Computação Paralela

Trabalho Prático 2

Luís Araújo
PG54004
Universidade do Minho
Mestrado em Engenharia Informática

Mateus Pereira
PG54089
Universidade do Minho
Mestrado em Engenharia Informática

***Abstract*—This document explores optimizing a molecular dynamics simulation program using parallel computing techniques on the SeARCH cluster.**

I. Introduction

This report, as part of the *Computação Paralela* course's second practical work, focuses on optimizing a molecular dynamics simulation program by transitioning from the sequential MDseq.cpp made (and modified by us now) in the first assignment to the parallel MDpar.cpp. Our approach centers on identifying computational hot-spots, implementing parallel computing techniques, and evaluating their performance on the SeARCH cluster's compute node.

II. Identification of Hot-Spots

In our project, we first identified key areas for parallelization to enhance program performance. We used the tool *perf* to identify code sections with significant overhead, particularly the *computeAccelerationsAndPot* function, which accounted for 98.87% of the overhead. This function combines Potential and *computeAccelerations*, centralizing most of the program's workload. Based on this analysis, we decided to parallelize this function using OpenMP directives. Additionally, we parallelized the computeAccelerations function, which had a smaller overhead of 0.33%, as it was the only other significant source of overhead in the perf report.

III. Alternatives and Attempts of Parallelizing the Hot-Spots

Our initial attempts into parallelizing the hot-spots with *pragma omp critical* and *atomic* yielded minimal improvements, prompting a strategic shift to restructure the loops. This change significantly reduced the reliance on these synchronization constructs, leading to a more efficient and streamlined code execution. One challenge encountered was slight numerical discrepancies in the *cp_output* file, particularly in its 12th digit, which raised questions about computational precision. We addressed this by rigorously ensuring that the parallelization process did not compromise the simulation's accuracy and that no data races were present in our code.

IV. Implementation and Optimization

A. Parallelization Strategy

The primary step in the optimization process was to transition from the sequential MDseq.cpp implementation to a parallelized approach in MDpar.cpp. This involved integrating parallel computing constructs, specifically targeting the computational hot-spots identified earlier. For that we utilized OpenMP and extensively used its pragmas, more specifically the *#pragma omp parallel for* directive, which was applied to key loops.

B. Load Balancing and Data Management

Dynamic scheduling in OpenMP was adopted for balanced workload distribution among threads, enhancing the code's speed. Loop variables were inherently private due to their in-loop declaration, simplifying the code and reducing data management errors.

In the *computeAccelerationsAndPot()* function, as well as *computeAccelerations()* and *Potential()*, only the outer loop was parallelized, effectively balancing computational efficiency and parallel execution. This strategy avoided the complexities and overheads of nested parallelism, such as excessive thread creation and synchronization. The outer loop, handling distinct pairs i,j, was well-suited for parallel processing, significantly reducing execution time. Conversely, the inner loop was kept sequential due to its coupled computations, avoiding the inefficiencies of parallelizing related calculations.

C. Optimization Techniques

Further optimizations were achieved through additional OpenMP pragmas and reductions. For example, a critical section of the code was optimized using a reduction pragma:

```
#pragma omp parallel for reduction(+:var)
```

This pragma facilitated the safe accumulation of values into the variable across different threads, ensuring data consistency and efficiency in concurrent environments.

D. Challenges and Solutions

Adjusting the code for optimal parallel performance involved overcoming challenges related to efficient thread management and memory use. We had to change how most variables are created by declaring them inside the loops, removing the need for the private pragma statements. We also needed to verify there were no data races by checking if there were variables being edited at the same time by multiple threads. For that we had to change the structure of our code multiple times while testing with atomic and critical statements in key areas. After testing multiple times we removed unnecessary loops in the *computeAccelerations()* function, making the function significantly faster compared to the version from phase1 and taking out of the equation the need for the critical statements.

