Parallel Computing
Simple Molecular Dynamics

Gustavo Pereira
Informatics Department
University of Minho
Braga, Portugal
pg53723@uminho.pt

Nelson Almeida
Informatics Department
University of Minho
Braga, Portugal
pg52697@uminho.pt

***Abstract*—This paper presents an analysis and optimization program aimed at enhancing the performance of a simple molecular dynamics simulation code designed for the study of Argon gas atoms. The original code, FoleyLab/MolecularDynamics: Simple Molecular Dynamics, employs a generic approach to simulate the dynamic behavior of particles over discrete time steps, adhering to Newton's laws of motion. The given code lacks efficiency due to its exhaustive loops that result in a heavy computational load leading to substantial execution times. In this paper we will analyse the code and apply various optimization techniques, taking advantage of ILP (Instruction Level Parallelism) and Memory Hierarchy, in order to reduce execution time.**

I. Introduction

This project aimed to explore optimization techniques applied to a single-threaded program (a simple Molecular Dynamics simulation code applied to atoms of Argon gas) utilizing tools for code analysis and profiling to minimize the execution time. This optimization will be based on concepts such as ILP (Instruction Level Parallelism) and Memory Hierarchy. Our initial approach involved the implementation of fundamental high-level optimizations, being the groundwork for subsequent enhancements through techniques such as Loop Unrolling and Vectorization. Since the input value was constant, our performance assessments were centered around key metrics, including execution time, cycle count, and the CPI (Cycles per Instruction) metric. Ultimately, the acquired results were subjected to a comparative evaluation against the anticipated theoretical outcomes.

II. Code Analysis

A. Profiling

Using *gprof2dot* we took a snapshot of the original call graph (Figure 1) where we found out that the most called functions were Potential, called 201 times occupying 64.29% of the execution time, and computeAccelerations, called 202 times spanning over 35.63% of the execution time. These two together use 99.92% of the execution time, making it obvious that these were the hot spots that should be optimized.

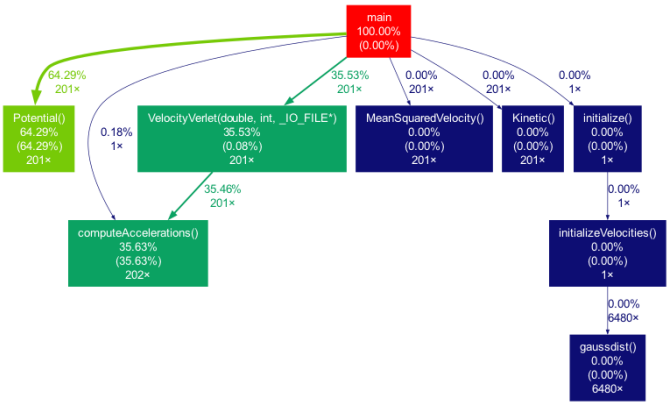

Fig. 1. Call graph for the source code

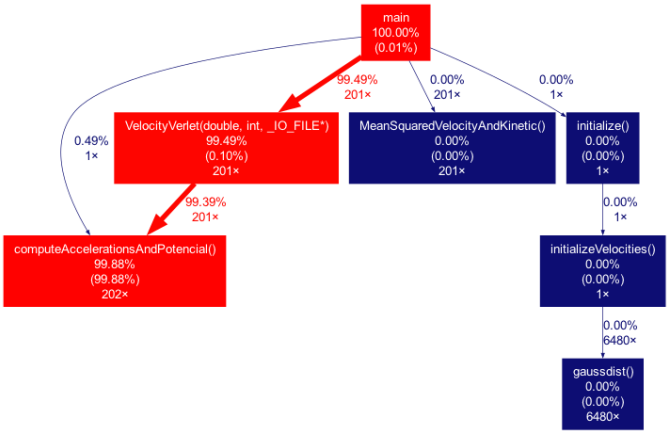

Fig. 2. Call graph for the optimized code

B. Complexity Analysis

The reason behind these large chunks of time can be found with a glance at the source code. These functions consist of nested for loops with fixed $N * N$ iterations that, given our fixed N value of 2,160, result in 4,320 iterations per call, giving us a total of 1,740,960 iterations in total. In the case of *computeAccelerations()*, we have an extra loop to set the accelerations matrix to zero which gives us plus N*3 iterations, and inside the N*N loop, we have two other for loops that run 3 times each.

