Work Assignment - Phase 1

Afonso Ferreira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg52669@alunos.uminho.pt

Catarina Costa
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg52676@alunos.uminho.pt

I. Introduction

In our work assignment for the Parallel Computing course, we focused on optimizing a single-threaded program. We employed code analysis tools like *gprof* to pinpoint areas for improvement and evaluate the execution time of the program. Furthermore, we leveraged parallelism techniques to enhance performance.

The assigned program is a molecular dynamics simulation code that models the behavior of argon gas atoms. It employs Newton's law to simulate particle movements over time steps, and our goal was to boost program execution speed without compromising simulation accuracy.

For code execution and analysis, we made use of tools such as *perf*, which provided valuable insights into the program's performance characteristics. These tools enabled us to measure and analyze various performance metrics, aiding in our optimization decisions.

II. Code Analysis

The group began the work assignment by following a systematic approach, starting with a thorough analysis of the code. We recognized the importance of understanding the simulation and its underlying mechanisms before diving into the optimization process. This initial analysis allowed us to gain insights into the code structure and identify potential areas for optimization right from the start.

To further aid our optimization efforts, we utilized the code profiler *gprof*, which was suggested as a valuable tool. By running the code through *gprof*, we were able to generate a call graph that provided a visual representation of the code's execution flow. This call graph proved to be instrumental in identifying the specific blocks of code that required our immediate attention.

Figure 1 showcases the call graph obtained from the *gprof* analysis. This visual representation made us recognize that the two most important functions to optimize were first the *Potential()* and secondly, the *computeAccelarations()*.

By combining our initial analysis with the insights gained from the *gprof* analysis, we established a solid foundation for our optimization efforts. This comprehensive approach allowed us to make informed decisions and prioritize our optimizations effectively, ultimately leading to significant improvements in the code's performance and efficiency.

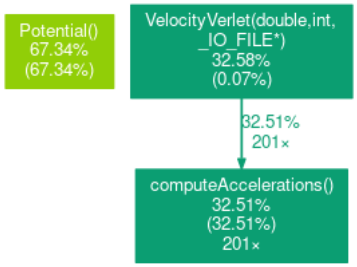


Fig. 1: Before Optimization

III. Optimizations

In our pursuit of optimization, we first focused on enhancing the performance of the crucial function, *Potential()*. This decision was influenced by its prominent presence in the *gprof* call graph, indicating its considerable impact on the overall execution time. Figure 2 (a and b) visually presents the optimization of the function *Potential()* through a side-by-side comparison of the before and after states.

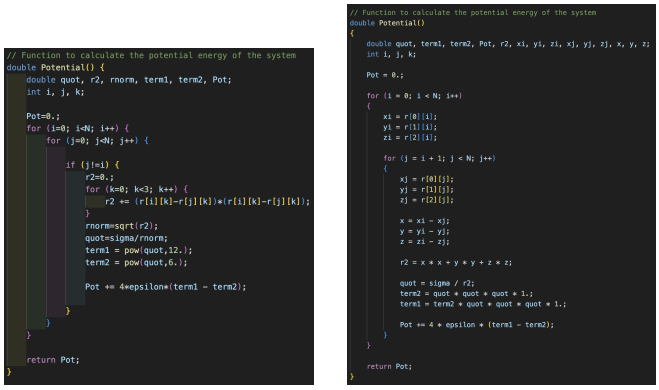
Subsequently, the function *computeAccelerations()* was identified as the next candidate for optimization.

To optimize these functions, we decided to:

- **Optimizing *pow()* and *sqrt()* functions** - We identified the usage of *pow()* and *sqrt()* functions in the *Potential()* function as possible bottlenecks. *pow()* can be computationally expensive, so we replaced it with more efficient manual calculations. We also were able to completely remove the *sqrt()* function for optimal performance. Here's an example that demonstrates how we eliminated the square root calculation while preserving mathematical integrity.

$$(\frac{\sigma}{\sqrt{r2}})^{12} = \frac{\sigma^{12}}{r2^6} = \frac{1}{r2^6}$$

- **Loop unroll** - The team explored the technique of loop unrolling. By expanding the loop body multiple times, we aimed to reduce the overhead of loop control and improve performance. We specifically targeted loops that were small enough (k = 3) to maintain code readability while still reaping the benefits of this optimization strategy.
- **Removal of the if condition** - The removal of an *if condition* within the *for loop* of the *Potential()* function.



