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

I. Phase 1 - Explore optimisation techniques applied to a single threaded program

***Abstract*—This comprehensive assignment encompasses the optimization of a single-threaded molecular dynamics simulation program focused on studying argon gas particle behavior. The initial phase involves analyzing and optimizing the sequential code, employing the Lennard Jones potential to model inter-particle interactions in terms of force and potential energy. Code analysis and profiling tools are utilized to enhance computational efficiency. The subsequent phase explores the introduction of shared memory parallelism using OpenMP directives, aiming to improve execution time and assess the benefits of parallel programming. The final stage delves into the design and implementation of a new version tailored for accelerators, specifically GPU with CUDA. This approach spans sequential, parallel, and accelerator-based optimizations, providing a comprehensive evaluation of the program's performance across various computing architectures.**

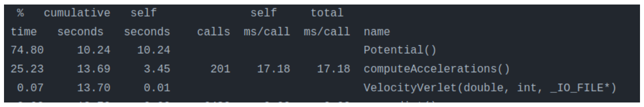
***Index Terms*—CPI, performance, metrics, optimization, execution time, instructions, cycles, OpenMP, parallel, sequential, threads, speed up, CPI, performance, metrics, optimization, execution time, instructions, cycles. cache-misses.**

A. Implementation

1) 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



```
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  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

2) 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

3) 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

4) 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.

5) 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

TABLE IV

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

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);
```

6) 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) );
```

7) 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

8) 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

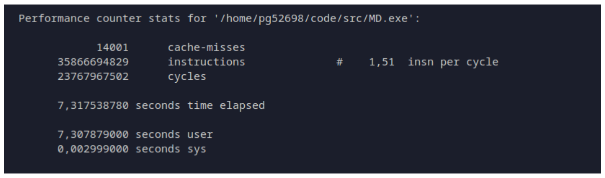
Instructions	***Cycles***	***CPI***
43893996266	24527076636	0,558

9) 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
```

B. 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

C. Conclusion of Phase 1

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.

II. Phase 2 - Explore shared memory parallelism with OpenMP

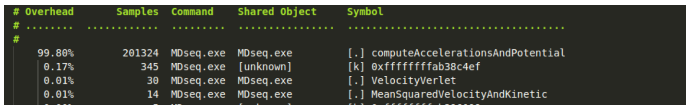
A. *Sequential code*

After increasing the number of atoms from the previous phase, the sequential code deteriorated in terms of execution time, that now runs in 50.93 seconds. We attempted to improve the code compared to the previous phase by adjusting some calculations that were being unnecessarily repeated. Following these adjustments, we moved on to the objectives of this second phase.

B. *Identify hot-spots*

Before diving into parallelizing the code, we took a close look at the sequential version. We used the *perf record* command to figure out how much time each function was taking, helping us identify which functions were consuming the most execution time. This step is important because it helps us find the parts of the code that could really benefit from parallelization. The result was as follows:

- *computeAccelerationsAndPotential*: With an overhead of 99.8% and a time complexity of O(N²), this function involves repeated computation of expressions and multiple accesses to the same data structure. In this phase, we will attempt to parallelize this function, allowing multiple threads to execute these calculations and accesses concurrently and independently.



```
# Overhead       Samples  Command    Shared Object     Symbol
# ........  ............  .........  ................  ..................................
#
    99.80%        201324  MDseq.exe  MDseq.exe         [.] computeAccelerationsAndPotential
     0.17%           345  MDseq.exe  [unknown]         [k] 0xffffffffab38c4ef
     0.01%            30  MDseq.exe  MDseq.exe         [.] VelocityVerlet
     0.01%            14  MDseq.exe  MDseq.exe         [.] MeanSquaredVelocityAndKinetic
```

Fig. 3. Code's profile

C. *Analyse alternatives to explore parallelism*

The criterion for selecting functions to parallelize was based on their computational weight within the overall program, as per the earlier analysis. Subsequently, we will elaborate on alternatives considered for implementing parallelism in the single function that was previously analyzed.

We began by applying *#pragma omp parallel* to create separated threads, where each had a copy of the local variables. This allowed each thread to operate independently of the others. We also employed *#pragma omp for reduction(+:Pot)* to handle concurrent modification of the variable *Pot* through different threads. Consequently, at the end of the parallel region, we summed all the copies into a single variable.

