Work Assignment 3 - Improvement of the execution time of the previous WA using CUDA

Hugo Marques
Universidade do Minho
Braga, Portugal
pg47848@alunos.uminho.pt

Nuno Mata
Universidade do Minho
Braga, Portugal
pg44420@alunos.uminho.pt

***Abstract*—This paper extends the previous work assignments, namely, optimizing a single-threaded Molecular Dynamics Simulation program, previously enhanced through various optimization techniques in the first phase and in the second phase, take up the work that was done and implement OpenMP directives in order to try to improve the program's execution times. The main focus of this final work is to design and implementation of High Performance Computing technologies, specifically the use of CUDA (Compute Unified Device Architecture) to explore the destruction potential of GPUs. The goal is to overcome performance limitations in previous phases, where the program was optimized with multi-threaded programming techniques using OpenMP.**

In this report, we detail how we adapted the code to operate efficiently on GPU architectures, addressing challenges such as memory management, parallelizing intensive calculations, and avoiding data races. Specific optimizations implemented include parallelization of intermolecular force calculations, position and velocity updates, and kinetic and potential energy calculations.

The results demonstrate significant improvements in execution times, highlighting the positive impact of GPU computing on the efficiency of molecular dynamics simulations. Performance comparison with previous versions of the program illustrates the potential of GPUs to accelerate computationally intensive applications.

***Index Terms*—Optimization, OpenMP, Parallel, CUDA, kernel**

I. Introduction

In this final work the goal of the project is to improve computational efficiency through parallel processing, reducing time execution, using GPUs accelerators (GPU with CUDA). The first phase laid the groundwork by improving the algorithmic and software implementations of a single-threaded MD simulation, focusing on the optimization of a program that inherently involves complex and computationally demanding tasks. These simulations, crucial for understanding atomic and molecular behavior over time, have proven to be intensive in terms of computational resources.

In this report, we address the optimization of a molecular dynamics simulation program that was initially developed as a single-threaded application. In previous work, the program was improved through optimization and parallelization techniques, including the implementation of OpenMP directives to improve performance on multi-threaded architectures. At this stage, our goal is to further explore the potential of hardware acceleration, specifically through the use of GPUs and the CUDA architecture, to overcome performance limitations still present after previous optimizations.

The introduction of GPUs in computing turns out to be important to deal with the computational intensity of molecular dynamics simulations in a more efficient way, as they have a massively parallel architecture and are particularly suitable for accelerating calculations that can be effectively distributed across thousands of threads. A adaptação do código que fizemos para esta nova arquitetura envolve desafios significativos, incluindo a gestão eficiente da memória da GPU, a paralelização de algoritmos e a garantia de integridade dos dados num ambiente de execução paralela.

This report aims to detail the process of adapting the molecular dynamics simulation program to run on GPUs, including restructuring the code, implementing parallelization techniques specific to CUDA, and optimizing performance. Furthermore, we present a comparative analysis of the performance between the versions previously optimized with OpenMP and the new version with CUDA, indicating the improvements achieved.

A. Common MD Simulation Techniques

Optimization techniques in molecular dynamics (MD) simulations are critical for improving computational efficiency and accuracy. Here are some common methods used to optimize MD simulations:

- **Parallelization:** Using multiple processors or GPUs to distribute the computational load. This includes using high-performance computing (HPC) clusters and cloud computing resources.
- **Integration Algorithms**: Choosing appropriate time integration algorithms like Verlet or Leapfrog algorithms for better stability and efficiency.
- **Load Balancing**: In parallel computations, ensuring equal distribution of computational load across processors to avoid bottlenecks.
- **Use of Specialized Hardware**: Utilizing hardware optimized for MD simulations, such as GPUs as shown in this project.
- **Software Profiling and Optimization**: Analyzing the performance of the simulation code to identify and optimize bottlenecks.

B. OpenMP vs CUDA

There are numerous differences between parallelization done with OpenMP and parallelization done with OpenMP. It is important to understand the fundamental differences in their parallelization approaches, target hardware architectures, and the types of applications for which each is a best choice. Let's explore these differences:

