Work Assignment - Phase 3 (Final)

1st João Brito
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg53944@alunos.uminho.pt

2nd Paulo Oliveira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg54133@alunos.uminho.pt

***Abstract*—This final assignment focuses on enhancing the parallel implementation of a case study, offering students the choice between optimizing the existing OpenMP implementation, designing a new version for GPU accelerators using CUDA, or implementing a distributed memory version in C with MPI. The primary objective is to reduce execution time while ensuring the correctness of parallelism and addressing potential *data races* issues. A key addition to this phase is the incorporation of a comprehensive set of tests for performance assessment, encompassing various input sizes, machine configurations, and scalability analyses.**

***Index Terms*—CPU, CUDA, threads, parallel, Kernel**

I. Introduction

In the context of the Parallel Computing course, the final phase of the practical work focused on expanding the concepts of parallelism initially addressed in the previous phase. In the first phase of the project, the group's focus and objective were to improve the application's performance by optimizing the sequential version of the project as much as possible, aiming for the minimum total instructions and the shortest possible execution time.

In the second phase of the project, the group was tasked with developing a parallelized version of the sequential version to enhance the model's performance by dividing the work performed in each iteration among threads. This parallelization method required careful consideration of concurrency areas in the code between threads to avoid *data races* and ensure that the obtained result matched the expected outcome. The group then utilized **OpenMP** directives to ensure correct code parallelization and achieve the expected results and improved performance.

In this final phase of the project, to deepen our knowledge of parallelism, we decided to design and implement a new version for accelerators using **CUDA** on **GPU**. For the implementation of this phase and the formulation of a functional logic, it was important to consider the number of blocks used, the number of threads per block, as well as the work performed by each thread and their reuse for better performance with a minimum number of threads.

II. Analysis of previous assignments

In the initial stages of our analysis of the molecular dynamics simulation code, our group encountered challenges in improving its execution time. The decision to add local variables to functions did not yield the expected performance boost. Consequently, a shift in focus was imperative, leading us to utilize gprof functionalities and identify potential bottlenecks in the functions ”potential()” and ”computeAccelerations()”.

The optimization strategies adopted were rooted in careful consideration of compilation flags and their impact on performance. Flags such as O3, Ftree-vectorize, fno-omit-frame-pointer, ffast-math, mavx, march=native, and pg were selected after thorough testing and evaluation. These flags proved instrumental in enhancing the program's overall efficiency, especially in the targeted functions.

A pivotal aspect of our optimization approach was the reevaluation of mathematical expressions within the identified functions. The substitution of computationally expensive pow and sqrt functions with simpler arithmetic operations not only reduced execution time but also improved code clarity. Furthermore, the introduction of local variables for expressions with constant repetitions and the optimization of array insertions contributed to the overall efficiency.

The incorporation of loop unrolling, a technique aimed at reducing loop control overhead and increasing work done in each iteration, proved to be a strategic move. This technique, coupled with the merging of the ”potential()” and ”computeAccelerations()” functions, led to a substantial reduction in execution time. The creation of a global variable, PEE, facilitated the consolidation of results without compromising accuracy.

In the results analysis, we ensured the consistency of our optimizations by comparing the obtained results with the initial ones. The command ”diff cp-output.txt calculo1-output.txt” confirmed the equality of results up to 12 numerical and decimal places. Additionally, a significant improvement in execution time and a notable reduction in the number of instructions demonstrated the effectiveness of our optimization efforts.

In summary, the strategic decisions made and implementations carried out in this initial phase yielded substantial improvements in program performance, successfully achieving the objective of significantly reducing execution time.

In the second phase of our project, n this phase of the project, the group focused on addressing application bottlenecks, specifically targeting hot-spots – portions of the code consuming disproportionate computational resources. Thorough logging and testing revealed that the ”computeAccelerations()” function presented a significant challenge in terms of resource consumption.

To tackle these identified hot-spots, the group opted for shared memory parallelism using **OpenMP** directives. The exploration of **OpenMP** directives, including omp parallel for, reduction(), and others, provided insights into scalable parallel programming. The primary objective was to ensure efficient resource utilization while mitigating issues such as data inconsistencies and race conditions.

The implementation of **OpenMP** directives yielded promising results in terms of execution times. As the number of threads increased, there was a notable improvement until reaching a peak at 20 threads. However, beyond this point, a slight increase in execution time indicated a potential saturation point in scalability. Factors such as uneven distribution of work and synchronization overhead were considered in this analysis.

