Parallel Computing
Practical Assignment 1

António Silva
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57867

Duarte Leitão
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57872

Vasco Faria
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57905

***Abstract*—This project simulates fluid dynamics using Jos Stam's stable fluid solver in 3D. The simulation incorporates dynamic events such as adding density sources and applying forces at specified timesteps. The main goal in this phase is for us to analyse and optimise the program in order to reduce it's execution time, by mainly using optimisation techniques that impact the program's performance and also techniques that make the code more legible to prepare it for upcoming phases in this assignment.**

***Index Terms*—performance, optimisations, execution time, ILP, memory hierarchy, data structures, vectorisation, legibility**

I. Introduction

The primary objective of this initial phase was to enhance the performance of a single-threaded program by applying optimisation techniques. To achieve this, we utilized code analysis and profiling tools to identify performance bottlenecks and improve the overall efficiency of the program.

Our process began with a preliminary assessment of the program's execution performance, followed by a more precise analysis using tools such as `gprof` and `perf`. This allowed us to pinpoint the specific functions and code segments that contributed significantly to execution time.

Based on these insights, we implemented a series of targeted optimisations aimed at reducing the program's runtime and improving its performance.

II. Code Analysis

A. Maintaining the Integrity of the Specifications

We conducted a thorough analysis of the code to ensure adherence to the defined specifications. To evaluate the performance of the code, we employed the `perf` tool, which enabled us to monitor execution time.

In addition to performance monitoring, we utilized profiling tools such as `gprof` to visualize the execution flow and pinpoint areas where optimisation was necessary. This graphical representation helped us understand the relationship between different functions and their contributions to the program's execution time.

By systematically analyzing the results, we were able to make informed decisions regarding potential optimisations while ensuring that the fundamental functionality and accuracy of the fluid simulation were preserved.

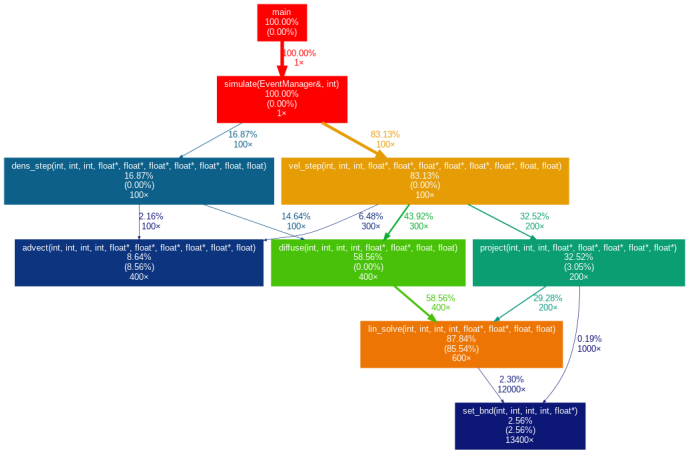

Fig. 1. Execution flow visualization from gprof.

One critical discovery during our performance analysis was that the `lin_solve` function represented the most significant area for optimisation. Profiling results revealed that this function was responsible for a substantial portion of the total execution time. To address this, our initial focus was on optimizing data access patterns, aiming to reduce cache misses and improve overall computational efficiency, which would result in faster execution without compromising the accuracy of the calculations.

To establish a baseline for our performance analysis, we measured the execution time of the default code provided as part of the project. The default implementation of the fluid simulation completed its execution in approximately 15 seconds. This baseline execution time serves as a reference point for evaluating the impact of our optimisations.

III. Optimisations

Once we identified our optimisation targets, we started focusing on applying optimisations:

A. lin_solve

In the `lin_solve` function, several optimisations were implemented to enhance performance and efficiency. Initially, the indices for neighboring grid points were computed repeatedly within the nested loops, leading to unnecessary

redundancy. By computing these indices (leftIndex, rightIndex, belowIndex, aboveIndex, backIndex, frontIndex) at the start of each iteration and storing them in variables, we significantly reduced redundant calculations. This change not only minimized the number of arithmetic operations but also improved cache efficiency by reducing memory access latency. Additionally, the neighboring values were accessed multiple times directly from the array, which was inefficient. By storing these values in temporary variables (left, right, below, above, back, front), we minimized memory accesses, leading to a noticeable improvement in execution speed due to fewer memory read operations.

B. advect

The advect function also underwent several optimisations to enhance its performance. Previously, indices and values for the velocity components (u, v, w) and density (d, d0) were computed and accessed multiple times within the loop. By precomputing these indices and storing the velocity components in variables (u_val, v_val, w_val), we reduced the number of times the program needed to access and compute the same values. This change saved computational resources and sped up execution. The clamping of coordinates, which was originally done using multiple conditional checks, was simplified using ternary operators. This not only streamlined the clamping logic but also reduced the number of conditional checks, improving execution speed. Furthermore, the bilinear interpolation calculation was optimized by using precomputed indices (i0, i1, j0, j1, k0, k1) and temporarily storing the density values, which reduced the number of memory accesses required.

C. project

Changing the loop order in the project function enhances performance by improving cache locality and reducing L1 cache misses. By iterating over the depth dimension (k) first, followed by the height (j), and then the width (i), the modified code accesses all relevant data within the same k slice before proceeding to the next. This sequential access pattern allows the CPU to leverage cache line fetching effectively, ensuring that once a cache line containing related data is loaded, subsequent accesses to that data are much faster. Consequently, this optimisation leads to fewer cache misses, as the inner loops access data that is already in the cache, resulting in shorter execution times and improved overall efficiency in the fluid simulation process.

D. Makefile

To further optimise our program we also used the following compiler flags in our Makefile:

- **-lm**: linking the math library with the -lm flag provides access to a robust set of mathematical functions and ensures optimized performance.
- **-O2**: improves performance by enabling a range of optimisations that enhance execution speed and efficiency, without significantly increasing compilation time or altering code structure.
- **-ffast-math**: enhances performance by allowing the compiler to make aggressive optimisations for floating-point arithmetic, potentially sacrificing some precision and adherence to standards for faster computation.

Other flag combinations were evaluated, however, they delivered suboptimal execution times compared to the selected optimisations.

IV. Results comparison

With the implemented optimisations, we achieved gains in several metrics. The execution time came down to approximately 8,7 seconds (+/- 0.0598 seconds). The L1-dcache hits percentage also saw a huge drop to 2,20%.

V. Conclusion

In this first assignment we were able to apply various compiler and code optimisations and analyse their impact into the final performance of our program, specially in terms of execution time. This initial phase of our project has allowed us to delve deeply into the optimisation of a fluid dynamics simulation program. By leveraging a combination of profiling tools and targeted optimisation strategies, we were able to reduce the execution time significantly, achieving a performance improvement from approximately 15 seconds to about 8.7 seconds.

Moving forward, we recognize the importance of continued performance monitoring and iterative optimisation. Ultimately, this project has provided valuable insights into the intricacies of performance optimisation in computational simulations, laying a solid foundation for future developments.