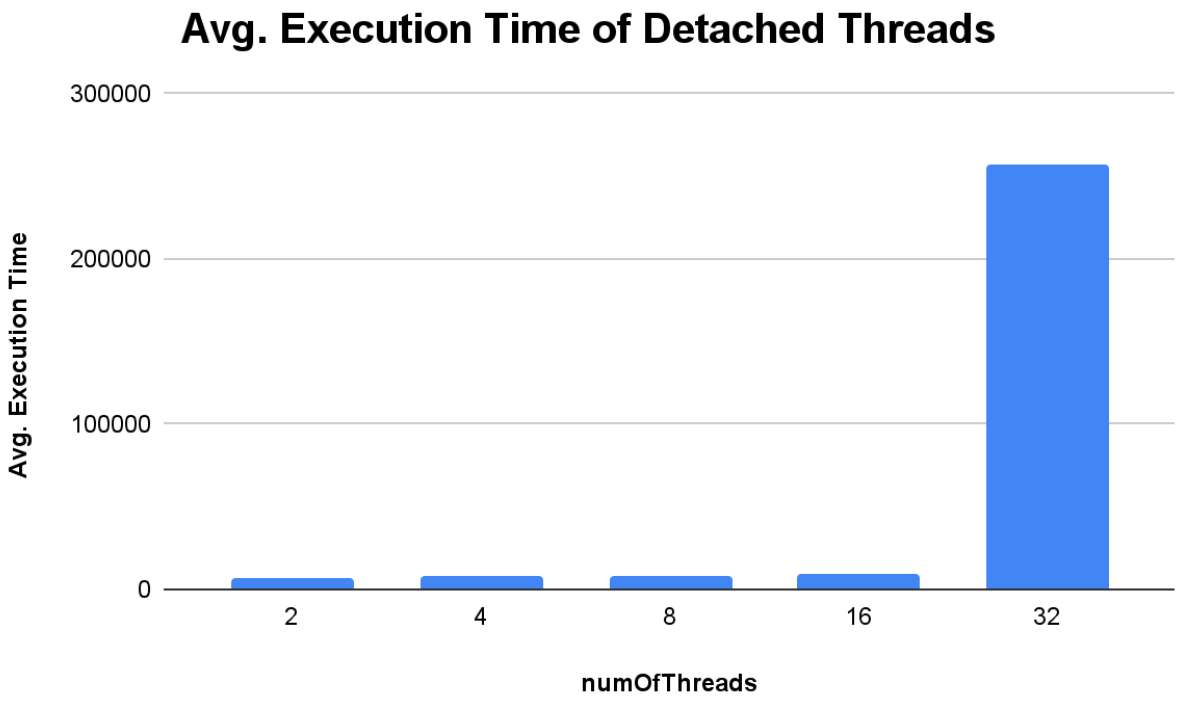
Previously we've talked about threads and specially the joinable ones. Where there is another type of threads called **Detached Threads.**

Where difference between joinable threads and detached ones lies in the handling of the main program's lifecycle:

Joinable Threads: Main program waits for these threads to complete their execution before terminating. It's akin to synchronizing the main program's flow with the threads, ensuring they finish before the program concludes.

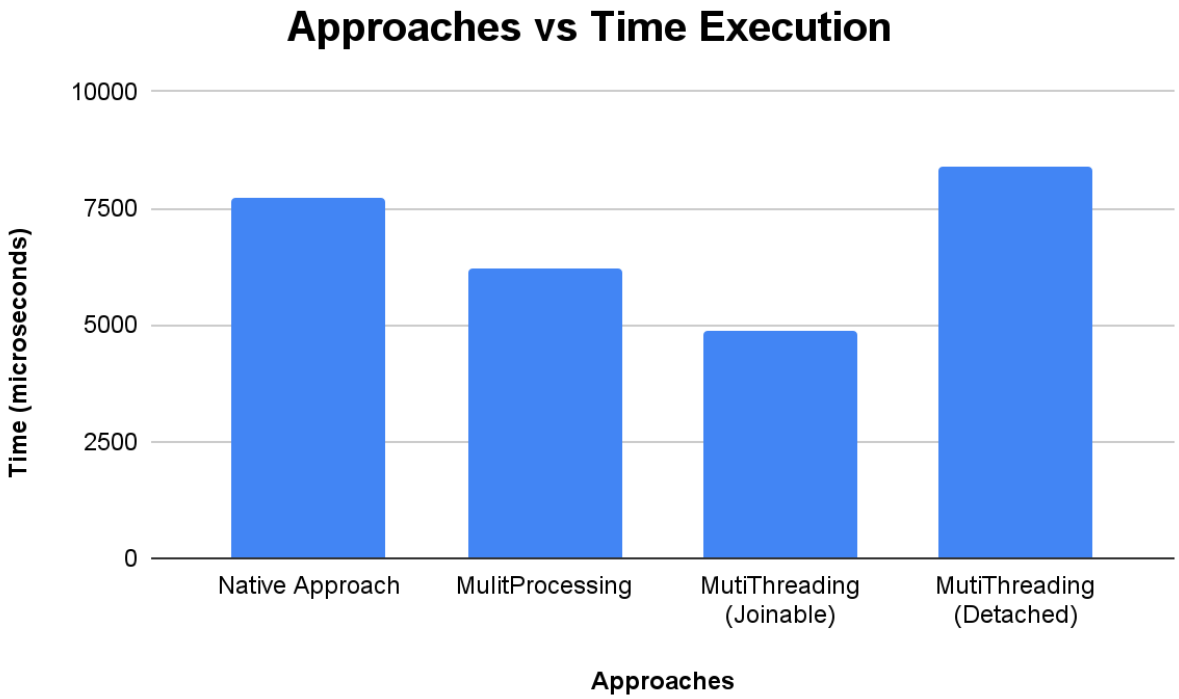
Detached Threads: These threads run independently, and the main program doesn't wait for their completion. Once the main program finishes its execution, detached threads might continue their operation or terminate suddenly, having no impact on the main program's flow.

