

# Work Assignment - Phase 2

Afonso Ferreira  
Mestrado em Engenharia Informática  
Universidade do Minho  
Braga, Portugal  
pg52669@alunos.uminho.pt

Catarina Costa  
Mestrado em Engenharia Informática  
Universidade do Minho  
Braga, Portugal  
pg52676@alunos.uminho.pt

## I. INTRODUCTION

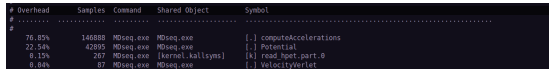
In this project the main goal was to explore shared memory parallelism (*OpenMP*-based) to improve the overall execution time, by developing a sequential and parallel version of the same code previously provided, a molecular dynamics simulation code that models the behavior of argon gas atoms.

Our journey commenced with the sequential implementation, laying the groundwork for its parallel counterpart. While navigating this process, we addressed challenges inherent to parallel programming, such as critical zones and potential data races. Overcoming these intricacies was crucial in refining the parallel implementation and optimizing the simulation's performance. Our focus on shared memory parallelism aimed not only to accelerate computations but also to overcome nuanced hurdles, resulting in a robust and efficient parallelized codebase.

## II. IDENTIFY CODE HOT SPOTS

To enhance code performance, the group began by adjusting the number of particles (global variable  $N$ ) to the required count of 5000. Subsequently, we initiated the optimization process by identifying code blocks with the highest overhead, i.e., those consuming the most computation time.

To achieve this, we employed the profiling and tracing tool *perf* to establish a performance analysis setup, aiding us in identifying hot spots within the application. The results of this analysis are presented in the figure below, depicting the execution outcomes when using the `-O2` compiler flag.



#	Overhead	Sample	Command	Shared Object	Symbol
#					
#	76.85%	148888	MDseq.exe	MDseq.exe	[.] computeAccelerations
#	22.14%	42889	MDseq.exe	MDseq.exe	[.] Potential
#	0.15%	282	MDseq.exe	kernel32.dll	[K] ReadFile.part.0
#	0.04%	87	MDseq.exe	MDseq.exe	[.] GetCpuId

Fig. 1. Perfreport results

## III. OPTIMIZATIONS

### A. Sequential MDseq.cpp

After a comprehensive code review, the team identified an opportunity for improvement overlooked in the initial project phase. This enhancement involved consolidating two functions, *Potential()* and *computeAccelerations()*, into a unified function. Formerly handling the computation of total potential energy and forces separately, the merger not only enhanced code readability but also delivered a performance boost.

To accommodate this change, a new global variable, *Pot*, was introduced, necessitating adjustments to the main function to avoid errors.

Both functions, dealing with particle interactions and featuring an overlap in computations, were merged to eliminate redundancy, resulting in a more concise and efficient codebase.

The unified function not only streamlines maintenance but also adheres to software development best practices, fostering a cleaner, modular code structure. This strategic consolidation reflects our commitment to code optimization, anticipating positive impacts on both readability and potential runtime efficiency.

Besides this major change, a compiler flag, `-Wall`, was introduced to check for all warnings, leading to minor code adjustments, including resolving inconsistencies and removing unused variables.

### B. Exploring parallelism with OpenMP in MDpar.cpp

To develop the parallel version of the application, we analyzed the previously mentioned sequential code implementation to identify performance bottlenecks in terms of execution time. Afterwards, we took this sequential implementation and introduced *OpenMP* directives and efficiently distributed the computational load among various parallel threads.

Our primary focus was on implementing the *OpenMP* directives within the loop cycles of the *computeAccelerations()* function, as this is where the most significant computational load is concentrated.

```
#pragma omp parallel for reduction(+:Pot) reduction
(+:a[:MAXPART][:3]) private(j, rSqd, rSqd4,
rSqd7, rij, f) schedule(dynamic, 40)
for (i = 0; i < N-1; i++)
{
    for (j = i + 1; j < N; j++)
    {
        rSqd = 0.;
        // Calculate the position of atom i relative
        to atom j
        rij[0] = r[i][0] - r[j][0];
        (...)
```

The group strategically positioned the *pragma* directive at this precise location due to its alignment with the peak computational demands of the code. A meticulous examination of this code segment led to the determination that only the outer loop possesses the capacity for seamless parallel execution.

By using the *pragma*, it was possible to parallelize the loop iterations where the computation of forces and accelerations between pairs of particles takes place. This section is a computationally intensive part of the code, and parallelization can lead to significant performance gains. The reduction clauses ensure that the shared variables *Pot* and *a* are correctly updated across all threads preventing data races.

