Parallel Computing - Work Assignment 1

Gabriela Santos Ferreira da Cunha
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg53829@alunos.uminho.pt

Nuno Guilherme Cruz Varela
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg54117@alunos.uminho.pt

***Abstract*—This document consists in presenting and justifying the optimisations that were made to a provided program which is part of a simple molecular dynamics' simulation code applied to atoms of argon gas.**

***Index Terms*—optimisation, instruction level parallelism, memory hierarchy, vectorisation**

I. Introduction

In this work assignment, it was proposed to explore optimisation techniques applied to a (single threaded) program, using tools for code analysis/profiling and to improve and evaluate its final performance (execution time). Thus, an initial code to simulate particle movements over the time was provided. Using the Newton law, the code follows the next approach:

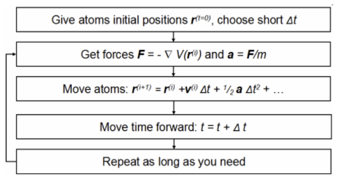
Give atoms initial positions $\mathbf{r}^{(t=0)}$, choose short Δt
Get forces $\mathbf{F} = -\nabla V(\mathbf{r}^{(i)})$ and $\mathbf{a} = \mathbf{F}/m$
Move atoms: $\mathbf{r}^{(i+1)} = \mathbf{r}^{(i)} + \mathbf{v}^{(i)} \Delta t + \frac{1}{2} \mathbf{a} \Delta t^2 + \ldots$
Move time forward: $t = t + \Delta t$
Repeat as long as you need

Fig. 1: Approach to simulate particle movements.

II. Code Analysis

To analyse the code and get its profile we used *perf* and *gprof*, which conceded us an initial idea about the existent bottlenecks on the provided code. Both tools are commonly used and well known for code profiling and analysis.

Accordingly, the first step in order to know the current performance of the provided code was creating a call graph, using *gprof*. On the other hand, *perf* was used during all the development to get the number of clock cycles, instructions and execution time of the different versions.

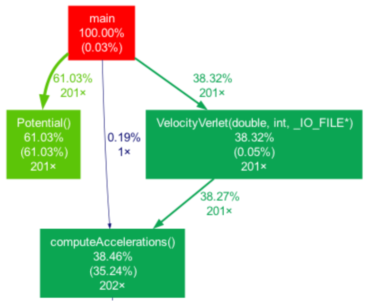

Fig. 2: Call graph using *gprof* tool.

By examining the call graph and inclusive time percentages, we identified the functions that were contributing the most to the program's execution time. As Amdahl's Law predicts, the performance is limited by these sections so we understood `Potential` and `computeAccelerations` functions were the ones that would require us more effort to optimise. Nevertheless, other functions were also optimised, although they did not have as much impact on the final result.

III. Optimisations

A. *Optimisation flags*

To improve ILP, we started by testing compilation with optimisation flags: `-O2`, `-O3` and `-Ofast`. The optimisation level 2 includes various optimisations such as inlining functions, loop optimisations and instruction scheduling. The optimisation level 3 includes more aggressive optimisations and can lead to better performance but at the cost of longer compilation times. The `Ofast` sacrifices some adherence to strict language standards in exchange for maximum performance, so it may perform optimisations that could change the semantics of the code.

B. *Arithmetic operations*

As the exponentiations were all with relatively small integer exponents, we replaced `pow` calls with direct multiplications. Calling `pow` and `sqrt` functions involves overhead related to parameter passing, stack operations and the function call itself, so we tried to avoid calling these functions due to their performance bottleneck.

Still regarding arithmetic operations, we also tried to reduce the number of divisions as it is the most complex basic operation.

For example, in the `Potential` function, we intended to remove the use of the `sqrt` operation due to its inefficiency. Hence, we made the following simplification:

$$4 \times \epsilon \times \left(\left(\frac{\sigma}{\sqrt{r}} \right)^{12} - \left(\frac{\sigma}{\sqrt{r}} \right)^{6} \right) = 4 \times \epsilon \times \left(\frac{\sigma^{12}}{r^6} - \frac{\sigma^6}{r^3} \right) \quad (1)$$

The square root disappears and the $4 \times \epsilon$, σ^{12} and σ^6 can be pre-computed only one time at the beggining of the program because they are constant expressions. To perform this operation, we used the inverse of r to achieve the inverse of r^6 and r^3 by multiplications. This allowed us to perform only one division instead of 2 ($\frac{1}{r^6}$ and $\frac{1}{r^3}$).

C. Potential Cycles

After an observation in the `Potential` function, we concluded that there were unnecessary cycles because the inner loop was starting in the index 0. For example, for the pairs $(1, 3)$ and $(3, 1)$ the computation is the same, so we may simplify and deal with this two pairs in a single iteration multiplying by 2. The inner cycle is now initiating in $i + 1$. This way, both memory hierarchy and ILP were improved by saving a lot of memory accesses that eventually could generate misses and unnecessary calculations.

D. Memory accesses

Increasing the data and temporal locality of a program can have a huge impact on its performance. In this case, the initial code has already a good data and temporal locality since the inner cycles iterate over all the numbers greater than the one that is fixed on the outer loop. Despite that, we could still improve in terms of memory hierarchy, reducing memory accesses, also contributing to minimize cache misses. For that, we used an accumulator to update accelerations in `computeAccelerations`, avoiding repeatedly accesses to the array.