- **Programming Model:** With OpenMP, are used compilation directives to parallelize blocks of code and it automatically manages thread allocation and distribution of work between them. In the other hand, with CUDA, it requires writing kernels, which are functions executed on the GPU and provides greater control over parallelization and performance optimization on the GPU.
- **Types of Tasks and Applications:** With CUDA, it is ideal for applications with a high degree of parallelism and computational intensity like this case with the MD simulations program. With OpenMP, it is suitable for parallelizing existing applications with less restructuring effort and ideal for applications that benefit from parallelism at the multicore level, but still have significant sequential execution components.
- **Scalability and Performance** It is more scalable using CUDA which is capable of efficiently handling large volumes of data and calculations, but it requires a good understanding of GPU architecture and specific optimization techniques. Using Open MP, it has scalability limited by the number of cores available on the CPU.
- **Target Hardware Architecture** With CUDA it is designed for programming on GPUs, which are ideal for high performance computing and best for tasks that can be run on parallel on a large scale. With OpenMP, it is intended for parallelization in CPUs, which have less cores than GPUs, but each core is more powerful and it is ideal for tasks that require less parallelism.

C. Analysis of optimizations with CUDA

The optimization of the MD simulations program with CUDA was carried out in order to improve the code performance in terms of time execution and make it more scalable. Several techniques were used to achieve this goal such as:

- **Efficient Random Number Generation with CURAND**

The code parallelized with CUDA uses the **CURAND** library to generate random numbers directly on the GPU. This technique allowed us to reduce the overhead associated with generating random numbers on the CPU and subsequently transferring them to the GPU.

- **GPU Memory Optimization**

Allocates and manages data directly in GPU memory, minimizing data transfers between the CPU and GPU, which allowed to take advantage of high GPU memory bandwidth.

- **Thrust Library for Reductions and Complex Operations**

The Thrust library was used to perform reduction operations and other complex operations, allowing the use of GPU-optimized algorithms without having to implement them manually.

- **Configuration of Threads and Blocks**

We chose to manually configure the number of threads per block and the number of blocks based on the size of the problem. This ultimately ensures an efficient use of GPU resources,

- **Using Shared Memory and Registers on the GPU**

The use of shared memory for data frequently accessed or shared by threads within a block leads to reduced memory access latency and increases performance by minimizing access to global GPU memory.

All these optimizations led to a significant improvement in code performance compared to the original code with a single thread and with the code that was parallelized with OpenMP. Detailed analysis of these results will be discussed in the results section.

D. Benefits of CUDA Optimizations

Optimizations performed with CUDA in high-performance computing applications, such as molecular dynamics simulations, offer several significant benefits. These benefits derive mainly from the architecture and capabilities of modern GPUs. Some of the key benefits of CUDA optimizations are:

- **Massive Parallelism:** The GPU can process thousands of threads simultaneously, which is ideal for systems with a large number of particles.

- **Reducing Data Transfer Bottlenecks:** By performing calculations directly on the GPU, the need to frequently transfer data between the CPU and GPU is minimized.

- **Efficient Use of GPU Memory:** Allocating and managing data directly in GPU memory improves performance by reducing memory access delays.

- **Improved Calculation Intensive Performance:** Operations such as force calculations and state updates, which are calculation intensive, are substantially accelerated.

II. Optimizations implemented using CUDA

As we noted in the previous section, several optimization techniques were implemented in the program in order to maximize parallelism and GPU efficiency, efficiently manage memory and data and guarantee computational efficiency, taking into account that we are working with computationally intensive applications. as is the case with molecular dynamics simulations.

To carry out a detailed analysis of the implemented optimizations, it is important to understand how each function

was adapted from the previous program model that had been parallelized with OpenMP, to the GPU-based model (CUDA) and what the benefits of these adaptations are.

1) **Initializing Velocities (initializeVelocitiesKernel)**

   - With CUDA, it employs **initializeVelocitiesKernel** with CURAND to generate random numbers directly on the GPU. This change eliminates the need to transfer random number data from the CPU to the GPU.

2) **Initializing Positions (initializePositionsKernel)**

   - This function replaces the nested loop in the initialize function of the OpenMP version of the program. Each CUDA thread initializes the position of a particle, which is significantly faster due to the massive parallelism of the GPU.

3) **Kinetic Energy Calculation (kineticEnergyKernel)**

   - Replaces the kinetic energy calculation performed by the *kinetic* function in OpenMP. The **kineticEnergyKernel** function calculates the kinetic energy of each particle in parallel, and the final sum can be performed using the *Thrust* library for efficiency.

4) **Potential Energy Calculation (pairwisePotentialKernel)**

   - Similar to calculating kinetic energy, the **pairwisePotentialKernel** function calculates the potential energy between pairs of particles in parallel. This is a huge improvement over the OpenMP model, especially for systems with many particles.

