Parallel Computing Molecular Dynamics' Simulation

José Mendes
Masters in Software Engineering
University of Minho
Braga, Portugal
pg53967@alunos.uminho.pt

Ana Coelho
Masters in Software Engineering
University of Minho
Braga, Portugal
pg52671@alunos.uminho.pt

***Abstract*—A molecular dynamics simulation aims to compute thermodynamic properties of a material (such as pressure), by taking a given gas to simulate, a given initial temperature and the number density. In this paper we implemented and analysed an optimised version of this algorithm.**

***Index Terms*—molecular dynamics' simulation, code optimisation techniques, loop optimisation, overhead reduction, execution time**

I. Introduction

In this second phase of the project we had to analyze and parallelize the code we altered in the first phase, minimizing once again the execution time. In doing so we were able to explore different parallelization techniques by using *OpenMP*.

II. Implementations

Once again after analyzing the application we identified that the "hot-spot" in this algorithm was present on the function *Potential_plus_Accelaration()*, so our main focus was set on parallelizing it.

A. What to parallelize?

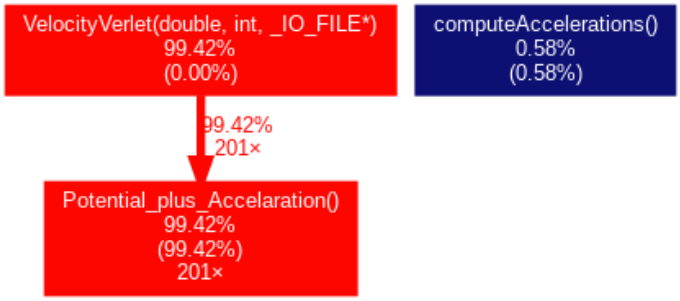

Fig. 1. Output of gprof on sequential execution

The image provided above is the reason we decided to put our efforts on parallelizing *Potential_plus_Accelaration()*.

With this approach we would be able to calculate the potential energy as well as the acceleration of each particle simultaneously and in parallel. This can be achieved by changing the nested loop inside this function in a way that allows for the use of threads. For this and as said previously we are going the resort to the library *OpenMP* in order to successfully implement the needed changes, that consist in the addition of parallelizable regions and the proper handle of memory accessibility.

B. First contact with parallelization of Potential_plus_Accelaration()

At first we tried to add two *OMP* directives to the second nested for loop. Using the *#pragma omp parallel num_threads(2)* we created a parallel region with the entire for loop inside it and next we added *#pragma omp for* to see what type of time would result from this alteration and also what were the values that would differ due to the "data-races" existing in the code. Unlike what was to be expected these changes had no effect on run-time, this happened due to the unresolved "data-races" in the code.

C. Resolving "data-races"

After our first test we realized that handling these "data-races" the right way was the solution to achieve our goal, decreasing the execution time of the algorithm.

We identified data-races in two variables that needed to be dealt with, the global variable *Pot* and the global matrix *a*. Because *Pot* was left as a global variable to threads, each thread was trying to access it's value and as such data-races were being created. Regarding the global matrix *a* because each thread accesses different values of i we expect no data races, but in this case the j variable is the problem. Since j is a private variable of each thread and it goes from i to N when the *a* matrix is being accessed using j there might be another thread working with the same memory space leading to a data race and therefore to a wrong calculation and possible time loss. The situation around *Pot* was relatively easy to solve, since in every iteration it's value was incremented by the variables that were being calculated, applying a *reduction* to it was the way to go.

The same happens to the matrix *a*, but this time the syntax is a little bit different, since it was required to define the size of it which is 5001 lines and 3 columns.

D. Type of distribution

After all this the only thing that remained was to try and understand the best way to handle the distribution of the work

load throughout every thread, and since our nested for loop doesn't follow a "normal" pattern (the inner loop goes from i+1 to N) we decided to use the *schedule(dynamic)* clause.

The reason behind this, is the way that dynamic operates, were when a thread ends its chunk it tries to continue with the next possible one, leading to a more balanced distribution.

The results of our choice will be explained further into the report.

III. Results analysis

A. Speed up

In our program, as mentioned before, our focus was to parallelize the algorithm. Despite this there are still sections where we were unable to do it, for example the initialization of the matrices.

This results in us being unable to achieve the ideal speed-up, as according to the Amdahl's Law a program speed-up is limited by the sequential code. Having this said we tried to calculate the theoretical maximum speed-up we could get by analysing the complexity of the more heavy functions that existed in our code which are, *computeAccelerations* and *Potential_plus_Accelaration*.

Knowing that N was 5000 and the number of iterations is 200 these are the speed-ups according to our calculations:

#Threads	#SpeedUp
2	1.990
4	3.941
8	7.73
12	11.377
16	14.889
20	18.272
40	33.500

TABLE 1: EXPECTED SPEED-UP

B. Work-load

As mentioned earlier, we are using the *schedule(dynamic)* clause, this is being done with the goal of ensuring that each thread has a similar workload, thereby preventing threads from waiting for others to finish. While we successfully achieved a satisfactory distribution of workload, it was anticipated that the initial threads would bear a heavier load than the later ones. This discrepancy can be illustrated by the images in the appendix, obtained through the use of *perf record* and *perf report*. These images allow for a comparison of work-load distribution when using static scheduling versus dynamic scheduling.

C. Parallelization scalability

After using the mentioned directives to resolve potential data races, we tested our code's scalability by measuring it's execution time for 1, 2, 4, 8, 12, 16, 20 and 40 threads and calculated the speed-up relative to it's sequential execution using the script provided to us by the professors.

