Parallel Computing
Simple Molecular Dynamics

Gustavo Pereira
Informatics Department
University of Minho
Braga, Portugal
pg53723@alunos.uminho.pt

Nelson Almeida
Informatics Department
University of Minho
Braga, Portugal
pg52697@alunos.uminho.pt

***Abstract*—This paper presents an analysis and optimization program aimed at enhancing the performance of a simple molecular dynamics simulation code designed for the study of Argon gas atoms. The original code, FoleyLab/MolecularDynamics: Simple Molecular Dynamics, employs a generic approach to simulate the dynamic behavior of particles over discrete time steps, adhering to Newton's laws of motion. The given code lacks efficiency due to its exhaustive loops that result in a heavy computational load leading to substantial execution times. In this paper, we will analyze the code and apply optimizations utilizing OpenMP following the previous assignment enhancements.**

I. Parallelization with OpenMP

To optimize the performance of our program, we targeted the most time-intensive function, `computeAccelerationsAndPotential`, which consumed 99.88% of the overall execution time, through performance analysis tools such as `perf report` and insights gained from prior assignments.

```
Samples: 29K of event 'cpu_core/cycles:Pu/', Event count (approx.): 31147308251
Overhead  Command    Shared Object          Symbol
  99,88%  MDseq.exe  MDseq.exe              [.] computeAccelerationsAndPotencial()
```

Fig. 1. Execution Overhead Analysis

Upon examining the sequential code, we identified a potential optimization concerning the nested loops. Specifically, we eliminated conditional statements within these loops, transforming the loop complexity from $N \times 3 \times N \times 3$ to $N \times 3 \times N$.

However, the introduction of parallelization presented a critical issue related to potential data races. Simultaneous access and modification of the same array `a` by multiple threads could yield incorrect results. To rectify this, we devised a solution using an auxiliary matrix, `aTemp`, where each thread accesses its own array, mitigating potential data races. For this, we initialized the matrix with `malloc`, leveraging the default value of 0 as the neutral element for operations with this auxiliary array, eliminating the need for additional loops to set these values to 0.

```
int numThreads = omp_get_max_threads();
double** aTemp;
aTemp=(double**)malloc(numThreads * sizeof(double*));
for (i = 0; i < numThreads; i++) {
    aTemp[i]=(double*)calloc(N * 3, sizeof(double));
}
```

This parallelized section addressed data race concerns but introduced a new challenge: aggregating values from the auxiliary array into the original array of accelerations (`a`). The order of these floating-point operations mattered, necessitating an additional nested loop, this time involving the number of threads (`numThreads`) multiplied by `numThreads * N * 3`.

To resolve the data race issue with the variable `PEA`, altered by each thread, we employed the OpenMP primitive reduction, ensuring a swift and efficient solution to the problem.

Regarding thread execution scheduling, considering options of static, dynamic, or guided scheduling, and acknowledging the higher overhead of dynamic and guided scheduling, we tested their use with a focus on overall performance. Consequently, we opted for dynamic scheduling.

```
#pragma omp parallel num_threads(numThreads)\
private(privateVars)
{
    threadNum = omp_get_thread_num();
    #pragma omp for reduction(+:PEA)\
    schedule(dynamic, numThreads)
    for (i = 0; i < N * 3 - 4; i += 3) {
        for (j = i + 3; j < N * 3; j += 3) {
            // auxiliary computing

            // storing the values
            // in the auxiliary matrix
            aTemp[threadNum][i] += tempx * f;
            aTemp[threadNum][j] -= tempx * f;
        }
    }
}
```

II. Performance

A. Expectations

After the implementation, we used Amdahl's law to get a expected speedup for our program. The speedup according to Amdahl's Law is given by the following formula:

$$\text{Speedup} \leq \frac{1}{(1-P) + \frac{P}{S}} \tag{1}$$

Since we only parallelized the hot-spot which took 99.88% of the overall execution time, replacing the P in the formula above we get the following equation:

$$\text{Speedup} \leq \frac{1}{(1 - 0.9988) + \frac{0.9988}{S}} \tag{2}$$

This gives us the expected speedup in relation to the number of running threads.