5) **Acceleration Calculation (computeAccelerationsKernel)**

   - This function calculates accelerations based on interaction forces. Each thread deals with one particle, drastically reducing the time needed to compute forces compared to CPU parallelization..

6) **Update Positions and Velocities (updatePositionsKernel and updateVelocitiesKernel)**

   - These kernels update the particle positions and velocities. Unlike the version with OpenMP, where these updates are parallelized on the CPU, the CUDA optimization perform these operations for all particles in parallel on the GPU.

All these techniques applied to the MD simulation program gave us significant improvements in efficiency and performance, specially for large-scale simulations. CUDA kernels were important for performing computations in parallel across a large number of threads. To calculate forces and state updates, complex functions used in previous work assignments were replaced by CUDA kernels, such as (computeAccelerationsKernel, updatePositionsKernel, updateVelocitiesKernel). Optimizations take advantage of the parallelism and processing power of modern GPUs. While OpenMP leverages multicore CPUs for parallelization, the CUDA version exploits the parallel processing capability of GPUs, ideal for intensive and repetitive calculations in molecular dynamics simulations.

III. Results

The CUDA implementation has demonstrated a remarkable performance enhancement compared to its predecessors. The previous versions of the program, a sequential optimized version and a parallelized version using OpenMP, executed in average 150 seconds and 90 seconds respectively. In contrast, the CUDA version showcased a significant reduction in execution time, completing the simulation in merely 2 seconds, this improvement by itself demonstrates the power of utilizing GPUs for heavily parallelizable tasks. As said, this drastic reduction in execution time by the CUDA version can be attributed to its highly parallelized nature leveraging the computational capabilities of the cluster's GPUs and promising scalability with the increase in the number of particles (N). This represents a significant advancement in computational efficiency. Its ability to execute complex calculations in a fraction of the time taken by CPU-based versions exemplifies the potential of GPU computing. In the following graph, we present the execution time for the CUDA version of our molecular dynamics simulation. This visualization offers a clear perspective on the performance efficiency achieved through parallel computing on GPU architecture.

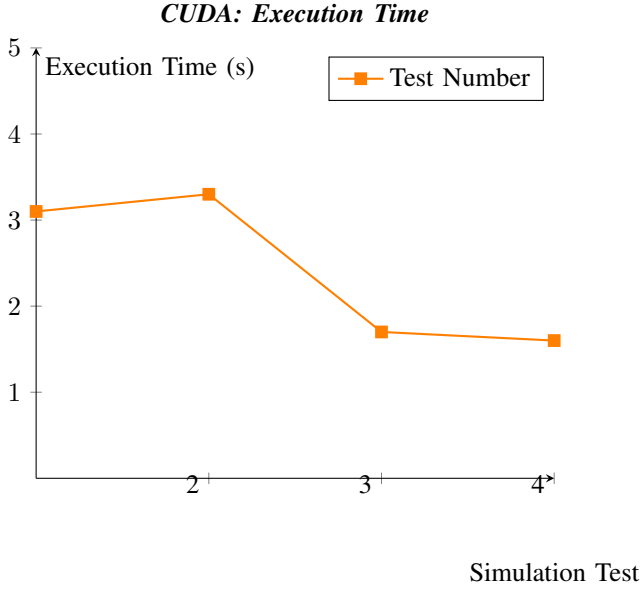


Fig. 1. Execution time of the CUDA program

As for the tests to the solution's performance the main focus is going to be the analysis of performance statistics since we mainly followed the Lab Practical Sessions learnings and applied these techniques in order to study the performance of

each version. So in our upcoming analysis, we will present the performance results of our molecular dynamics simulation through Table 1 and Table 2, to provide an understanding of the efficiency of each implemented version. Our results mainly show that the Sequential Optimized version, which was the first developed in the WA1, exhibited results in line with our initial expectations, demonstrating a consistent performance that lays the foundation for understanding the impact of further optimizations. As for the WA2 the OpenMP version, despite its parallel nature, displayed somewhat disappointing results, meaning we did not implement an optimal solution. This opens up an avenue for future investigation and potential optimization to enhance its performance. Contrasting this, the WA3 CUDA version, leveraging the power of GPU computing, showed remarkably good results. This highlights the significant impact of GPU acceleration in computational tasks, particularly in the context of heavily mathematical and complex simulations like ours. The juxtaposition of these results not only underlines the varying effectiveness of different parallel computing approaches, but also the real difficulty in applying these parallelization techniques.

The results presented in Table 1 and Table 2 are an agglomerate of many tests, performed in the cluster mostly through the use of tools such as **perf** , and the values written are the average values across all the documented tests.