E. Joining functions

One of the major changes to improve the execution time was joining the function that computes the accelerations to the one that calculates the potential energy. Initially, we verified if it was possible to make this improvement, confirming that the information `Potential` needed was not updated after `computeAccelerations` was called. This change could be made because both functions were iterating over the same data structure, `r`, and some of the calculations were the same. As a result, we saved a lot of memory accesses and calculations. It is important to notice that we maintained a `computeAccelerations` version without joining `Potential` because the first time it is called (outside the loop) we do not need to calculate the potential energy.

We also joined the functions to calculate the mean squared velocity and kinetic energy, since they were also iterating over the same data structure and needed the same information to perform the last calculation which differs between each other.

For these two improvements, we turned `PE`, `KE` and `mvs` into global variables to be updated inside each function, since they were updated with the returned value on `Potential`, `Kinetic` and `MeanSquaredVelocity` functions, respectively.

F. Vectorisation

Vectorisation was one of the goals to be achieved by the group. At the beggining, we started to vectorise the code automatically by using the compiler flags `-ftree-vectorize` to auto-vectorise and `-mavx` to use AVX instructions whenever possible. To get the maximum benefit of this we had to refactor all of the data structures by transposing the matrices because the way they were disposed before was a block for the compiler to auto-vectorise.

Although auto-vectorisation had a significant positive impact on the program performance, we wanted to achieve more and we implemented a manual vectorisation in the code by using the AVX instructions provided by `immintrin` library. Since a double is 64 bits and we are operating with 256 bits registers, we can fit 4 doubles ($4 \times 64 = 256$) in a single register. Another way used to improve the performance, was using aligned memory addresses because it leads to more efficient memory accesses. The `MAXPART` was changed to 5004 to be a multiple of 4.

IV. Results discussion

In this section, we will present and discuss the results obtained on the optimisations that were done. The results are displayed on table I and were obtained on SEARCH cluster using `gcc 9.3.0` version. The number of instructions (#I), clock cycles (#CC) and L1 cache misses are measured with a billion as base reference and the execution time in seconds.

In the `0)`, `1)`, `2)` and `3)` optimisations, we analyse the impact that compiler flags may have in the program's performance. As we mentioned before, the usage of `-Ofast` flag could introduce changes on our output values. After checking that there were no significant changes on it, we opted to use this flag due to its better performance.

The `pow` and `sqrt` functions were a huge bottleneck. The program went from 266.37 (844×10^9 CC) to 62.38 (188×10^9 CC) seconds (`0)` and `1)` optimisations) while being compiled with `-O0`. Comparing with the other optimisations not related to compile flags, this one represents the most impactful one, which leads us to conclude that the usage of these functions calls were the biggest impasse to get better performance on the provided code.

As expected there is a huge difference in the number of instructions from the `10)` and `11)` optimisations to the `9)` optimisation due to the vectorisation. This occurs because operations with vectors are being made, which will lead to the instructions' decrease.

As we can visualize, the final execution time of our program with optimisations is about $1.57s$, obtaining a gain of $17,000\%$ compared to the initial code. We are awared that manual vectorisation led to less code readability, but we are also satisfied with the final performance, since it was the main goal in this work assignment and we believe we optimised it as much as we could.

V. Attachments

TABLE I: Results

Optimisation	#I (B)	#CC (B)	L1 Cache Misses (B)	Exec Time (s)	CPI
0)	1248.623	843.614	3.448	266.37	0.7
1)	1015.568	666.507	2.558	208.16	0.7
2)	993.146	776.211	4.224	248.39	0.8
3)	51.344	64.042	0.694	21.63	1.2
4)	308.237	187.658	0.680	62.38	0.7
5)	221.918	123.040	0.420	42.43	0.6
6)	103.955	121.883	0.516	39.32	1.2
7)	103.899	68.014	0.426	21.83	0.7
8)	70.952	46.005	0.324	14.84	0.6
9)	18.230	11.042	0.320	3.52	0.6
10)	6.095	5.641	0.326	1.87	0.9
11)	4.687	4.552	0.327	1.47	1.0

Since optimisation 4), each table line represents an optimisation that was implemented throughout the code at that time. This means, for example, in optimisation 6) the present optimisations were less cycles, less divisions and removal of `pow` and `sqrt` calls.

- **0) :** Initial code without optimisations
- **1) :** Initial code compiled with `-O2` flag
- **2) :** Initial code compiled with `-O3` flag
- **3) :** Initial code compiled with `-Ofast` flag
- **4) :** Without pow and sqrt calls, no optimisation flags
- **5) :** Less divisions, no optimisation flags
- **6) :** Less cycles on `Potential`, no optimisation flags
- **7) :** Memory access reduction, no optimisation flags
- **8) :** Join functions, no optimisation flags
- **9) :** Same as 8), compiled with `-Ofast` flag
- **10) :** Matrices transpose, compiled with `-Ofast -ftree-vectorize -mavx` flags
- **11) :** Manual vectorisation, compiled with `-Ofast -ftree-vectorize -mavx` flags