Optimizing Multi-Threaded Molecular Dynamics Simulation for Argon Gas

Eduardo Figueiredo
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52679@uminho.pt

Gonçalo Senra
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52683@uminho.pt

***Abstract*—This article discusses the techniques used to transform a single-threaded molecular dynamics simulation program into a multi-threaded one. The goal of this work assignment is to improve program performance, reduce execution time, avoid data races, and consequently avoid data corruption. After the transformation, we analyze the speedup achieved by parallelizing the code and we compare it to the theoretical speedup for each number of threads.**

I. Introduction

The program we analyzed and optimized is a simple molecular dynamics simulation code for argon gas. The original version of the code is available in the Foley Lab/Molecular Dynamics, specifically in the "Simple Molecular Dynamics" section, and underwent analysis and improvement with the aim of enhancing performance and execution time in the previous goal. A key objective was the parallelization of the code with a primary focus on the 'ComputeAccelarationsPotential' function, identified as a critical hotspot.

***Index Terms*—Optimizing, Instructions, Execution time, Performance, Code, Accelerations, Flags, Threads, Data race, Parallel, Parallelization, Multi-threaded, Speedup, Real gain, Optimal gain, Reduction, OpenMP, SeARCH, Hotspot**

II. Hotspots identification

This section discusses the identification of the code blocks with the highest execution time, although we tested them in various locations. First, we identified the following hotspot, however after parallelizing in additional locations and not observing a noticeable speedup gain, we decided not to parallelize them.

A. ComputeAccelerationsPotential Function

In the last work assignment, we were asked to optimize this Molecular Dynamics program without parallelization, so one of the strategies adopted was to merge the two most complex functions (ComputeAccelerations() and Potential()) into one (computeAccelerationsPotential), which saved us a bunch of unnecessary cycles. That solution left us with a major code block that was consuming 99.43% as we can see in the call graph 4 of the execution time and, as a consequence of this fact, the target of parallelization had to be this function.

III. Parallelization

In this section, we explain the formal details of the techniques and 'tags' used to transform a single-threaded program into a multi-threaded program with OpenMP.

A. #pragma omp parallel

In the first place, we selected a section for parallelization, this tag (with braces), is responsible for indicating the place where the threads will be created/launched and terminated during the execution.

B. #pragma omp for schedule(dynamic) reduction(+:Pot)

a) for: This directive name indicates an assignment of loop iterations to threads. In this directive, we had to make a choice between putting it in the first 'for' loop, or in the second loop. We decided to place it in the first one because the second loop has more dependencies than the first, so positioning it in the first loop will allow the threads to run more independently which will make the parallelization more efficient. Another reason that led us to place it in the first 'for' loop was the scalability would be more uniform, as the number of instructions per cycle in the first loop collectively is more identical among threads.

b) schedule(dynamic): The keyword schedule indicates how the loop iterations will be distributed to the threads, in this case, we chose dynamic distribution so that the threads request a new chunk to handle every time they finish the current chunk. We could also specify the chunk size, but we didn't get better execution times due to those changes, so we kept the default size. Additionally, we conducted tests with other scheduling types, such as static, auto, and guided. However, in each experiment with these configurations, we noticed that the execution time increased, or data race conditions arose among the shared data between threads. This analysis led us to use the dynamic distribution option, which provides a satisfactory balance between performance and avoiding data race problems.

c) reduction(+:Pot): This is a crucial step in code parallelization and data race avoidance, the reduction tag will create copies of the indicated variable for each thread. At the end of the parallelized section, the operation indicated before ':', (in our case "sum"), will be applied to all copies and store the

result in the original variable. This technique is used to avoid multiple threads' access to the same variable and consequently prevent the corruption of the indicated variable. Furthermore, the execution time will be improved because we don´t have to use any synchronization technique and hold back threads that could be running their tasks independently.

C. Matrix 'a' reduction and #pragma omp critical

a) Matrix 'a' reduction: In the context of reducing the 'a' matrix, each thread in the parallel section maintains its private copy of the 'private_a' array. This private array is used to independently compute contributions to the global matrix without causing contention or data races between threads. After a parallel loop, these private copies are combined and the global matrix is updated with the aggregated results. We implemented this private array because we were using an earlier version of OpenMP that did not support the direct reduction of arrays. Newer versions (4.0) allow direct reduction of arrays. However, to overcome this limitation, we manually created and maintained a private copy during parallel processing to ensure that the global matrix was updated correctly.

b) #pragma omp critical: This is used to create a critical section, allowing only one thread at a time to execute the block of code. In our case, it helps protect the process of updating the global matrix a. Without this critical section, multiple threads could modify the shared matrix simultaneously, potentially leading to data races. Critical sections ensure that updates to the global matrix occur synchronously, preventing conflicts and maintaining the integrity of shared data.

IV. Analysis of Parallelization

Further, we will analyze the practical and theoretical advantages of program parallelization compared with sequential execution.

A. Execution time evolution

The execution time of the program diminishes every time the number of threads is increased, especially when the number of threads is low after the number of threads reaches 20-25 the execution time stops improving, this behavior was expected and could be explained by the machine's limitations, and the high demand on the SeARCH's capacities. On top of that, the code isn't fully optimized which can explain the fact that the speedup curve didn't reach the optimal gain curve (more detailed explanation about speedup in the next subsection).