(a) Before (b) After

Fig. 2: Potential() before and after optimizations

By carefully analyzing the code, we determined that this condition was unnecessary and removing it resulted in improvement of efficiency.

- **Avoid code redundancy** - Our analysis revealed the presence of repeated parcels throughout the code. To eliminate this redundancy, we isolated them in a variable, significantly improving the efficiency of the main functions.
- **Loop Optimization** - Minimizing operations, reducing memory access, and improving cache utilization in loops improves execution times and resource efficiency. We strategically declared and assigned variables before specific inner loops, reducing assignments for improved performance. The group enhanced efficiency and performance of *Potential()*, *computeAccelerations()*, and *VelocityVerlet()*. These optimizations improved execution speed and resource utilization, resulting in more efficient code, while maintaining legibility.
- **Spatial Locality** - Optimizing for spatial locality is crucial as it can greatly impact the cache performance of the program. To improve spatial locality, we transposed the four main 2D arrays (*position*, *acceleration*, *force*, and *velocity*) and made corresponding code adaptations. This resulted in a notable reduction in cache-references and cache-misses. Additionally, the group incorporated compiler flags to facilitate automatic vectorization, further enhancing performance in this context.
- **Compiler Flags** - These flags are settings and directives used to control the compilation process in C or C++ compilers. They influence code optimization, debugging information, warning levels, and more. In our Makefile, we used the following flags:

  -pg: Enables profiling support to collect information on function calls and execution time, aiding performance analysis.

  -ftree-vectorize: Enables automatic vectorization of loops using SIMD instructions, improving performance for certain computations.

  -msse4 and **-mavx**: Enable SSE4 and AVX instructions, respectively, providing additional SIMD instructions for improved performance.

  -march=native: Generates code optimized for the machine's native architecture, utilizing specific features and capabilities of the processor.

  -mtune=native: Optimizes code for the machine's native architecture, focusing on performance on the target machine.

  -O3: Enables aggressive optimization levels, performing various optimizations to improve code performance.

IV. Results

After all the optimizations, we got the following results:

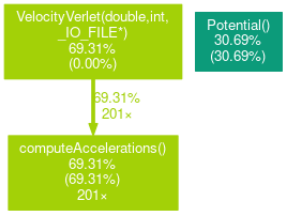


Fig. 3: After Optimization

```
Performance counter stats for './MD.exe':

    29,192,412,794      inst_retired.any:u
    24,512,859,574      cycles:u
    29,192,420,737      instructions:u            #    1.19  insn per cycle
    24,512,798,558      cycles:u
           340          cache-misses:u            #    0.148 % of all cache refs
       230,383          cache-references:u

      7.656753596 seconds time elapsed

      7.646867000 seconds user
      0.000000000 seconds sys
```

Fig. 4: Best results obtained in the SeARCH cluster

Conclusion

Overall, this work assignment provided us with a valuable opportunity to apply optimization techniques to a real-world simulation code, significantly enhancing our understanding of parallel computing and performance optimization.

While the desired instruction count was not achieved, we successfully minimized cache misses and cache references, demonstrating a significant improvement in this aspect.

Moving forward, if we were to continue this project, we would leverage OpenMP directives and explore memory hierarchy optimization. OpenMP, a programming model for shared memory systems, would enable us to further parallelize the code and exploit the memory hierarchy for enhanced performance.

Additionally, manual vectorization could be considered as a potential avenue for optimization, although it may require substantial involvement to be deemed viable.

In conclusion, this assignment has not only sharpened our practical skills in optimization but also ignited our curiosity to explore further avenues for performance improvement in parallel computing.