Amdahl's law played a crucial role in highlighting the impact of non-parallelizable sections, particularly in the computeAcceleration() function. The sequential execution of mathematical calculations in this function limited the overall scalability of the program.

In conclusion, this phase showcased the application of parallel programming concepts through **OpenMP** directives. Despite efforts to optimize the parallel solution, the observed scalability suggested inherent limitations in the program's structure, notably in the computeAcceleration() function. The group recognizes the need for further exploration and refinement to achieve optimal scalability.

III. Changes before Implementation

Given the objective of improving and extending the parallelized solution from the previous phase, the initial focus returned to code analysis and identification of code regions that would require careful attention to concurrency and performance. In this way, we identified two regions that needed rectification:

- Sequential Region: where the focus was on parallelizing this region, taking due care to avoid *data races*. This region includes the function:
  - computeAccelerations: responsible for calculating accelerations and the PEE variable for subsequent calculations.
- Parallel Region: where the attention was on extending and better conceptualizing the previously implemented implementation, making use of expanded parallelism concepts.

With this scope, the goal was to parallelize the computeAccelerations algorithm completely, abandoning the concepts implemented in **OpenMP** in the previous phase in favor of adopting the **CUDA** environment. The reasons for this approach were the fact that this platform allows harnessing the immense parallelism brought by the **GPU** in terms of number processing, as well as gaining experience with new tools.

IV. Implementation

Once using the **CUDA** environment, as mentioned earlier, we had to pay attention to the two factors that helped in the implementation of this phase, namely the number of blocks used and the number of threads per block. With this, we divided this implementation into several components, emphasizing the following topics:

- Change of data structures;
- Elimination of sequential regions;
- Invocation of *Kernels*. using **CUDA**;

A. Change of Data Structures

To improve the organization and performance of the sequential code, we decided to change the two-dimensional arrays for acceleration and position and use one-dimensional arrays to represent them in the **GPU** environment. The choice of using these one-dimensional arrays is due to greater simplicity in terms of memory access, requiring only one index to retrieve an element, due to space savings, and an improvement in application performance by reducing complexity in accessing these arrays. However, these proved significantly more complex to implement in terms of search indices.

B. Elimination of sequencial regions

Given the **CUDA** approach, we now need to keep in mind the appropriate use of the number of blocks and the number of threads per block to mitigate sequential regions and avoid *data races*. Thus, the logic used involves initially sending the following data structures to the **GPU**:

- N: number of particles;
- sigma: global variable used in calculations;
- arrayRGPU: array containing positions derived from the global array r;
- arrayAGPU: array containing accelerations derived from the global array a;
- POTGPU: variable that stores the calculated Pot value in **GPU**.

Subsequently, the *computeAccelerations* algorithm runs entirely in parallel on the **GPU**, thanks to the implementation of three *Kernels*..

- **setAccelerationsKernel**: *Kernels* responsible for initializing the arrayAGPU to 0 on the **GPU**;
- **computeAccelerationsKernel**: *Kernel* responsible for performing acceleration and Pot variable calculations and storing them for later copying to the **CPU**;
- **launchComputeAccelerationsKernels**: *Kernel* responsible for copying global arrays and variables from the CPU to the **GPU** and vice versa, and for calling the remaining *Kernels*.

C. Kernel setAccelerations

In the invocation of this *Kernel*, the total number of threads called is equal to the number of blocks multiplied by the number of threads per block. Each thread handles a particle, with the thread's id as the id, and the index traversed along the array, with each thread accessing three positions and setting them to zero. To simplify the logic, we separated this cycle from the rest of the *computeAccelerations* function implementation to perform this function in parallel, improving its performance.

D. *Kernel computeAccelerations*

In the initial phase of the project, we considered the best way to use the fewest possible threads to achieve the best performance results.

With this in mind, we opted for the use of MapReduce, a programming technique used to process large datasets in distributed environments. This approach allowed us to efficiently parallelize and distribute tasks in a computer cluster. We divided our initial *computeAccelerations* function into two phases: Map and Reduce.

In the first stage of the process and in the Map phase, each *thread* divided each particle representing three positions of the one-dimensional array, with each *thread* adding accelerations and storing them in the arrayAGPU. We noticed that each *thread* would perform a different workload depending on the array's position, where initial *threads* would compare with all other particles, but final *threads* would make fewer comparisons, leading to an unbalanced workload and possible *thread* reuse.