Next, we evaluated the different forms of thread scheduling for 'for' loop iterations. We began by testing with the static schedule, which proved to be the best alternative in terms of cache-misses, instructions, and cycles. Since all threads have approximately equal cost, it makes sense to divide the total number of iterations by the number of possible threads. As for dynamic scheduling, it significantly worsens compared to others. This happens because, at the end of each iteration, it looks for available threads and assigns operations to them, increasing overhead. The increase in cache-misses is significantly higher, as threads may work on different portions of the iteration space at different times, leading to unpredictable memory access patterns. Finally, we tested the guided schedule. Despite being a better scheduling option than dynamic, it shows a decline compared to static, most likely due to the poor performance of dynamic. Based on this, we implemented the static schedule approach.

D. *Scalability Analysis*

The following graph illustrates the execution time of the parallelized program based on the number of threads used. The optimal execution time occurs at the maximum number of threads (40). However, beyond 16 threads, the code doesn't exhibit a significant difference in execution time. This phenomenon may occurs because a substantial part of the program isn't parallelized (Amdahl's Law).

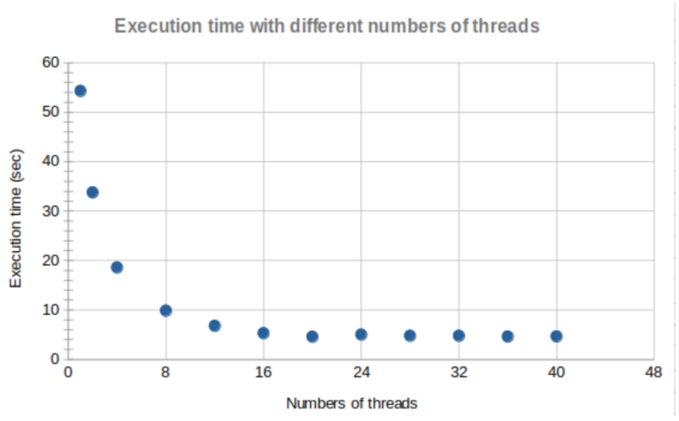


Fig. 4. Execution time with different nr of threads

E. *Measure and Discuss Performance*

1) Speedup: To analyze the performance and scalability evolution of the program, we present the following graph. It illustrates the speedup of the program (red curve) as a function of the number of threads used, alongside the expected speedup according to Amdahl's Law (blue curve).

By examining this plot, we can observe that the optimal speedup occurs between 16 and 20 threads. The code showed

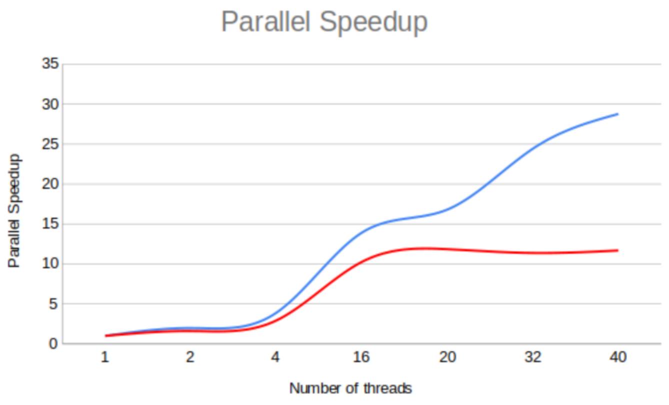
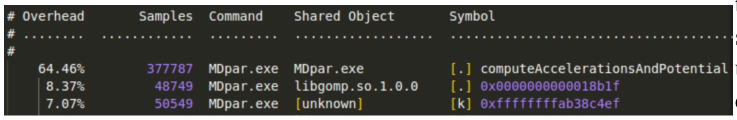


Fig. 5. Parallel speedup

significant improvement when using 4 to 16 threads. However, as the number of threads increased to 28 until 40, the speedup experienced only marginal improvement. This difference in gain is attributed to the substantial number of sequentially executed instructions, making it challenging to parallelize the entire code. Consequently, the speedup is limited (Amdahl's Law), as the sequential portion of the code limits the potential for parallelization. In conclusion, we can see when the number of threads is 18 and 40(estimated) there is a balance between the resources distribution and the paralellism, making the program faster than with other numbers of threads.