TABLE I
PERFORMANCE METRICS FOR DIFFERENT NUMBERS OF THREADS

Num. Threads	Expected Speedup
1	1
2	1.997
4	3.985
8	7.933
10	9.893
16	15.717
20	19.554
32	30.852
40	38.211

It's clear that the expected speedup is different from linear since we aren't parallelizing the complete program but only the targeted hotspot.

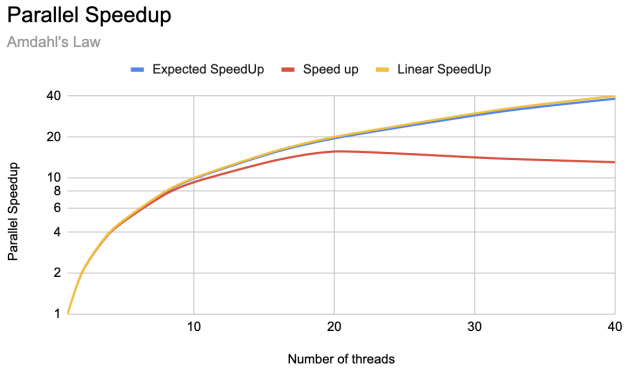
With this information in mind, we started testing the parallel implementation for the same number of threads as our expected results and got the following results:

TABLE II
PERFORMANCE METRICS FOR DIFFERENT NUMBERS OF THREADS

Num. Threads	Avg. execution time	Speed Up
1	35,071	1,00
2	17,633	1,99
4	8,902	3,94
8	4,595	7,63
10	3,767	9,31
16	2,577	13,61
20	2,240	15,65
32	2,528	13,87
40	2,684	13,06

We calculated our speedup by dividing the baseline which was our sequential implementation by our parallel time.

The data above can be better visualized in the following graph.



III. RESULT ANALYSIS

Analyzing the graph, it is evident that up to 20 threads, the speedup aligns well with the expected values. However, beyond this point, a significant loss in performance is observed. Considering that the cluster used for testing has 20 *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz*, which supports multi-threading, one would expect a continuous increase in speedup. Several factors contribute to this loss:

- **NUMA Architecture:** The CPU has NUMA nodes, and understanding the node distribution is crucial for optimizing parallel code. In our case, NUMA node0 includes CPU(s) 0-9 and 20-29, while NUMA node1 includes CPU(s) 10-19 and 30-39.
- **Thread Overhead:** Beyond a certain point, the overhead introduced by managing a higher number of threads may offset the gains from parallelization.
- **Memory Access Patterns:** The efficiency of parallelization can be affected by memory access patterns, especially in a NUMA architecture. Understanding and optimizing for these patterns is essential.

Further investigations into these aspects could provide insights into optimizing the parallelization strategy for better scalability.

IV. CONCLUSION

In conclusion, in order to parallelize, we focused on optimizing the critical function `computeAccelerationsAndPotential`. Using tools like `perf report` and insights from the previous assignment, we identified and addressed bottlenecks, significantly improving the loop complexity. The introduction of parallelization, however, introduced challenges related to data races, which were partially mitigated using an auxiliary matrix (`aTemp`) and OpenMP reduction directive.

Our expectations, based on Amdahl's Law, provided a theoretical framework for estimating speedup. The expected speedup aligned well with the observed performance of up to 20 threads, after which diminishing returns became apparent. The cluster's architecture, supporting multi-threading with NUMA nodes, raised considerations about thread overhead and memory access patterns beyond a certain thread count.

Performance testing confirmed the anticipated speedup of up to 20 threads, with a subsequent decline. The results suggest a need for a deeper understanding of the NUMA architecture, potential thread management overhead, and memory access patterns to optimize parallelization further. Exploring alternative parallelization strategies and considering affinity to NUMA nodes could potentially enhance scalability.

While our parallelization efforts demonstrated significant improvements in performance, especially in the identified hotspot, further optimization strategies are warranted to address challenges associated with NUMA architecture and threading overhead for sustained scalability.