In the second stage of the process and in the Reduce phase, we used *thread* to traverse arrays with partial results performed by each *thread* and add all these partial calculations to produce the final results. However, with this logic and all the improvements we could make, the performance results were not as desired, hovering around 24 seconds.

Despite considering this version the best in terms of scalability with the best implementation logic, we decided we had to look for an implementation where the execution time of the function would come down to the expected value. With this in mind, we decided to use ”atomicAdd” to produce our final version of the implementation. ”atomicAdd” is an atomic operation often used in parallel programming, especially when multiple *thread* or processes can access and modify a shared variable simultaneously. This function is used to perform an atomic addition, ensuring that no other *thread* interferes with the value during the operation. Atomicity in this context means that the operation is executed as a single instruction without interruptions.

After finding out about this atomic function and implementing it properly, we decided to use it to reduce the solution's complexity compared to the previous MapReduce version. We then used this function to make the atomic additions of the array a in the **GPU** environment, the led our implementation to show a performance improvement from 24 seconds to 11 seconds.

E. *Kernel launchComputeAccelerationsKernel*

In the invocation of this *Kernels* after a previous memory allocation of variables and global arrays on the **GPU**, this *Kernels* copies the values of arrays a and r to the previously allocated arrays in **GPU** memory and calls the *setAccelerationsKernel* and *computeAccelerationsKernel Kernels*. After these calls and the resolution of the necessary calculations, this *Kernels* copies the values stored in **GPU** from the Pot variable and arrayAGPU back to the **CPU** for their recording in **CPU** memory

V. **CPU** E **GPU**

In addition to the previously detailed explanation, it is worth highlighting the role and differences regarding the use of the **CPU** and the use of the **GPU**, as both are employed for different purposes.

- **CPU**: Used to allocate memory in data structures, as well as generate the initial information for each of these structures. Responsible for allocating sufficient space and copying to the **GPU** all data related to data structures, such as accelerations and particle positions. Performs iterations and corresponding calls to the *Kernels*.
- **GPU**: Used to compute all the information provided by the **CPU**. It is where the core of the algorithm processing takes place, in a parallelized manner, ensuring the absence of *data races*. It is invoked twice, as we have two *kernels* in each iteration of the algorithm.

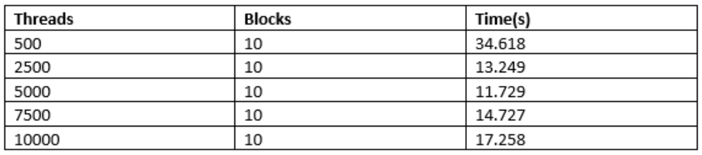
VI. TESTING AND SCALABILITY

To assess the magnitude of the various previously mentioned parameters, appropriate tests were conducted, along with the corresponding theoretical analysis of the program's performance concerning the range of parameter values. The evaluations for the following scenarios were performed:

- Test 1: Fixed sample size, varying the total number of threads.
- Test 2: Sample size varied (N) , with the total number of threads fixed.

A. *Test 1*

For this test, as described above, several experiments were conducted with a fixed N = 5000, N = 7500 and N = 10000 varying the total number of threads.



Threads	Blocks	Time(s)
500	10	34.618
2500	10	13.249
5000	10	11.729
7500	10	14.727
10000	10	17.258

Fig. 1. Results for N=5000

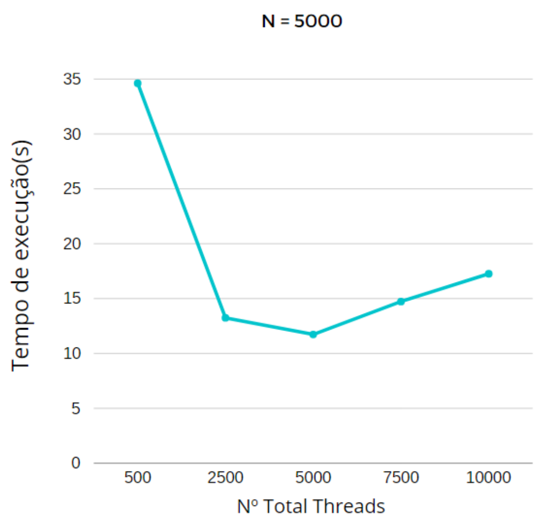


Fig. 2. Graph for N=5000

Threads	Blocks	Time(s)
2500	10	18.734
5000	10	16.334
7500	10	16.417
10000	10	21.103

