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 involves analyzing sequential code and improving its execution time through the implementation of shared memory parallelism, specifically using OpenMP directives. The main goal of this project was to evaluate the benefits of parallel programming in shared memory. This report will concentrate on evaluating the sequential version, proposing modifications to transform it into a parallel version with improved execution time, and discussing various metrics.**

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

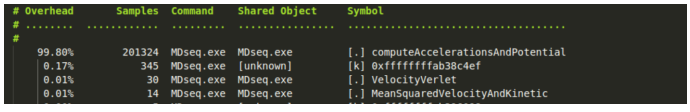
I. Sequential code

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.

II. 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. 1. Code's profile

III. 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.

IV. 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).

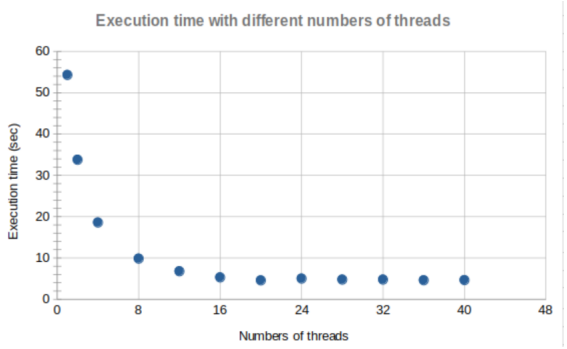


Fig. 2. Execution time with different nr of threads

V. Measure and Discuss Performance

A. 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).

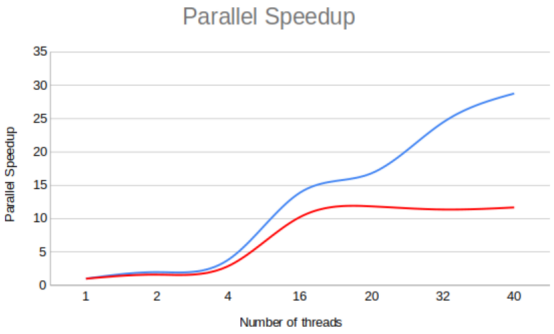


Fig. 3. Parallel speedup

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

B. 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    [.] 0x0000000000018b1f
     7.07%         50549  MDpar.exe  [unknown]           [k] 0xffffffffab38c4ef
```

Fig. 4. Final code report

VI. Conclusion

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.