TABLE I
SOLUTION TESTS FINDINGS

Simulation Version	**Cluster Testing Results**		
	#CC (millions)	***#I (millions)***	***Inst per Cycle***
Base	335 877	623 371	1.86
Optimized	35 433	52 255	1.47
OpenMP	95 163	78 288	0.82
CUDA	2 181	1 316	0.6

TABLE II
SOLUTION TESTS FINDINGS

Simulation Version	**Cluster Testing Results**		
	CPI	***L1_DMiss (millions)***	***Execution Time***
Base	0.8	2 579	133.1s
Optimized	0.8	670	17.2s
OpenMP	1.4	678	42.5s
CUDA	1.6	12	1.6s

Analyzing these tables we can conclude that the CUDA version is, in fact, much better optimized and scalable than the other versions. It shows huge improvements in execution times, a lower number of cache misses meaning it's utilizing the cache more effectively, and a lower number of instructions which can be an indicating factor of a reduction of redundant or unnecessary calculations that could have been minimized or eliminated.

Lastly in the Figure 2, we offer a comparative analysis of the execution times across the three distinct versions of our molecular dynamics simulation project, developed throughout the semester. Each version represents a milestone in our journey of progressive optimization and parallelization: the Sequential Optimized version from Work Assignment 1, the OpenMP version from Work Assignment 2, and the final CUDA version for the Work Assignment 3.

This figure provides a side-by-side comparison of how each approach fares in terms of computational efficiency. The graph aims to showcase the evolution from a CPU-centric sequential computation to a multi-threaded OpenMP implementation, and ultimately to the highly parallelized CUDA architecture. It also highlights the significant strides made in reducing execution times and improving performance with each successive version, offering a compelling narrative of the journey from sequential processing to the advanced realm of GPU computing.

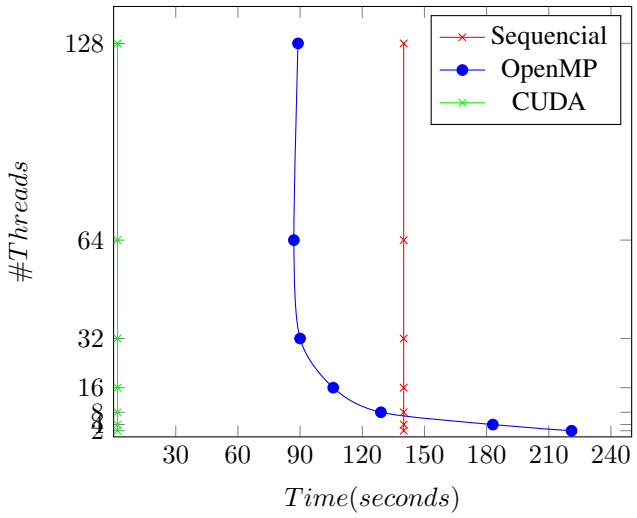


Fig. 2. Execution time comparison between versions: Sequential Optimized, OpenMP, CUDA

IV. CONCLUSION

We believe that this academic unit covers topics of greater complexity, which proved to be an added challenge throughout the semester. However, despite the difficulties we experienced in developing the versions and completing the assignments, we gained very important knowledge. We concluded the last practical work with the understanding that parallel computing and the parallelization techniques we learned have broadened our horizons. We recognize that these skills will always be integral to the development of efficient software. The truth is that through these three assignments, we have gained a clear understanding of the significant impact that efficient utilization of a machine's resources has on its performance.

ANNEXES

Work Assignment 1 - Optimization Techniques Applied To a Single Threaded Molecular Dynamics Simulation Code

Hugo Marques
Universidade do Minho
Braga, Portugal
pg47848@alunos.uminho.pt

Nuno Mata
Universidade do Minho
Braga, Portugal
pg44420@alunos.uminho.pt

***Abstract*—This paper investigates the application of diverse optimization techniques to a single-threaded Molecular Dynamics Simulation program. The computational demands of such simulations are often substantial, necessitating the need for improved algorithms and software implementations in order to manage hardware resources more efficiently. As such, the focus of this paper is to study the impact surrounding the integration of optimization techniques that result in improvements in computational efficiency and execution speed. The paper aims to highlight the critical role of the implementation of such optimization techniques in accelerating the performance of computationally demanding programs and to better leverage the capabilities of modern processor and memory architectures. The experimental results demonstrate significant performance gains enabling the Molecular Dynamics Simulation program to achieve much lower execution times while maintaining computational efficiency. These findings further emphasize the crucial role of tailored optimization strategies in accelerating the execution of computationally demanding programs, thereby facilitating their applicability in a wider range of computational resources.**