Fig. 3. Results for N=7500

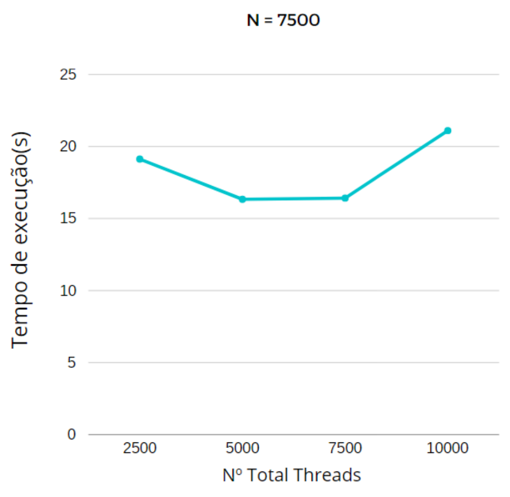


Fig. 4. Graph for N=7500

Here we can see the tables of Time values regarding each number os particles table and each ammount of threads as well as their corresponding Graph. Given the metrics and values obtained through these tests in this aspect, we can understand that, for a sample of a given size, there exists an upper and lower limit regarding the number of threads to use. Therefore, we recognize that increasing the number of threads aims to improve performance and reduce execution time, but without imposing overhead through an excess of threads.

Threads	Blocks	Time(s)
7500	10	27.238
5000	10	29.753
10000	10	28.504

Fig. 5. Results for N=10000

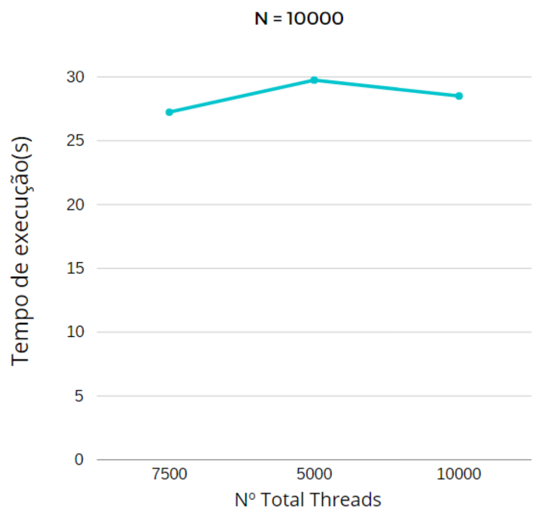


Fig. 6. Graph for N=10000

B. Test 2

For Test 2, experiments were conducted with varied dimensions of the variable N and a constant number of threads to investigate whether there is any relationship between two variables of different sizes (in this case, 4 times larger). The goal was to determine if the optimal number of threads for one variable N corresponds to the optimal number of threads for another variable N.

N size	Threads	Blocks	Time(s)
5000	5000	10	11.729
7500	5000	10	16.335
10000	5000	10	29.753
15000	5000	10	49.462
20000	5000	10	82.69

Fig. 7. Thread 5000 Table Results

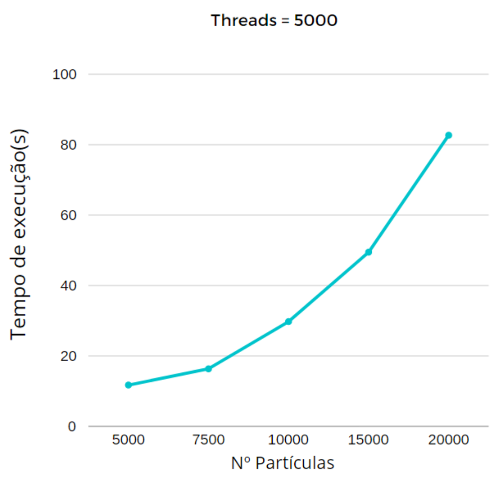


Fig. 8. Thread 5000 Graph Results

Utilizing a consistent number of threads across samples of varying dimensions introduces a nuanced scenario where the impact on performance may manifest as either improvement or degradation. The intricacies of the relationship between sample size and the optimal thread count are complex, and the group recognizes the need for a nuanced approach. It is acknowledged that, beyond a certain size, the reliability of results might diminish. This raises considerations about the scalability of the chosen thread configuration and prompts a deeper exploration into the optimal balance between thread count and sample dimensions to ensure consistent and dependable performance outcomes.