We did not parallelize the inner loop because the calculation of particle accelerations depends on the results of the previous iterations. Specifically, the acceleration calculation for particle *i* depends on the positions and distances involving all particles with indices greater than *i*. Since these calculations have data dependencies, using the parallelism in this section, would require careful synchronization mechanisms and could potentially introduce race conditions, making it more complex and error-prone.

Reasons why we applied the pragmas:

- **#pragma omp parallel for:** This directive instructs the compiler to immediately parallelize the for loop. It implies that iterations of the loop can be executed concurrently by multiple threads.
- **reduction(+:Pot):** We used this clause since each thread will have its local copy of *Pot*, and at the end of the parallel region, these local copies will be combined into a single global value. This results in the reduction of the *Pot* variable in an addition context.
- **reduction(+:a[:MAXPART][:3]):** Similar to the previous clause, this one specifies a reduction for the acceleration array *a*. Its notation signifies that the reduction is applied to each element of the two-dimensional array. It's important to note that each element of the matrix is treated as an independent variable, and the local copies from each thread are combined at the end of the loop.
- **private(j, rSqd, rSqd4, rSqd7, rij, f):** We put these variables at private for each thread, so they could have their own local copies for each thread, preventing data races.
- **schedule(dynamic, 40):** We used dynamic scheduling because it allows the loop iterations to be distributed dynamically among the available threads. The parameter 40 indicates the number of loop iterations assigned to each thread at a time.

#### IV. RESULTS

To analyze the behavior of the code under different thread configurations, we executed the batch file *.script.sh* on the *SeARCH cluster*. The table below presents the execution times for the parallel and the speed up values for the different number of threads. Two graphs depict Scalability (Speedup vs Threads) and Efficiency (Efficiency vs Threads).

The two following graphs were obtained by running the MDpar.exe in the *SeARCH cluster*.

We also ran our sequential code on the *SeARCH cluster* a indefinite number of times. For reference, its best time was 48.914 seconds.

Number of Threads	Execution Time	
	Parallel(s)	SpeedUp(s)
4	12.222s	1.000
8	6.215s	1.9665
12	4.205s	2.9065
14	3.630s	3.3669
18	2.859s	4.2749
22	2.635s	4.6383
24	2.613s	4.6774
26	2.626s	4.6542
30	2.558s	4.7780
34	2.545s	4.8024
36	2.534s	4.8232
40	2.469s	4.9502

It was observed that from a specific juncture onward, augmenting the number of threads in the *OpenMP* parallelization does not yield a substantial improvement in speedup, as corroborated by the Efficiency Analysis below.

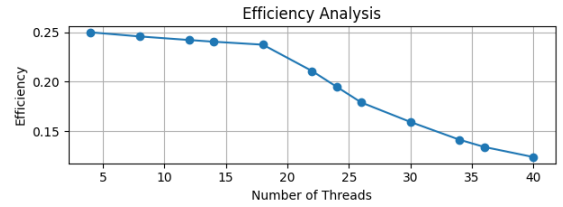


Fig. 2. Efficiency Analysis

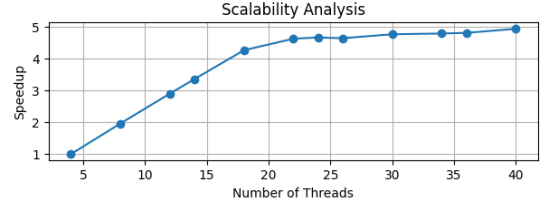


Fig. 3. Scalability Analysis

#### V. REFLECTIONS ON THE SEARCH CLUSTER

During the conception of this project we applied the chosen parallelization approach by incorporating *OpenMP* directives. The code underwent an optimization specifically to the *SeARCH cluster*, taking into account the architectural nuances and the resources available on the compute node within the *cpar* queue.

#### VI. CONCLUSION

In conclusion, our *OpenMP* exploration significantly improved parallel computing understanding. Successful directive application boosted computational efficiency, proving efficacy in real-world simulations.

Looking ahead, we aim to further leverage *OpenMP* for memory hierarchy optimization, seeking additional performance gains. While manual vectorization is considered, its complexity necessitates careful evaluation.

This task improved our *OpenMP* skills and sparked excitement for continuous optimization. The knowledge acquired and positive results lay a strong foundation for future work in the dynamic field of parallel computing.