Parallel Computing (Phase 1)

Lucas Oliveira
PG57886

João Silva
PG55618

Rafael Gomes
PG56000

***Abstract*—The program that was provided to us on this phase, it's a single-threaded program of non-interactive version of the fluid dynamics simulation code, a 3D version of the Jos Stam's stable fluid solver. The goal of this phase is for us to analyse and optimise the program to improve the execution time, by using optimisation techniques that would impact the program's performance and also techniques that would make the code more legible.**

***Index terms*—Execution Time, Vectorisation, Performance, Optimisations, Legibility**

I. Introduction

The main objective of this pratical work was to optimize a single-threaded program by employing various **optimization techniques** and **utilizing tools** for code analysis and performance evaluation to improve the program's overall efficiency.

Our first step was to analyze the provided code. Then we began by executing the program to measure its execution time. Following this, we use the tool ***gprof*** to identify the functions that contributed significantly to the program's execution time.

II. Code Analysis

A. Estimated Time Execution

Initially, we conducted a thorough analysis of the original code to gain a deeper understanding of the existing dependencies within the data structures. During this analysis, we identified several potential optimization areas, in particularly excessive iterations through vectors caused by repeated calls to certain data-independent functions, each with different vector arguments.

Next, we executed the program in its original state to gather information on its execution time, utilizing the *perf* tool to measure the duration of the code execution.

B. Code Profiling

To find out exactly which functions weighed the most in the execution, we used the *gprof* tool, and we realized by looking at the image in Fig.1 that the functions were costing the most were the *linear_solve*, as we can see, the function spends 85.33% of the time executing the function body of itself. So we focused a lot of our work optimizing the performance of this function.

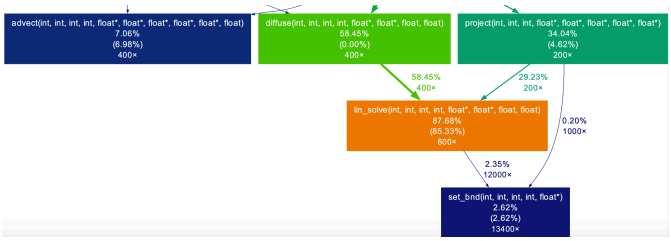


Fig. 1. Results from ***gprof2dot*** of the functions performance for the **non-optimised** code

III. Optimization

A. New Functions

It was noted that at a certain point in the code, the function *diffuse* is called consecutively 3 times, each time with a different pair of vectors calling the function *lin_solve* with the respective vector pair. We can see that each vector pair doesn't interfere with the other, meaning there is no data dependency, so we decided to compress the various calls of these functions into a minimum. This way we expect a decrease in the number of instructions, hoping for a better runtime. In order to do this we decided to create some new functions.

diffuse_3: Since the *lin_solve* function is called by the *diffuse* function, to reduce the number of times the first one is called, we need to start by the *diffuse* function. We implemented the new function *diffuse_3* wich receives not a pair of vectors, but the pair of vectors and will invoke the next new function

lin_solve_3: This new function has the same 4 cycles that *lin_solve* has, but, instead of receiving just a pair of vectors, also receives 3 pair of vectors from the *diffuse_3* function. This way, we can just iterate through the 4 loops just once, instead of 3 times, and for each iteration, we perform and operation on each pair of vectors.

Lastly we also implemented the *add_source_3* and *advect_3* function with both receive 3 pair of vectors instead of one. While *add_source* and *advect* are not very costly, we can very simply reduce their consecutive calls into just one, avoiding repetitive loops. These new functions execute the operations in *add_source* and *advect*, respectively, but in a single loop, execute the operations for each vector pair on each iteration.

B. Traversing the Vectors

We noticed that the way we are iterating through the vectors is not the most correct one. Let's say that in a given iteration we are looking at the element in the position *(i,j,k)*, in the way the *"for cycles"* are nested in the original code, in the next iteration we will be looking at the element in the position *(i,j,k+1)*, and if we look at the definition of the defined macro *"IX"*, this 2 positions have a distance of 44^2 from each other, meaning that between where they are stored in the memory, exist other 44^2 elements of the vector. This leads to a big number of cache misses, because when getting the value of the current element we are iterating through, it most likely won't be in the cache, because it wasn't loaded to the cache in the block fetched by the last iteration. We also note that given 2 elements at positions *(i,j,k)* and *(i,j+1,k)*, there are 44 elements between then in the memory, so we can also expect a big number of cache misses if we change the *"for cycles"* so that *j* is being iterated in the innermost cycle. So, we conclude that the best way rearrange the *"for cycles"* is to make it so that the outermost cycle is iterating the variable *k*, and the innermost cycle if iterating the variable *i*, because an element at the position *(i,j,k)*, the very next one in the memory if the element at the position *(i+1,j,k)*, meaning that in this way, at any given point in the cycle, it is almost certain that the current element is already in the cache, being contained in the previous block fetched by some previous iteration.

