Parallel Computing - Phase 1

Gonçalo Costa
PG55944
Universidade do Minho
Braga, Portugal

Marta Rodrigues
PG55982
Universidade do Minho
Braga, Portugal

Pedro Lopes
PG55992
Universidade do Minho
Braga, Portugal

I. Introduction

In the first phase of this parallel computing project, we were tasked with properly analizing and improving the performance of the code provided, a 3D version of Jos Stam's stable fluid solve, applying optimizations keeping it single threaded. This report aims to detail and explain all the strategies applied, in order to demonstrate all the changes that have affected the performance.

II. Source Code Analysis

We started by taking a deep look into the provided source code with the help of the tools *perf* and *gprof*, in order to make an initial profile of the original code.
The *perf* command allowed to us analyze the code's performance in a wider range, by checking CPU cycles, cache misses, branch misses, total execution time, and more. With the help of this tool it was possible to check that it took approximately 26,92 seconds to run.

(só houver tempo meter aqui as complexidades das funcoes principais)

III. Tools Used for Performance Analysis

For the analysis of the code's performance during this phase, as suggested in the assignment, we used *gprof*, *gprof2dot* for visualizing the execution graph, and, most importantly, *perf*. Throughout the project, we primarily used the *perf* command since it provides extensive useful information about the code's performance, such as CPU cycles, cache misses, branch misses, total execution time, and more. Additionally, for the initial analysis of the code and whenever we had doubts about where the code was taking more time, we utilized *gprof* to check the percentage of time consumed by functions and how much time each one took, in order to identify and resolve any bottlenecks caused by functions. Finally, to represent these values graphically and more effectively, we used *gprof2dot*. That said, after improving the readability of the code and completing the initial code analysis, we discovered that the biggest bottleneck was in the *lin_solve* function, and knowing this, we turned our initial focus here in terms of functions.

IV. Compilation Optimization

For the optimization of the code's compilation, we decided to use a series of compilation *flags* that proved useful for optimizing the performance of the project, in addition to code optimizations. The flags used were as follows:

- ***-O3***: Enables level 3 optimization, which includes all possible optimizations (loop unrolling, vectorization, and more) without sacrificing compliance with standards and mathematical precision.
- ***-funroll-loops*:** Performs loop unrolling to improve execution efficiency by reducing CPI caused by reducing loop overhead, which will decrease the number of *cycles*.
- ***-ftree-vectorize***: Enables automatic loop vectorization, which transforms operations that can be performed in parallel (e.g., array summations or multiplications) into SIMD instructions.
- ***ffast-math***: Turns on additional aggressive mathematical optimization flags that might sacrifice accuracy for higher performance. Since the original expected value did not deviate significantly (only 0.1s), we also used this flag.
- ***-march=native*:** Optimizes the code for the architecture of the machine where it is being compiled.
- ***-mavx*:** Enables AVX instructions. If the CPU supports AVX, these instructions will be used.

V. Code Optimization

All of the attempts to improve the optimization of the provided code were done with the main goals to improve its execution time, as well as reduce the number of instructions carried out and the number of execution cycles that the program performs. With that said, all the changes and improvements made to the code are listed as follows:

- Checking and improving Macros : After the initial code analysis and readability improvement were done, we started doing the performance upgrades to the code by analysing the macros the code had and improving wherever we could improve anything significant. **RUN TIME WITH THIS STEP : 8.6 s**

- Loop permutation : With this step, like we saw in the classes, we had the main goals of improving memory and

cache access through better spacial data locality and decreasing memory dependencies. This was done through the reorganization of data acess in triple or more nested loops, which made the different data values closer in memory and more easily accessed between loop iterations, decreasing cache misses massively and close to a second of run time. **RUN TIME WITH THIS STEP : 7.5 s**