C. Mathematical Operations

Inside the main loops, one can find mathematical operations for the formulas that mimic the properties of our Molecular Dynamics simulation. Translating those formulas into code leads to the use of operations such as multiplication, division, exponentiation, etc. Depending on the operations used, these calculations can lead to a greater number of cycles, since a division takes 36 more cycles than a simple addition operation instruction which takes up to 3 cycles. With that in mind, a possible optimization could be simplifying the mathematical formulas to reduce the number of division or exponentiation operations.

III. Optimization Techniques

Given the intensive use of costly mathematical operations and nested for loops, as seen previously, our first optimization was reducing the number of heavy mathematical operators such as the division and exponentiation, by reformulating mathematical equations, adopting new strategies to reduce the number of repetitive loop iterations and merging functions that use the same type of loops, which was the case for *Potencial()* and *ComputeAccelerations()* functions. These optimizations resulted in a large decrease in the total number of cycles (since the cycle-consuming operations were reduced) and a reduction in the total number of instructions thanks to our merged functions.

Total	Before	After
Instructions	1,236,991,944,534	260,326,290,217
Cycles	766,640,483,170	163,770,254,165

Besides these, other optimization techniques can be applied to decrease execution time, such as Loop Unrolling and Vectorization. These take advantage of concepts such as ILP (Instruction Level Parallelism) and Memory Hierarchy and can be employed by the gcc compiler when specific flags are used. For these to work we had to facilitate our code, which meant turning our matrices into vectors and converting our nested loops into N*3 loops.

In this version, we analyzed to assess the impact of various optimization techniques, specifically focusing on the application of gcc compiler flags such as -O2, -O3. When combined with the "-funroll-loops" and "-ffast-math" flags, these optimizations allow for the utilization of loop unrolling methods within the code and a compiler-driven math optimization. The table below presents the execution times associated with each optimization. These tests were conducted using the source code on the cluster. As expected, the execution time exhibits a decrease with the escalation of optimization levels, with the best performance achieved using the -O3 -funroll-loops -ffast-math flags.

IV. Results

TABLE I
Performance Counter Comparison

Metric	Source	Optimized
Cache misses	28473957	11795
Instructions	46255998416	32537845062
Cycles	782545448209	19379819430
Time Elapsed	269.721383703 seconds	6.015 seconds

V. Conclusion

In this study, we embarked on a journey to enhance the performance of a straightforward molecular dynamics simulation code designed for modeling the interactions of Argon gas atoms. The initial code, known as FoleyLab/MolecularDynamics: Simple Molecular Dynamics, grappled with inefficiency due to computationally intensive loops, resulting in protracted execution times. To address these challenges, we employed a range of optimization techniques, focusing on harnessing Instruction Level Parallelism (ILP) and Memory Hierarchy.

Our journey commenced with a profound code analysis and profiling, providing us with invaluable insights into the performance bottlenecks. Notably, the functions "Potential" and "computeAccelerations" emerged as the culprits, characterized by nested loops with an extensive number of iterations. Our first optimization strategy revolved around the simplification of mathematical operations, thereby mitigating resource-intensive computations such as division and exponentiation. Additionally, we strategically consolidated similar functions like "Potential" and "computeAccelerations" to minimize the repetitive iterations, substantially diminishing the total number of instructions and cycles.

Going a step further, we explored advanced optimization techniques, including Loop Unrolling and Vectorization, which harnessed the power of ILP and Memory Hierarchy. The culmination of our efforts led to a significant reduction in execution time and further enhanced the code's overall performance.

It's worth noting that our study delved into the impact of specific compiler flags, including -O3, -ffast-math, -ftree-vectorize, -mavx, -mfpmath=sse, and -march=x86-64. The judicious application of these flags was pivotal in achieving optimal performance. Among these, the -O3 flag, when complemented by the -ffast-math and -ftree-vectorize flags, demonstrated exceptional results.

In conclusion, our expedition underscores the transformative potential of optimization techniques in elevating the efficiency of a basic molecular dynamics simulation code. These optimizations not only curtailed execution time but also ushered in a new era of improved performance. Our findings contribute to the realm of parallel computing and optimization, offering valuable insights into the enhancement of scientific simulations and akin applications.