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.

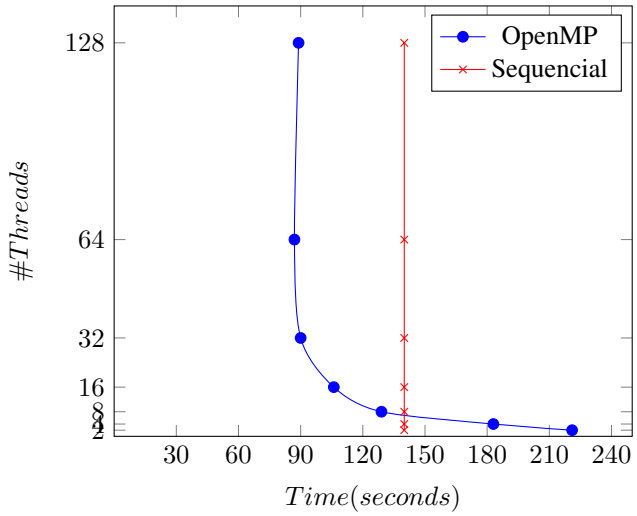
In the end, we opted for a more conservative approach. We reverted to a functional version of the **computeAccelerations** function that was free from data races. While this version was not as optimized as we had hoped, it offered a balance between improved performance and operational stability.

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.