***Index Terms*—optimization, single-thread, computational efficiency, parallelism, memory hierarchy, data structures, vectorisation, loop unrolling**

I. Introduction

Molecular dynamics simulations programs are computational techniques used to study the behavior of atoms and molecules over time. These simulations have been conducted on computers rather than in real life due to several practical and technical reasons, such as: **Scale and Complexity**, since systems studied in molecular dynamics simulations often involve a large number of particles, and creating physical setups to mimic such complex systems is practically unfeasible. Computers also allow for **Precise and Controlled** manipulation of various parameters, including temperature, pressure, and external forces which enable researchers to explore a wide range of conditions. The **Cost and Time** of real-life experiments can be extremely expensive and time-consuming, especially when dealing with intricate molecular interactions. Alternatively, molecular dynamics simulations provide a cost-effective and time-efficient solution. Lastly, another big factor is **Observability** since computer simulations offer a unique advantage by providing insights into the atomic and molecular processes that are challenging to observe directly in real experiments.

In conclusion, the simulations aim to provide insights into the dynamics and properties of materials at the atomic and molecular levels, including their structural changes, thermodynamic properties, and behavior under different conditions. By simulating the trajectories of particles under the influence of inter-atomic or inter-molecular forces, it's possible to better understand concepts such as phase transitions, diffusion, chemical reactions, and complex molecular interactions.

II. Explaining The Molecular Dynamics Simulation Algorithm

The program provided with this assignment aims to simulate a molecular dynamics system for a noble gas (Helium, Neon, Argon, Krypton, or Xenon). It initializes a set of particles with specified positions and velocities, and then updates their positions and velocities over a series of time using the Velocity Verlet algorithm (the velocity Verlet algorithm provides both the atomic positions and velocities at the same instant of time). The program then calculates various thermodynamic properties referent to the simulation environment, such as temperature, pressure, kinetic energy, potential energy, mean squared velocity, gas constant, and compressibility (the measure of how much a given volume of matter decreases when placed under pressure). The program uses the Lennard-Jones potential (which is a simplified model that describes the essential features of interactions between simple atoms and molecules) to compute inter-molecular forces and accelerations of the particles. After executing the program, it generates two output files, one with all the calculated values for the physical properties surrounding the environment in the executed simulation and another with just the averages of all the previous values. One more thing to mention is that the simulation initially takes 3 user inputs, the noble gas to use in the simulation, the initial temperature in Kelvin and lastly the number density in $moles/m^3$. Finally, throughout the simulation, in the CLI (command line interface), there gets printed some information: Prompts asking for the initial parameters if they weren't provided via text file, the overall

simulation progress percentage towards the completion, and the results stored in the averages files mentioned previously.

III. Optimization techniques

In this section of the paper, the optimization techniques that were implemented in the optimized version of the code are demonstrated. In the optimized version, several optimization techniques were employed to improve the overall efficiency and execution time, such as:

- **Mathematical optimization** - The expression `L = cbrt(Vol)` was added using the `cbrt` (cube root) function from the `<cmath>` library, which is more efficient than calculating the cube root manually.
- **Calculation Optimization** - In the optimized version, calculations such as " double quot 2= quot * quot " and "$f = 24\times\left(\frac{2}{rSqd^8} - \frac{1}{rSqd^4}\right)$" have been introduced in order to reduce the number of repetitive multiplications.
- **Changes to the Loop Structure** - Some loops have been modified/restructured in order to try to improve the efficiency of the loops, to take advantage of data locality or to reduce the total number of iterations.
- **Memory Hierarchy** - No specific techniques (like blocking) are used to exploit cache memory. The arrays are accessed in a fairly straightforward manner. However, some local variables are used in loops, which can be beneficial for cache usage.
- **Data Structures Organization** - The original code already has two-dimensional arrays that are used to store information about the particles, such as position (r), velocity (v) and acceleration (a). This is an efficient organization, as it allows contiguous access in memory, which is beneficial for cache performance.
- **Efficient Arithmetic Operations** - In the `Potential` function, instead of using `pow` for computing power, as in the the original version, multiplication is used, which is computationally cheaper.
- **Use of Efficient Algorithms** - The Gaussian distribution number generator is used for initializing velocities.
- **Compiler Flags** - After experimenting with many compilation flags we achieved what performed best for our case which improved the performance of compiled code.

