Computação Paralela

Work Assignment - Phase 1

Miguel Ângelo Martins Guimarães
PG55986
Universidade do Minho

Miguel Jacinto Dias Carvalho
PG57590
Universidade do Minho

Pedro Andrade Carneiro
PG55993
Universidade do Minho

I. Introduction

The primary goal of this work assignment is to improve the performance of a single-threaded program that simulates fluid dynamics.

II. First Approach (**ATTACHMENT 1, 2 AND 3**)

Note: All code executions presented in this document were performed using the following command:

```
srun --partition=cpar --exclusive perf
   stat -r 3 -M cpi,instructions
-e branch-misses,L1-dcache-loads,L1-
   dcache-load-misses,cycles,
   duration_time,
mem-loads,mem-stores ./fluid_sim
```

After running the provided code, we measured an initial execution time of 11 seconds. Profiling revealed that about 80% of the time was spent in the `lin_solve` function, so our optimization efforts focused primarily on this function.

During the analysis of the `lin_solve` function, several issues were identified that contributed to its high execution time. First, there were unnecessary mathematical recalculations, where identical expressions were repeatedly computed inside loops instead of being precomputed outside of them. This increased the total number of operations performed by the CPU.

Additionally, the function exhibited a high data dependency between iterations. These dependencies limited the potential for parallelism and prevented the use of more advanced optimization techniques, such as vectorization, since many operations could not be executed independently.

Another issue was poor spatial locality in memory accesses, resulting in a high number of L1 cache misses. This behavior negatively impacted the function's efficiency and the overall performance of the program.

Finally, the inability to vectorize the function was a significant limiting factor. Due to the data dependencies, it was not possible to take advantage of vector processing in many parts of the function, reducing the potential to execute multiple operations in parallel.

III. First Improvement: Application of `-Ofast` and Arithmetic Simplifications (**ATTACHMENT 4, 5 AND 6**)

The first improvement was achieved by applying the `-Ofast` compiler optimization flag and simplifying arithmetic operations in the `lin_solve` function.

The original version of this function involved frequent access to 3D grid indices and division operations, both of which can be computationally expensive, particularly when performed repeatedly inside nested loops.

A. Impact on Performance Metrics

The new profiling data shows an improvement in performance metrics:

Execution Time: The execution time dropped from **10.794** seconds to **7.6324** seconds, representing a **29.3%** reduction in the overall execution time.

Cycles and CPI: The number of cycles dropped from **34,771,627,739** to **24,741,049,732**, and CPI decreased from **1.8** to **1.3**. This reduction in cycles indicates that the CPU now completes instructions more efficiently, primarily due to aggressive optimizations in arithmetic operations and the elimination of complex divisions due to the `-Ofast` flag.

B. First Improvement Profiling

The profiling after optimization indicates a significant reduction in the overall time spent in `lin_solve`. While `lin_solve` was optimized, other parts of the program, such as `project` (31.89%) and `vel_step` (66.87%), begin to dominate a large portion of the execution time. Since `lin_solve` is responsible for the majority of the time spent in `vel_step`, further optimizations will be focused on it

IV. Second Improvement: Spacial/Temporal Locality (**ATTACHMENT 7, 8 AND 9**)

In the second round of optimizations, we modified the loop ordering in the `lin_solve` function to enhance spatial and temporal locality, which directly impacts the performance of the program by improving cache utilization.

A. Impact on Performance Metrics

After this second set of optimizations, the execution time was reduced from **7.6324** seconds to **5.9122** seconds. The significant reduction in L1 cache misses, dropping from **32.33%**

to **7.38%**, clearly indicates that the change in loop order has substantially improved memory efficiency.

B. *Second Improvement Profiling*

The second profiling shows a significant improvement in execution time and cache efficiency, primarily due to the loop reordering, which optimized the memory access pattern and greatly reduced cache misses. As in the previous optimization, `lin_solve` remains the main bottleneck in the speed of `vel_step`.