- Data Processing through Blocks : Even with the previous change made, it was noticeable that there was still room for improvement to the data and cache accesses in the loops. Thinking about the way the code works, it basically processes a big 3D cube of data throughout its 3 axis and, during this step of the optimization, we noticed that the code was processing the whole block of data all at once, which was overflowing the cache. So, in order to put the cache to better use, we divide the data in smaller cubes, big enough to fit in the cache, which also improved the space/time locality massively and the runtime of the application. Although, with all these advantages, the size of the blocks needs to be carefully calculated, which was something we found out later and will be discussed in a topic below. **RUN TIME WITH THIS STEP : 5.38 s**

- Division Handling and other improvements : From this step forward, we started to run ou t of options on what we could improve in the code in order to decrease even more runtime, cache misses and Clock Cycles. Knowing this, we started checking for division calculations in the code because these are very CPU intensive operations. In doing so, we found out that there was a division operation inside the inner loop of the nested loop in the *lin_solve* function, which meant this operation was slowing down the code a lot CPU wise. To correct this, we made the following change to all division operations we found:

```
float div = 1/c;
x[idx] = (x0[idx] + a * (x_im1 + x_ip1 +
x_jm1 + x_jp1 + x_km1 + x_kp1)) * div;
```

Another change that we made was, in order to decrease the number of times the macro IX was called, we found out the difference between the *IX(i,j,k)* and its neighbours in each axis. With this done, we decreases the number of multiplication operations and replaced them with sum operations and, also, we managed to call the IX expression in each J iteration instead of each I iteration, reducing even more the number of calls.

Sadly we didn't manage to both call IX only in every k iteration due to different results than the expected and not repeat the values of *idx* between iterations, more specifically the values of *idx* and *idx-1* in the next iteration.

Finally, we noticed that the function *set_bnd* was running an *if* statement in every iteration of every loop in its content. In order to fix this, we calculated the 3 boolean values of every *if* expression that were in the loops before actually entering them. This way, we only run the 3 *if* statements once in the function's execution. The calculation of one of the bool values can be seen below :

```
auto neg_mask = (b == 3) ? -1.0F : 1.0F;
```

RUN TIME WITH THIS STEP : 5.05 s

- Change in Block size : During this step, we really weren't finding anything that we could change and improve in the code, after so many changes. So, after some frustration and time, we decided to tweak with the block size and, after changing it from 8 to 4, we ended up doing one of our biggest runtime cuts. This happened because, with shorter blocks, the data is closer to each other, is easier and faster to access and creates less cache overhead. (Talvez possa cortar aqui cenas) **RUN TIME WITH THIS STEP : 4.16 s**

VI. Performance Statistics

Finally, with all these changes applied, we've come to the following statistics:

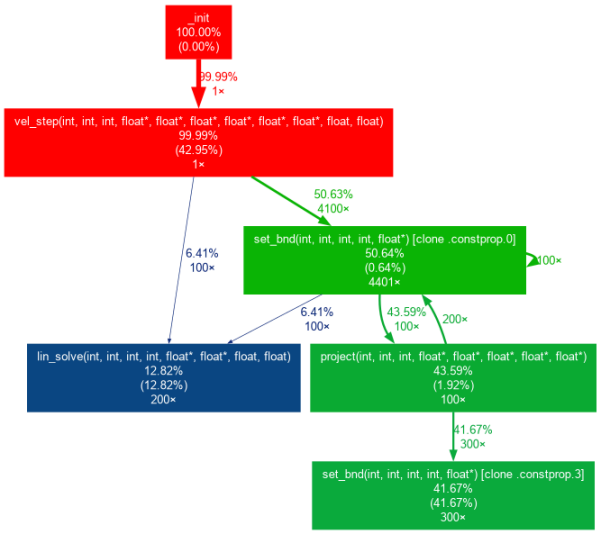

Fig. 1: Performance Graph created by *gprof2dot*.

VII. Conclusion

In summary, this work assignment provided us with a valuable opportunity to learn and apply optimization techniques to a real-world simulation code, which greatly enhanced our understanding of performance optimization. Although we are happy with the execution time we got, we would've liked to obtain an even smaller execution time value. On the other hand, we have to mention that the number of cache misses and cache references we did obtain was pretty nice compared to what we expected. Our project stands as a testament to the profound impact that optimization strategies and compiler-level fine-tuning can have on the efficiency and speed of a program.