We ended up implementing some optimization techniques, but there is room for further improvement, especially in terms of exploiting modern hardware features such as vectorisation, parallelization and the implementation of parallelization libraries.

IV. Tests and Results

This section presents a study to the improvements achieved in the developed MD simulations optimised code. We will compare the tests results between the base code provided with the assignment *vs* our optimized version. As for the tests performed the main focus is going to be the analysis of performance statistics since we mainly followed the Lab Practical Sessions learnings and applied these techniques in order to study the performance of each version. These testing techniques are based on execution flags that enable the output of statistics when the program is executed. Lastly we will be able to understand the improvements achieved after implementing the techniques mentioned in the previous section.

Tables bellow introduce the results of many tests performed in the cluster and the values written are the average values across all the documented tests.

All the testing was done in the university's cluster, which has got a Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz CPU Model.

TABLE I
Solution Tests: Base *vs.* Optimised

Simulation Version	**Cluster Testing Results**		
	#CC	***#I***	***Inst per Cycle***
Base	335 877 million	623 371 million	1.86
Optimized	19 923 million	28 236 million	0.96

TABLE II
Solution Tests: Base *vs.* Optimised

Simulation Version	**Cluster Testing Results**		
	CPI	***LI_DMiss***	***Execution Time***
Base	0.8	2 579 million	133.1s
Optimized	1.2	349 million	6.01s

Looking at the tables, we can observe several significant improvements, including:

- 95% faster execution time;
- 86% lower number of cache misses meaning the optimised version is utilizing the cache more effectively;
- 95% lower number of instructions which can be an indicating factor of a reduction of redundant or unnecessary calculations that could have been minimized or eliminated.

V. Conclusion

In this study, we tackled the optimization of a molecular dynamics simulation implementation. Through the initial code analysis, we pinpointed several potential areas for enhancement and introduced specific optimizations mentioned in section II.

While the original version of the code was functional and provided accurate results, the optimizations introduced brought about some significant improvements in computational efficiency, as verified in the results presented previously. However, it was observed that additional optimization opportunities remain. Techniques such as the implementation of neighbor lists, enhanced vectorization and the use of dedicated parallel libraries such as openmp and cuda, could be further explored in future works.

In conclusion, with the comparison of the tests between the base version and the optimized version, we can observe significant improvements not only in terms of performance but also in the efficiency of the use of computational resources, and therefore, we are pleased with the result of our work.

REFERENCES

[1] Abraham MJ, Murtola T, Schulz R, Páll S, Smith JC, Hess B, and Lindahl E (2015). GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX 1, 19–25. [Google Scholar]

[2] K. Chen and K. Zeng, "Performance Optimization Model of Molecular Dynamics Simulation Based on Machine Learning and Data Mining Algorithm,". Department of Basic Science, Jiaozuo University, Jiaozuo 454000, Henan, China.

Work Assignment 2 - Exploring parallelism with OpenMP

Hugo Marques
Universidade do Minho
Braga, Portugal
pg47848@alunos.uminho.pt

Nuno Mata
Universidade do Minho
Braga, Portugal
pg44420@alunos.uminho.pt

***Abstract*—This paper extends the work of optimizing a single-threaded Molecular Dynamics Simulation program, previously enhanced through various optimization techniques. The primary objective of this phase is to identify and exploit parallelism in the program's most time-consuming sections, known as hot-spots. By strategically implementing OpenMP directives, we aimed to decompose computational tasks, parallelize independent tasks, and efficiently utilize multi-core processor capabilities. This approach diverges from the initial single-threaded optimization by leveraging the inherent parallel nature of molecular dynamics simulations. The implementation of OpenMP resulted in a notable reduction in execution times, highlighting the effectiveness of parallel computing in scenarios where computational tasks can be concurrently executed. The experimental results reveal a substantial improvement in performance, demonstrating the significant impact of parallel processing techniques in further accelerating a computationally intensive program. This study not only emphasizes the necessity of parallelism in high-performance computing but also showcases the transformative potential of OpenMP in enhancing the execution speed and efficiency of demanding computational tasks, thereby broadening the scope of their applicability across various multi-core platforms.**

***Index Terms*—Optimization, OpenMP, Parallel, hot-spots**

I. Introduction

Building upon our previous work in optimizing Molecular Dynamics (MD) Simulation programs, this phase of the project aims to improve computational efficiency through parallel processing. The first phase laid the groundwork by improving the algorithmic and software implementations of a single-threaded MD simulation, focusing on the optimization of a program that inherently involves complex and computationally demanding tasks. These simulations, crucial for understanding atomic and molecular behavior over time, have proven to be intensive in terms of computational resources.