C. Inlining

We declared our most costive function *lin_solve* with *"attribute((always_inline)) inline"*. The keyword *"inline"* suggests the compiler to inline the function, meaning that when it is called, instead of doing all the work associated with function calls, such has creating a new stack frame and pushing arguments, it will replace the function call with the code of the actual function being called. The keyword *"attribute((always_inline))"* brute forces the compiler to always inline this function, regarding other compiler choices or different optimizations.

D. Utilized Flags

We started by adding the *"-Ofast"* flag, wich significantly reduced the runtime of the program, this option applies aggressive optimizations to maximize the performance of the code, simplifying arithmetic operations, ignoring certain standards of accuracy according to *IEEE* standards in floating point operations, like non-associativity, allowing the compiler to rearrange these operations to parallelize these instructions more effectively, bettering the vectorization of the program. In the end we only got a *0.2* from the original value, so we decided to include this flag.

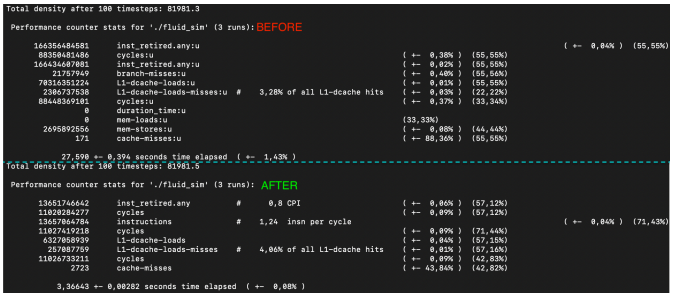
The flags *"-O2"* and *"-O3"* alone greatly increased the runtime of our program, although not as much as the *"-Ofast"* flag, leading us to use the use this last option.

Nextly, we tried the *"-funroll-loops"* flag, in order to unroll the various loops. By reducing the number of iterations in the cycles, this process results in a smalled number of control instructions. This method provided good results, but we noticed that unrolling the loops manually in the code gives even better results, this is because the compiler has limitations when unrolling the loops, it might not be able to detect all the data independencies, making it more efficient to execute this process manually. Given all this, we decided to manually unroll the loops when writing the code, you can see a short example of the loop-unrolling in Section VI.B

The flag *"-ftree-vectorize"* was included, to make the compiler vectorize the code, wich it did successfully. We can verify this by looking at the assembly code generated in Section VI.A, noting the use of instructions like *"movess"* and the *"%xmm"* registers.

Lastly we also added the *"-msse4"* flag, to take full advantage of the *SSE4* instruction.

IV. Performance



Comparing now the execution data between the original program and the final one, the difference is vast.

Firstly, the final program has **2049649779** less cache misses than the original one. This big difference comes from the way we are traversing the vectors in our code.

The number of instructions was also reduced by **152704737939**. This is because of the *"-Ofast"* flag, and because the loops were manually unrolled.

The CPI is approximately **0.8**, primarily due to the use of the *"-Ofast"* optimization flag.

Lastly, we managed a runtime of about **3.3 seconds**, a big difference from the original runtime of **28 seconds**.

V. Conclusion

In conclusion, we completed the task as requested, consolidating our understanding and application of the techniques discussed in the classes. This allowed us to enhance both our practical skills and theoretical knowledge.

VI. Appendices

A. *Example of vectorization code in the function "advect"*

```
movss 0(%r13,%r11,4), %xmm1
mulss %xmm8, %xmm1
mulss %xmm3, %xmm0
addss %xmm12, %xmm1
```

B. *Example of loop unrolling code in the function "add_source"*

```
.L67:
  movss (%r8), %xmm1
  addq  $64, %rax
  addq  $64, %r8
  mulss %xmm0, %xmm1
  addss -64(%rax), %xmm1
  movss %xmm1, -64(%rax)
  movss -60(%r8), %xmm1
  mulss %xmm0, %xmm1
  addss -60(%rax), %xmm1
    ....
  cmpq  %rax, %rdx
  jne .L67
```