Parallel Computing - Optimization and Performance Evaluation of a Molecular Dynamics Simulation

1st Francisca Lemos
Universidade do Minho
Braga, Portugal
pg52693

2nd Nuno Costa
Universidade do Minho
Braga, Portugal
pg52698

***Abstract*—This assignment phase focuses on optimizing a single-threaded molecular dynamics simulation program used for studying the behavior of argon gas particles. The program employs the Lennard Jones potential to model the inter-particle interactions, encompassing both force and potential energy. This report details the process of analyzing, optimizing, and evaluating the program's execution time. Through the application of code analysis and profiling tools, this study seeks to enhance the program's computational efficiency.**

***Index Terms*—CPI, performance, metrics, optimization, execution time, instructions, cycles.**

I. Implementation

A. Understanding the initial code

The code given for analysis ran with several numbers of instructions and cycles, as you can in the table below. We generated a gprof report, which provided us with insights into the most inefficient functions that consumed the most time, as shown in Figure 1.

TABLE I

Instructions	*Cycles*	*CPI*
1243571701967	780921294042	0.634

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
74.80	10.24	10.24				Potential()
25.23	13.69	3.45	201	17.18	17.18	computeAccelerations()
0.07	13.70	0.01				VelocityVerlet(double, int, _IO_FILE*)

Fig. 1. Code's profile

B. Removing pow function

After a little research and some tests, we consider it necessary to replace all occurrences of the *pow* function with the respective multiplication because this function is not efficient in terms of execution and can lead to precision errors. With this optimization, our code's CPI has worsened, i.e, we now have more cycles per instruction. However, the number of instructions has significantly decreased.

TABLE II

Instructions	*Cycles*	*CPI*
65921812849	81799676950	1.242

C. Using auxiliary variables

After a brief brainstorming session, we have concluded that incorporating auxiliary variables into most functions is a valuable practice. Using auxiliary variables not only optimizes memory access patterns but also leads to faster execution. This optimization, in turn, significantly enhances performance by minimizing redundant calculations. This leads to the following improvement: While this adjustment may not significantly improve execution time, it is noteworthy that our code now runs in just 25 seconds.

TABLE III

Instructions	*Cycles*	*CPI*
65921464273	80651565016	1.223

D. Transforming the 2D arrays v, r, and a into one-dimensional arrays

In this phase, we transform the 2D arrays into one-dimensional. The 1D approach is likely to be faster for dense matrices since it offers better memory locality and less allocation. This happens because 2D case loses the cache locality and uses more memory. To allocate memory in a 1D array, we use static arrays, because they offer improved efficiency in terms of memory usage and access speed, particularly for small, fixed-size data structures.

TABLE IV

Instructions	*Cycles*	*CPI*
67334280505	83756922914	1,24

E. Improve Potential function

After reviewing the initial call graph, it is clear that the *Potential* function showed inadequate performance. Upon closer examination, it became evident that the function redundantly performed calculations related to the distance formula. We

identified an opportunity to optimize this by calculating key variables, namely x, y, and z. Then, through mathematical calculations, we optimized these operations by eliminating functions such as *sqrt* and unnecessary calculations, effectively removing constants like *epsilon* and *sigma*, as both were set to one. This led to a substantial reduction in computational overhead. You can see the code below:

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
Pot += 4*epsilon*(term1 - term2);
```

The final code remains as follows:

```
rSqd = x*x + y*y + z*z;
rSqd3 = rSqd * rSqd * rSqd;
rSqd6 = rSqd3 * rSqd3;
Pot += 4*((1-rSqd3) / rSqd6);
```

F. Improve computeAccelerations function

The same behavior occurs in *computeAccelerations* function. We noticed some unnecessary calculations, rather than calculating powers of 4 and 7 separately, we combined them into a common denominator by multiplying *pow(rSqd,4)* by the variable *rSqd3* (this variable is calculated above). Afterward, it was a matter of performing some calculations, and the simplification was complete. This allowed us to minimize redundancies. Went from this:

```
f = 24 * (2 * pow(rSqd, -7)
                    - pow(rSqd, -4));
```

to this:

```
f = 24*( (2-rSqd3) / (rSqd6*rSqd) );
```

G. Merge MeanSquaredVelocity and Kinetic functions

In this phase, we have chosen to combine two functions, namely, *MeanSquaredVelocity* and *Kinetic*, due to their similarity. Both of these functions involve similar calculation processes. To optimize our code, we have opted to employ two global variables, *KE* and *mvs*. At the end of each function, we assign the calculated results to these global variables for further use. With this optimization, our code improved in all metrics and now runs in 8,86 seconds.

TABLE V

Instructions	*Cycles*	*CPI*
48030792535	28395955649	0,591

H. Merge Potential and computeAccelerations functions

After examining the code, we noticed similarities in the calculation of potential energy and the acceleration of each atom in the functions *Potential* and *computeAccelerations*, respectively. We made adjustments to the for loops and then proceeded to merge the two functions using the same calculations, using a global variable (*PE*) to store the value of the potential energy of the system. Now our code runs in 7,48 seconds and improved in terms of CPI.

TABLE VI

Instructions	*Cycles*	*CPI*
43893996266	24527076636	0,558

I. Flags

We decided to use certain flags to aid in the optimization of the code. We employed the *-O2* flag to enhance program speed. The *-funroll-loops* flag proves useful as it unrolls loops where the number of iterations is known, although we manually unrolled many loops, we decided to keep this flag. The *-ffast-math* enables some floating-point optimizations and *-ftree-vectorize* enables, when possible, the execution of multiple operations simultaneously, which is quite beneficial in our program. Finally, we made the transition to the AVX instruction set using the *-mavx* flag, which enables operations with large arrays, taking into account the architecture of our machine with *-march=x86-64*, and we further optimized floating-point arithmetic operations with the *-mfpmath=sse* flag to mitigate precision issues. Our final compilation flags are:

```
CFLAGS = -Wall -pg -O2 -funroll-loops
-ffast-math -ftree-vectorize
-mavx -mfpmath=sse -march=x86-64
```

II. Final Performance

After these optimizations, we considered reaching our goal, and we managed to significantly reduce the number of cycles, instructions and seconds. Here are the final metrics generated, running with the following command: 'srun –partition=cpar perf stat -e instructions, cycles /code/src/MD.exe < input-data.txt'.

```
Performance counter stats for '/home/pg52698/code/src/MD.exe':

          14001      cache-misses
    35866694829      instructions              #    1,51  insn per cycle
    23767967502      cycles

      7,317538780 seconds time elapsed

      7,307879000 seconds user
      0,002999000 seconds sys
```

Fig. 2. Final performance

III. Conclusion

Having considered the work as concluded, we believe that there is significant room for improvement and that we could achieve better optimization. Nevertheless, this project has aided us in reinforcing the topics covered in class and understanding how to build code while taking into account memory hierarchy, loop order, and calculation optimizations. In the future, we will continue to apply these insights to create more efficient and optimized code.