The advancement in computational technology, especially the widespread availability of multi-core processors, presents an opportunity to further optimize MD simulations. In this phase, we focus on exploiting this technological evolution by introducing parallelism into the previously optimized code. The aim is to identify the most time-consuming parts of the program - the hot-spots - and apply parallel computing strategies, particularly using OpenMP directives, to these section, which can offer a straightforward approach to parallelize existing code with minimal intrusion and high adaptability.

The rationale behind this approach is to distribute the computational workload across multiple cores, thereby reducing overall execution time and enhancing the simulation's efficiency. This is particularly pertinent in MD simulations where calculations for inter-atomic or inter-molecular forces and particle trajectories can be performed in parallel. By transforming the simulation from a single-threaded to a multi-threaded paradigm, we aim to leverage the full potential of modern computing architectures.

In essence, this phase intends to bridge the gap between the computational demands of MD simulations and the capabilities of current processor technologies. By integrating parallel computing techniques, we aim to significantly expedite the execution of MD simulations while maintaining, or even improving, their accuracy and efficiency. This could notably broaden the scope of MD simulations, enabling more complex and detailed studies within feasible time-frames.

II. Research Phase

In order to optimize our previous application performance, a structured approach is essential. This process can be broken down into three key steps, which were well documented in the work assignment paper: **(1)** Identifying our application hot-spots, in other words, the first step of optimizing our previous implementation begins with profiling the application to pinpoint code blocks that consume a significant portion of computation time, otherwise known as hot-spots. With this information we can then focus on studying the blocks that are the most time-consuming, as optimizing these areas offers the greatest potential for enhancing overall performance, which is our end goal. **(2)** Analyzing and presenting alternatives for parallelism, this step consists in analyzing these mentioned sections/blocks for opportunities to further introduce parallelism. Alternatives for parallelism can include breaking down tasks into smaller, independent units that can be executed simultaneously, or identifying data-parallel operations that can be performed concurrently. Each alternative should be assessed for its feasibility and potential performance gains. **(3)** Selecting an approach and justifying with scalability analysis, which is the final step. Here, we select the most appropriate approach for introducing parallelism, based on a scalability analysis. This means that we will evaluate how the performance improvements scale with additional resources, such as more CPU

cores or threads. The chosen approach will address the current performance bottlenecks but also demonstrate scalability for future demands, ultimately we will decide for the best balance between implementation complexity and performance gain we can achieve.

In our quest to optimize the application's performance, we utilized a suite of profiling tools, namely *gprof, perf stat, and perf report*, which played a crucial role in identifying key performance issues. These tools allowed us to dissect the application's execution at a granular level, providing insights into function calls, execution times, and resource utilization. The conclusion this profiling process led to a significant revelation: the **computeAccelerations** function emerged as the most frequently called function in our program. This function, evidently critical to the application's core operations, was consuming a disproportionate amount of computational resources. Its identification as a primary hot-spot highlighted it as a prime candidate for optimization, particularly through parallelization techniques.

Concluding, the combination of gprof, perf stat, and perf report was very helpfull in revealing the **computeAccelerations** function as a major bottleneck in our application, this discovery helped guiding us to our subsequent optimization efforts. With this knowledge, we were now well-equipped to explore and implement effective parallelization strategies to enhance the application's efficiency and performance.

III. Parallelization Techniques

In this section, we will describe the implementation of parallelization techniques using OpenMP, a widely-used library for parallel programming in C++. This API allowed us to parallelize sections of code with minimal intrusion, using what is called OpenMP Directives. To better understand what the hot-spots in our code would be, we used the "gprof" tool, which provides information about the time spent on each function. We came to the conclusion that the computeAccelerations() function was the one that took the longest to execute. The others functions were parallelized too, as we will see below.

Working on the **computeAccelerations** function, identified as the primary bottleneck of our program, we experimented with various versions and different directives to enhance its performance. Our approach involved rigorously testing a range of modifications, each aimed at maximizing efficiency and reducing execution time. However, the journey was not without its challenges and our final result, despite the efforts and numerous iterations, yielded less performance gains than anticipated. During the testing phase on the cluster we encountered unforeseen complications: our modifications had inadvertently introduced data races into the code. Data races occur when multiple threads access shared data concurrently, and at least one thread modifies the data, leading to unpredictable behavior and potential errors. This discovery was a critical juncture in our project. Data races not only compromise the integrity and reliability of the application but can also lead to subtle and hard-to-detect bugs. Recognizing the severity of this issue, we had to make a pivotal decision.