B. Speedup evolution

By dividing the parallel execution time by the sequential execution time, the speedup is computed. Amdahl's law establishes the theoretical maximum gain in the case of 20 virtual cores and 20 physical cores, implying that 20 should be the ideal acceleration. However, because of racial disparities and Amdahl's law, which restricts the performance of contiguous program components, the actual performance benefits can be less in real-world situations. Suboptimal gains, stagnation, and slowdowns can be caused by practical issues including

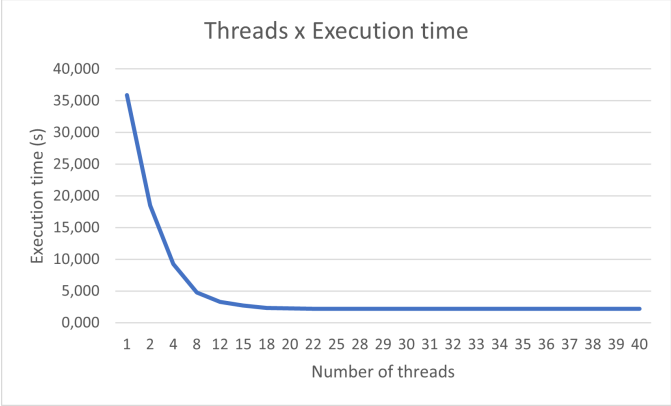

Fig. 1. Threads X Execution Time

resource contention, search system load, and scheduling challenges, particularly after 25 threads. These elements draw attention to how complicated parallel computing is. The benefits of parallelization can be limited by Amdahl's rule, which can lead to severe sequential corruption. Therefore, it's crucial to take into account all relevant practical considerations and optimize.

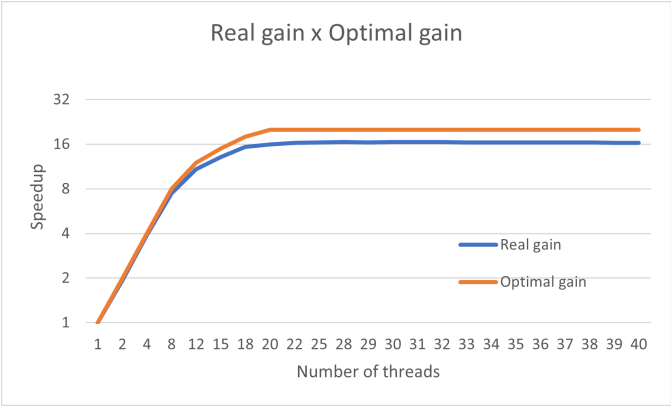

Fig. 2. Real Gain X Optimal Gain

V. Conclusion

After many attempts to parallelize the program with OpenMP, we found the best solution for this algorithm, we decreased the execution time without a significant loss of precision. The biggest challenge of this work assignment was the data race avoidance, as we mentioned earlier, and being constantly waiting in the SeARCH's queue.
In summary, we think that the goal of this work assignment was achieved, and the program was parallelized with success.

References

[1] Computer Organization and Design: The Hardware/Software Interface, David Patterson and John Hennessy, 5th Ed., Morgan Kaufmann, 2013
[2] Course slides
[3] http://search6.di.uminho.pt/wordpress/?page_id=55
[4] https://en.wikipedia.org/wiki/Parallel_computing

VI. ANNEXES

Physical cores	Threads	Exec time (s) 1	Exec time (s) 2	Exec time (s) 3	Mean (s)	Speedup
1	1	35,904	35,909	35,895	35,903	1
2	2	18,235	18,173	19,126	18,511	1,939496525
4	4	9,232	9,225	9,224	9,227	3,891044399
8	8	4,783	4,791	4,789	4,788	7,498990462
12	12	3,306	3,319	3,320	3,315	10,83036702
15	15	2,722	2,731	2,758	2,737	13,11752527
18	18	2,343	2,338	2,335	2,339	15,35176739
20	20	2,208	2,280	2,249	2,246	15,98753154
20	22	2,171	2,201	2,235	2,202	16,30210383
20	25	2,161	2,234	2,168	2,188	16,41139723
20	28	2,173	2,166	2,165	2,168	16,5602706
20	29	2,176	2,178	2,176	2,177	16,49433384
20	30	2,170	2,173	2,164	2,169	16,55263562
20	31	2,169	2,183	2,168	2,173	16,5196319
20	32	2,169	2,169	2,179	2,172	16,52723646
20	33	2,173	2,178	2,184	2,178	16,48171385
20	34	2,173	2,180	2,191	2,181	16,45904645
20	35	2,174	2,175	2,190	2,180	16,47163175
20	36	2,179	2,178	2,191	2,183	16,44899206
20	37	2,180	2,191	2,177	2,183	16,44899206
20	38	2,182	2,183	2,186	2,184	16,44145932
20	39	2,188	2,203	2,199	2,197	16,34415781
20	40	2,190	2,198	2,189	2,192	16,37646343

Fig. 3. Execution time

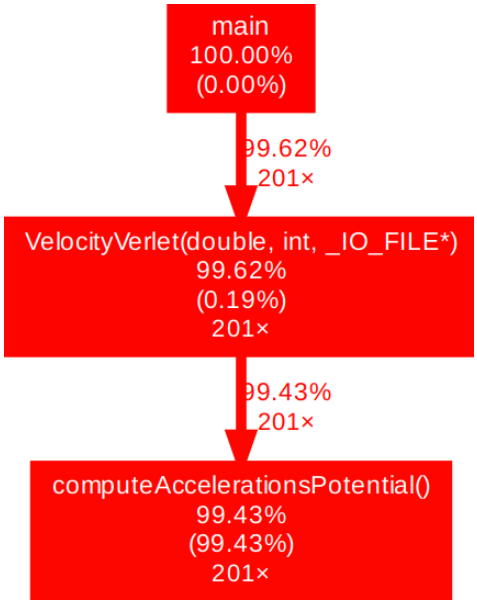

Fig. 4. Optimized call-graph