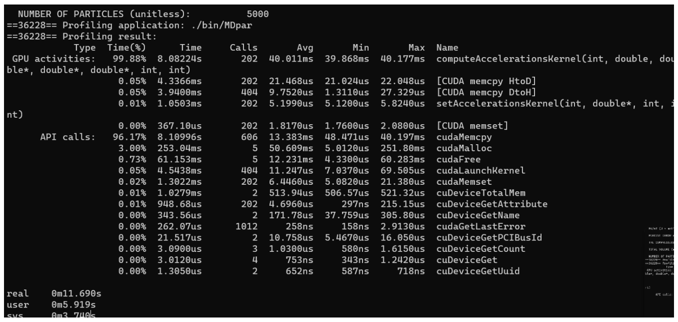
Regarding the developed solution, in a general scope of scalability, we can understand that the number of threads to be used should take into account the sample size and the quantity of *clusters*. Additionally, for relatively small dimensions, we should use a small number of threads to avoid imposing unnecessary overhead from too many threads. With larger sample sizes, choosing an appropriate number of threads is essential, always considering not to exaggerate this number given the overhead observed in test scenarios.

It is important to emphasize that for dimensions larger than those tested, we cannot confidently predict that the implemented solution is suitable and provides scalability. The number of threads to be used remains an aspect that cannot be precisely foreseen in such cases.

VII. Comparative Performance Analysis: MapReduce vs. Atomic

Following, there are two illustrative images depicting the **GPU** occupation of the MapReduce version and the Atomic version.

In the Atomic version, we can observe that it occupies 99.8% of its time, equivalent to 8.082224 seconds, in functions related to ”computeAcceleration”.

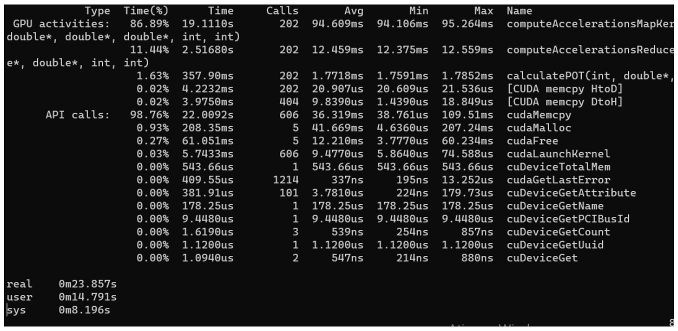


```
  NUMBER OF PARTICLES (unitless):          5000
==36228== Profiling application: ./bin/MDpar
==36228== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.88%  8.08224s       202  40.011ms  39.868ms  40.177ms  computeAccelerationsKernel(int, double, dou
ble*, double*, double*, int, int)
                    0.05%  4.3366ms       202  21.468us  21.024us  22.048us  [CUDA memcpy HtoD]
                    0.05%  3.9400ms       404  9.7520us  1.3110us  27.329us  [CUDA memcpy DtoH]
                    0.01%  1.0503ms       202  5.1990us  5.1200us  5.8240us  setAccelerationsKernel(int, double*, int, i
nt)
                    0.00%  367.10us       202  1.8170us  1.7600us  2.0800us  [CUDA memset]
      API calls:   96.17%  8.10996s       606  13.383ms  48.471us  40.197ms  cudaMemcpy
                    3.00%  253.04ms         5  50.609ms  5.0120us  251.80ms  cudaMalloc
                    0.73%  61.153ms         5  12.231ms  4.3300us  60.283ms  cudaFree
                    0.05%  4.5438ms       404  11.247us  7.0370us  69.505us  cudaLaunchKernel
                    0.02%  1.3022ms       202  6.4460us  5.0820us  21.380us  cudaMemset
                    0.01%  1.0279ms         2  513.94us  506.57us  521.32us  cuDeviceTotalMem
                    0.01%  948.68us       202  4.6960us     297ns  215.15us  cuDeviceGetAttribute
                    0.00%  343.56us         2  171.78us  37.759us  305.80us  cuDeviceGetName
                    0.00%  262.07us      1012     258ns     158ns  2.9130us  cudaGetLastError
                    0.00%  21.517us         2  10.758us  5.4670us  16.050us  cuDeviceGetPCIBusId
                    0.00%  3.0900us         3  1.0300us     580ns  1.6150us  cuDeviceGetCount
                    0.00%  3.0120us         4     753ns     343ns  1.2420us  cuDeviceGet
                    0.00%  1.3050us         2     652ns     587ns     718ns  cuDeviceGetUuid

real    0m11.690s
user    0m5.919s
sys     0m3.740s
```