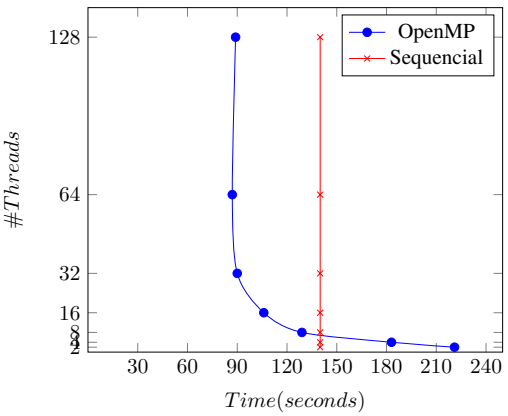
In the end, we opted for a more conservative approach. We reverted to a functional version of the **computeAccelerations** function that was free from data races. While this version was not as optimized as we had hoped, it offered a balance between improved performance and operational stability.

For this function, we employed a set of OpenMP directives, the specific directives used were: **(1) #pragma omp parallel for** which is used to parallelize loops by distributing the iterations across multiple threads. **(2) #pragma omp parallel for private** this directive is an extension of the previous one, with an added specification of private variables. The **private** clause ensures that each thread has its own copy of these variables, preventing interference and data inconsistency between threads. This is crucial in scenarios where variables are used within a loop and need to be initialized or modified independently in each iteration, maintaining the integrity of the function's computations across its parallel execution. Lastly **(3) #pragma omp atomic**, used to ensure that a specific memory operation is performed atomically, meaning that it is executed as an indivisible operation. This is critical in parallel computing to avoid data races and inconsistencies, especially when multiple threads are updating the same variable.

In more scenarios we used other OpenMP directives such as **#pragma omp parallel for reduction**, which is helpful when we need to accumulate a value across iterations in a parallel loop.

IV. Results

In this section, we will demonstrate the results of our implementation by comparing the sequential implementation from TP1 with the parallel implementation from TP2. The focus will be on how the performance varies when using different numbers of threads, specifically 2, 4, 8, 16, 32, 64 and 128 threads.



Based on the data gathered regarding execution times, we can draw two main conclusions: First, the paralyzed code using OpenMP demonstrates superior performance compared to the optimized sequential code from TP1.

Second, there is a non-linear relationship between the number of threads and the program's performance. An interesting observation is made at 32 threads, where performance begins to plateau, and further at 128 threads, where a decrease in performance is noted. This phenomenon is likely attributable to the overheads associated with managing a large number of threads. As the number of threads increases, the overhead costs of thread creation, management, and synchronization can outweigh the benefits of parallel execution. This is especially true when the number of threads exceeds the number of cores available on the processing unit, leading to excessive context switching and resource contention.

Therefore, while parallelization through OpenMP significantly boosts performance over the sequential approach, there is a threshold beyond which increasing the number of threads does not yield proportional performance gains and can even degrade performance.

V. Conclusion

In this work we evaluate the implementation of a parallel optimization to the a simple molecular dynamics program and a detailed analysis has been performed. We address the optimization of a set of critical functions in a molecular dynamics simulation, with a special focus on the implementation and effectiveness of parallelism using OpenMP. Through detailed analysis and careful modifications, we were able to significantly improve the performance of the initialize(), computeAccelerations(), Kinetic(), Potential(), and VelocityVerlet() functions.

The initialize function has been transformed from an iterative, sequential process to a parallel approach, leveraging the power of multiple cores. Using the #pragma omp parallel for collapse(3) directive, we significantly reduce the time required for this critical step. This optimization was particularly effective due to the conversion of the three loops into a single long loop, as explained in the code, allowing efficient distribution of iterations among the available threads.

For the computeAccelerations, Kinetic, Potential, and VelocityVerlet functions, we implemented parallelism with OpenMP to distribute the calculation of accelerations, kinetic energy, potential energy, and position and velocity updates, respectively. The key strategy in these optimizations was the use of the reduction clause to correctly and efficiently accumulate values between threads, avoiding common problems such as race conditions. The result was faster execution of these functions, crucial for simulations of large particle systems.

In summary, incorporating OpenMP into core simulation functions has proven to be a powerful strategy for optimization. The improvements resulted in a significant reduction in calculation time, an essential advantage for complex and computationally intensive simulations. This study reinforces the importance of parallelism and code optimization in scientific computing applications.