V. Third Improvement: Breaking Data Dependencies (**ATTACHMENT 10, 11 AND 12**)

The third round of optimizations was focused on addressing data dependencies that were limiting performance. By separating dependent operations into two distinct cycles and breaking the computational dependencies within the loops, we were able to further reduce the execution time and the number of CPU cycles. The part with data dependencies was separated from the part without data dependencies, allowing the part without dependencies to be vectorized, making it so that in the calculations with dependencies, it was only necessary to add this value, further reducing the execution time.

A. *Impact on Performance Metrics*

After this second set of optimizations, the execution time was reduced from **5.9122** seconds to **4.5954** seconds.

The L1 cache miss rate increased slightly to **11.33%**, but the overall number of L1 cache loads decreased. This indicates that the new memory access patterns are more efficient despite a small increase in cache misses.

The reduced execution time shows that separating the calculation of dependent variables from the independent ones was essential. The significant drop in CPU cycles further confirms that the function is now making better use of the available hardware resources, allowing the CPU to process more data in parallel.

B. *Third Improvement Profiling*

After applying this optimization, further improvements to `lin_solve` became limited. The time spent in `lin_solve` dropped from 41.87% to 34.06%. The relative time spent in `vel_step` increased to 81.16%. As `lin_solve` became increasingly optimized, the `project` and `advect` functions, which also make up `vel_step`, started to account for a larger computational weight. These functions were targeted to further reduce the overall execution time of the simulation.

VI. Forth Improvement: `project` Function (**ATTACHMENT 13, 14 AND 15**)

All the optimizations applied to the `lin_solve` function were similarly implemented for the `project` function.

We precomputed constants to avoid redundant calculations. We optimized index calculations and also improved memory access patterns, making them more sequential, reducing cache misses. This reduction in memory accesses was achieved by swapping the loop order to improve data locality.

A. *Impact on Performance Metrics*

The optimizations led to the total execution time dropping from **4.5954** seconds to **4.1883** seconds. The most significant impact came from the reduction in instructions executed, which dropped from **1,747,564,519** to **1,196,760,6979**. This decrease in instructions was the main reason of the improved execution time, as redundant computations were eliminated. While other metrics like cycles and cache misses also improved, the reduced instruction count had the biggest influence on overall performance.

B. *Forth Improvement Profiling*

There was a considerable decrease in the time spent on the `project` function. Consequently, `vel_step` also reduced its execution time but still consumes the majority of the time. Overall, the optimizations resulted in a more balanced execution, with better use of cycles and cache, contributing to the reduction in total execution time.

VII. Fifth Improvement: `advect` Function (**ATTACHMENT 16, 17 AND 18**)

A. *New Version*

Similarly to the changes made in `project`, in `advect`, unnecessary recalculations were removed, the loop order was swapped for better memory locality, and constants were precomputed.

B. *Impact on Performance Metrics*

After optimizing the `advect` function, the program saw significant performance improvements:

- **Execution time** dropped by **8.51%**, from **4.1883** seconds to **3.7726** seconds.
- **Cycles** decreased by **9.85%**, from **13.46** billion to **12.13** billion, indicating faster processing.
- **L1 cache load misses** reduced from 11.14% to 8.64%, improving cache efficiency.
- Although **branch misses** increased slightly, the overall gains in speed and efficiency outweigh this.

The optimization successfully reduced CPU stalls, leading to more efficient execution.

VIII. Flags and Conclusions

In terms of flags, `-O3` was used to achieve maximum performance at the code level, applying advanced optimizations such as loop unrolling and automatic vectorization. The `-Ofast` flag was also used, as previously explained, along with the `-pg` flag to obtain profiling via gprof.

In conclusion, was achieved a significant improvement in execution time as desired. These improvements were identified and developed through an iterative method (profiling, evaluating, optimization), always keeping the formula

$$T_{\text{exec}} = \#I \times (CPI_{\text{mem}} + CPI_{\text{cpu}}) \times f$$

in mind to determine which parts of the code required optimization.