2) Function Overhead and Execution Time: After parallelizing the previously mentioned function, we believe to have significantly reduced its overhead. It has now decreased to 64.46%. As for the execution time of the parallelized code, we have significantly improved it, now achieving a runtime of 3.7 seconds.



```
# Overhead       Samples  Command    Shared Object       Symbol
# ........  ............  .........  ..................  ....................................
#
    64.46%        377787  MDpar.exe  MDpar.exe           [.] computeAccelerationsAndPotential
     8.37%         48749  MDpar.exe  libgomp.so.1.0.0    [.] 0x000000000018b1f
     7.07%         50549  MDpar.exe  [unknown]           [k] 0xffffffffab38c4ef
```

Fig. 6. Final code report

F. Conclusion of Phase 2

Given the completed work, we believe that we could achieve better performance in terms of speedup and execution time. Initially, we encountered some difficulties working with OpenMP primitives, especially deciding which ones to use. However, after analyzing the sequential code, we quickly began to understand the problem. This phase helped us comprehend shared memory and its advantages.

III. Phase 3 - Design and Implement a New Version for Accelerators (GPU with CUDA)

A. Analyse code

In this final phase of the project, the group tried to implement a Cuda architeture developed by NVIDIA, which allows the execution of code in parallel on graphics processing devices(GPUs). To accomplish this, we had to make some changes to the code, that will be explained in the following pages. After analyzing Figure 6, which corresponds to the final code report, we observed that the overhead of the computeAccelerationsAndPotential function remains high. Consequently, we have opted to execute this function on the GPU and transfer its variables to the device.

B. Move data to the GPU

Initially, we transfer some relevant data to the GPU for better optimization. The selection of these data must consider dependencies between them, computational requirements, and the usefulness of the results obtained on the host. This operation can be performed using the *cudaMemcpy* function. One of the variables to be moved is the position vector (r). Transferring r to the device allows the GPU to efficiently access particle positions during parallel computation. We also allocate space for the variable a on the device because it enables parallel computation of forces, optimizing GPU performance. Finally, we transfer the potential energy (PE) to the device, allowing the GPU to update it during computation.

C. Returning to Matrices

After a brief investigation, we have decided that to avoid race conditions, it would be preferable to revert to passing the vectors *Position*, *Velocity*, and *Acceleration* as matrices. In some cases, the nature of data access patterns may favor the use of matrices or vectors. For instance, if your algorithm can be efficiently parallelized along a specific dimension, organizing your data as matrices may make sense to facilitate parallelism along that dimension.

D. Shared memory

The kernel function user shared memory to store a subset of particle positions *rf* in the variable *shared_r*. The purpose of using shared memory in this context is to allow each thread within the same block to be accessed at a faster level, by that means reducing global GPU traffic and optimizing memory accesses. Althought, leaves a question hanging in the air: Why not share potentials *temp_Pot* using shared memory? unlike positions, the potential is a single value computed independently for each particle, and shared memory is beneficial when multiple threads within a block need to access and share the same data.

TABLE VII

Number of atoms	***Threads per Block***	***real seconds***
1000	256	4.244
5000	256	4.373
10000	256	3.924

E. Perfomance analyse

Here is an example of tests with a different number of atoms, running with the same number of threads.

F. Conclusion oh Phase 3

With the accomplishment of this practical work, the group concludes that it was possible to consolidate knowledge taught during the classes of Parallel Computing. It was also a great way for us to explore *CUDA* and practice more about design and implement a new version for accelerators. Altought, some difficulties were encountered during the develop stage. Despite everything, we consider having achieved the requested objectives, and we were able to optimize the code to 4,373 with 256 threads.

G. Final Conclusion

In conclusion, the three phases of this practical work provided our group with a comprehensive learning experience. From foundational concepts and shared memory parallelization to GPU acceleration with CUDA, each phase contributed to our understanding of parallel computing paradigms. Despite encountering challenges, we successfully optimized the code, achieving notable speedups, while recognizing room for further improvement.