Fig. 9. Atomic Performance

In the MapReduce version, we can observe that it occupies 86.89%+11.44& = 98.33% of its time, equivalent to 21.6278 seconds, in functions related to ”computeAcceleration”.



```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   86.89%  19.1110s       202  94.609ms  94.106ms  95.264ms  computeAccelerationsMapKer
double*, double*, double*, int, int)
                   11.44%  2.51680s       202  12.459ms  12.375ms  12.559ms  computeAccelerationsReduce
e*, double*, int, int)
                    1.63%  357.90ms       202  1.7718ms  1.7591ms  1.7852ms  calculatePOT(int, double*,
                    0.02%  4.2232ms       202  20.907us  20.609us  21.536us  [CUDA memcpy HtoD]
                    0.02%  3.9750ms       404  9.8390us  1.4390us  18.849us  [CUDA memcpy DtoH]
      API calls:   98.76%  22.0092s       606  36.319ms  38.761us  109.51ms  cudaMemcpy
                    0.93%  208.35ms         5  41.669ms  4.6360us  207.24ms  cudaMalloc
                    0.27%  61.051ms         5  12.210ms  3.7770us  60.234ms  cudaFree
                    0.03%  5.7433ms       606  9.4770us  5.8640us  74.588us  cudaLaunchKernel
                    0.00%  543.66us         1  543.66us  543.66us  543.66us  cuDeviceTotalMem
                    0.00%  409.55us      1214     337ns     195ns  13.252us  cudaGetLastError
                    0.00%  381.91us       101  3.7810us     224ns  179.73us  cuDeviceGetAttribute
                    0.00%  178.25us         1  178.25us  178.25us  178.25us  cuDeviceGetName
                    0.00%  9.4480us         1  9.4480us  9.4480us  9.4480us  cuDeviceGetPCIBusId
                    0.00%  1.6190us         3     539ns     254ns     857ns  cuDeviceGetCount
                    0.00%  1.1200us         1  1.1200us  1.1200us  1.1200us  cuDeviceGetUuid
                    0.00%  1.0940us         2     547ns     214ns     880ns  cuDeviceGet

real    0m23.857s
user    0m14.791s
sys     0m8.196s
```

Fig. 10. Map Performance

Both show the majority of their time dedicated to functions responsible for calculations related to the ”computeAcceleration” function. However, there is a difference in execution time between the two, with the Atomic implementation taking 11.690 seconds, while the MapReduce version takes 23.857 seconds. This outcome reinforces the notion that, in the context of **GPU** utilization for compute-intensive tasks, fine-tuned approaches such as Atomic operations can yield substantial improvements in execution time.

VIII. Conclusion and Reflection on the overall project

The journey from the initial phases of sequential code optimization to the final implementation on **GPU** architectures has been both challenging and enlightening. Reflecting on the overall project, several key insights emerge:

- **Evolution of Parallel Computing Understanding**: The project allowed for a deepening understanding of parallel computing concepts. The transition from optimizing sequential code to implementing parallel solutions, first with **OpenMP** and later with **CUDA**, provided valuable hands-on experience.
- **Adaptability and Problem-Solving**: The necessity to adapt and refine the approach based on performance evaluations and challenges encountered showcased the importance of problem-solving skills in the field of

parallel programming. Flexibility in adopting different parallelization strategies was a crucial aspect of the learning process.

- **Team Collaboration and Communication**: Collaborating within the group underscored the significance of effective communication and teamwork. Discussing ideas, sharing insights, and collectively addressing challenges facilitated a more comprehensive and robust approach to problem-solving.
- **Recognition of Scalability Challenges**: The scalability challenges encountered, particularly in identifying optimal thread counts and addressing bottlenecks, highlighted the complexity of achieving optimal performance in parallel computing. These challenges underscore the ongoing need for refinement and exploration in large-scale computational tasks.
- **Continuous Learning and Application**: The project reinforced the notion that parallel computing is a continuously evolving field. Embracing a mindset of continuous learning and application of new technologies and techniques is essential for staying at the forefront of advancements in parallel programming.

As the project draws to a close, the group recognizes the strides made in the pursuit of optimized parallel solutions. While the achieved results in terms of execution time may not stand as the most optimal, they stand testament to the collective effort and unwavering persistence invested in showcasing the best possible outcomes. The lessons learned throughout this endeavor underscore the intricate nature of parallel computing, emphasizing the commitment to continuous improvement and the anticipation of further exploration in this dynamic realm. The group remains optimistic about the exciting potential for refinement and enhanced performance in future iterations, fueled by a dedication to pushing the boundaries of parallel computing capabilities.