#T	TExec(s)	#SpdUp	#Ix10^{11}	#CCx10^{11}	#L1-Mx10^{9}	CPI
1	32.683	1	1.240	0.816	1.877	0.658
2	16.549	1.975	1.241	0.823	1.880	0.662
4	8.393	3.894	1.242	0.826	1.883	0.665
8	4.332	7.545	1.246	0.835	1.888	0.670
12	2.985	10.949	1.251	0.845	1.894	0.676
16	2.333	14.010	1.257	0.860	1.899	0.685
20	1.943	16.821	1.265	0.882	1.904	0.698
40	1.845	17.714	1.303	1.655	1.834	1.266

TABLE 2: RESULTS OBTAINED USING DIFFERENT NUMBERS OF THREADS

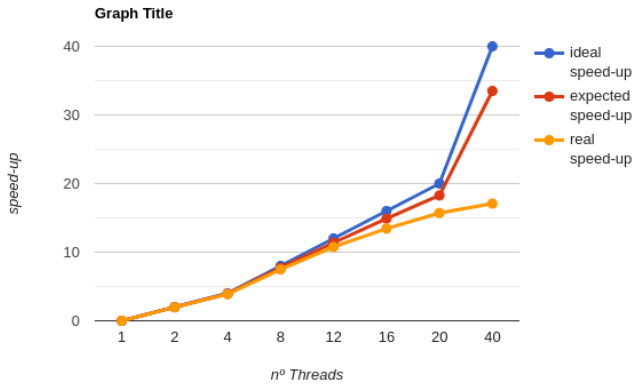

Fig. 2. Speed-up graphic

As it is possible to see the speed-up we obtained is in par with the expected one, only truly deviating once the number of threads starts to grow. The gap between the expected and the actual speed-up increases with the growth in the number of threads, indicating that the impact of the sequential code is bigger than could be calculated since the complexity doesn't provide a real comparison to the actual work performed by each function. Other reasons for the slight difference in results can be the initialization of each thread and their synchronization, both in the reduction of *Pot* and the reduction applied to the *a* matrix.

We don't attribute these differences to loads of cache since it can be seen previously that the L1-Misses don't have much difference from sequential to parallel, maintaining consistency across different thread counts. We also believe that work-load might have some impact but that it might be minimal, given that the work-load varies very little from thread to thread.

IV. Final thoughts

The version we incorporated yielded a better performance compared to the sequential code. The speed-up achieved through parallelization was good, despite not having much improvement after 20 threads . Despite this, we consider our implementation to be good overall, since the real speed-up was similar to the expected one . Our program demonstrated superior performance when utilizing 40 threads as our previous analysis indicates.

V. APPENDIX

A. Workload balance

With 12 threads:

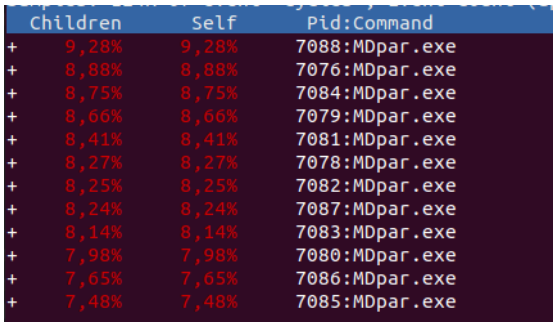

Fig. 3. Output of perf report - using dynamic

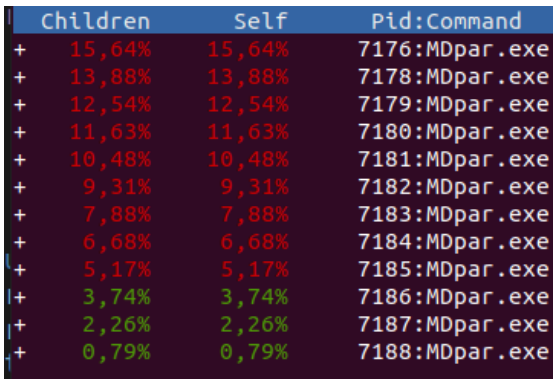

Fig. 4. Output of perf report - using static

With 4 threads:

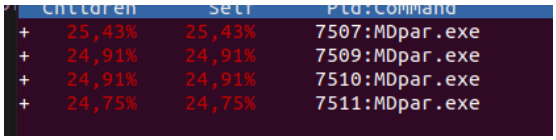

Fig. 5. Output of perf report - using dynamic

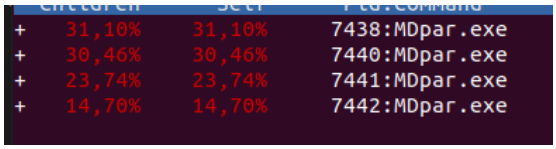

Fig. 6. Output of perf report - using static

B. Expected speed-up

The complexity we based ourselves on:

$$O(N * \frac{N}{2} + It * \frac{\frac{N}{2}}{num_threads})$$

Results obtained:

#T	Comp
1	25125000
2	12625000
4	6375000
8	3250000
12	2208333
16	1687500
20	1375000
40	750000

TABLE 3: RESULTS OBTAINED USING DIFFERENT NUMBERS OF THREADS

C. C. Parallelization scalability

In table 2:

- #T - Number of Threads
- TExec(s) - Execution time in seconds
- #I - Number of instructions x 10^{11}
- #CC - Number of clock cycles x 10^{11}
- #L1-M - Number of misses x 10^{9}
- CPI - Cycles per Instruction