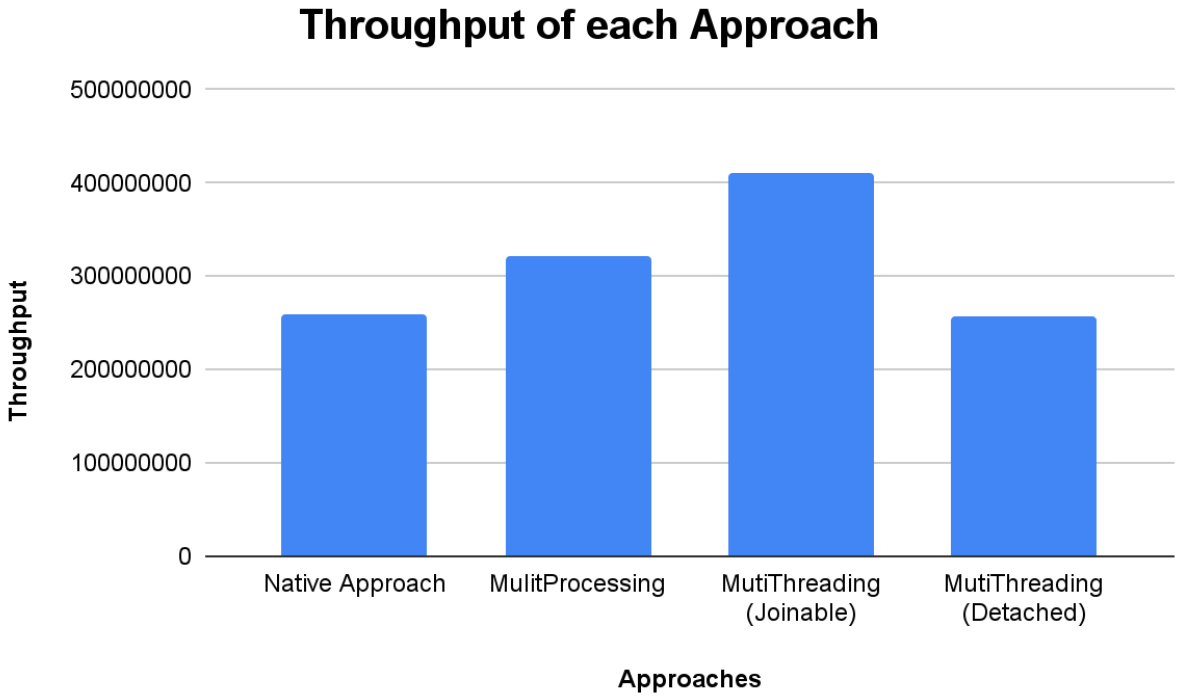




The significant disparity in the execution time of detached threads could be attributed to the approach employed to determine whether the thread has completed its execution or not.

$$Throughput\ for\ MultiThreading\ approach\ \ when\ using\ 8\ detached\ threads\ = 2\ *\ 10^{6}\ /\ 7771.3\ *\ 10^{-6}\ =\ \ 257357199\ operations/sec.$$





As depicted in the preceding chart, it's clear that leveraging parallelism, particularly through joinable multithreading, proves to be the most efficient method for matrix multiplication. However, it's important to acknowledge that these experiments are subject to various factors, including the computer's number of cores, changes in CPU states (potentially resulting in occasional inaccuracies in measurements) and external techniques employed to support the C program.

---

conclusion

In conclusion, the exploration of parallelism through different methodologies (utilizing single processes, multiprocessing, and both joinable and detached multithreading) reveals distinct performance characteristics and operational efficiencies.

The use of a single process, while straightforward, lacks the optimization potential offered by parallel computing. Although simple to implement, it may underutilize available hardware resources, leading to longer execution times for computationally intensive tasks.

Multiprocessing demonstrates the power of concurrent execution by dividing tasks among multiple processes. This approach maximizes CPU utilization, leveraging the available cores for enhanced efficiency. However, managing inter-process communication (IPC) and synchronization adds complexity to the implementation.

Where in multithreading, joinable threads synchronize the main program's flow, ensuring it awaits thread completion. This synchronization allows for coordinated parallelism, providing control over when to retrieve results. On the other hand, detached threads operate independently, releasing the main program from waiting for their completion. While offering potentially faster execution, managing detached threads demands careful consideration to avoid abrupt terminations and resource issues.

Each method (single process, multiprocessing, joinable, and detached multithreading) presents a unique trade-off between simplicity, performance gains, and control. The choice among these approaches hinges on the nature of the task,

resource availability, synchronization needs, and the level of parallelism attainable, ultimately dictating the most suitable strategy for optimizing task execution.

For tables which show the informations about time execution and throughput in details: [Google Sheet]