E. Final Optimization Outcome

When deploying MDpar.cpp on the SeARCH cluster, we observed a slight slowdown with a single thread due to OpenMP overhead but significant improvements with multiple threads. Applying Amdahl's Law, with a parallelization factor of 98.9% (*computeAccelerationsAndPot* overhead's), revealed that while there were substantial gains up to 20 threads, further increases led to diminishing returns due to non-parallelizable code and increased thread management overhead. This highlights the complex trade-off in parallel computing between maximizing parallel execution and managing additional overheads.

V. Performance Measurement and Discussion

TABLE I
Execution Time and Speedup Analysis

# Threads	Time (s)	Speedup	Ideal Speedup	Amdahl's Speedup
1	22.115	0.97	1	1.00
2	11.175	1.93	2	1.978
4	5.665	3.80	4	3.8725
8	2.888	7.45	8	7.428
10	2.376	9.05	10	9.099
12	1.982	10.87	12	10.705
16	1.533	14.03	16	13.734
20	1.265	17.01	20	16.543
24	1.225	17.56	20	-
28	1.187	18.12	20	-
32	1.144	18.82	20	-
36	1.118	19.26	20	-
40	1.089	19.76	20	-
44	1.278	16.83	20	-
48	1.315	16.38	20	-
52	1.339	16.07	20	-
56	1.384	15.56	20	-
60	1.421	15.15	20	-
64	1.471	14.63	20	-
68	1.506	14.29	20	-
72	1.555	13.85	20	-
76	1.576	13.66	20	-
80	1.617	13.30	20	-

VI. Conclusion

The transition from MDseq.cpp to the parallel MDpar.cpp significantly enhanced the molecular dynamics program's efficiency. While substantial speedups were achieved up to 20 threads, performance gains plateaued and even declined after 40 threads, due to the cluster's limited physical and virtual processing units. This illustrates the complex balance in parallel computing between speed gains and the management of thread overheads, highlighting the need for optimized resource utilization in high-performance computing.

A. Attachments

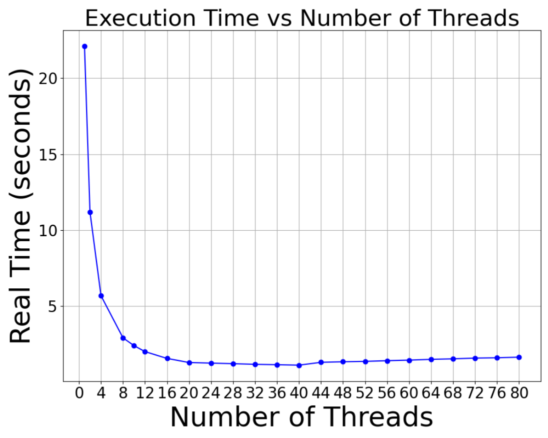

Fig. 1. Execution time for each thread

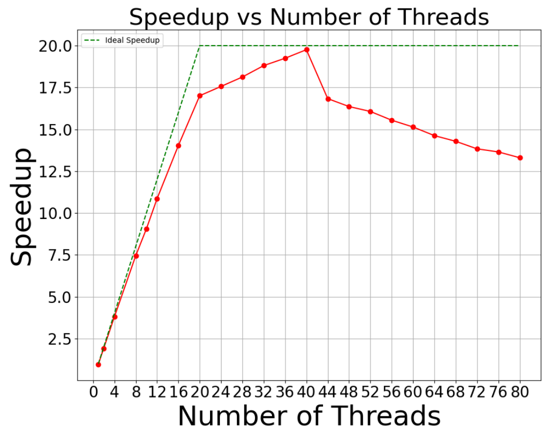

Fig. 2. Speed up

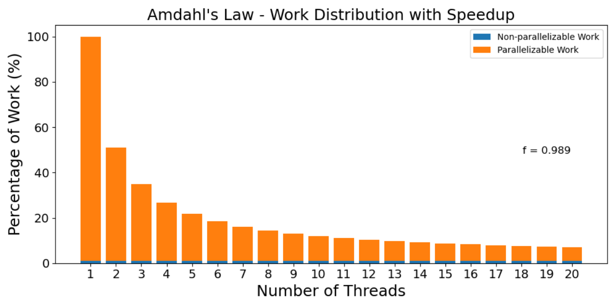

Fig. 3. Amdahl's Law