Parallel Computing - Work Assignment (Phase 2)

José Moreira
Software Engineering
University of Minho
Braga, Portugal
pg53963@alunos.uminho.pt

Santiago Domingues
Software Engineering
University of Minho
Braga, Portugal
pg54225@alunos.uminho.pt

***Abstract*—OpenMP is an API designed for parallel programming on multicore architectures, using the shared memory concept. This API offers programmers extensive capabilities in the world of parallel programming. Consequently, it becomes possible, in a straightforward and practical manner, to improve the execution of a program by breaking it down into various parts and distributing them across multiple threads. This enables the efficient workload of a code segment to be distributed among these threads, leading to a faster presentation of the program's outcomes.**

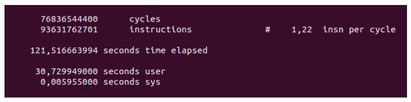
***Index Terms*—multicore, threads, parallel programming, shared memory**

I. Introduction

This document serves as the report for the second phase of the **Parallel Computing** work assignment, that consists in exploring shared memory parallelism with the main goal of improving the execution time of a given program (the same code used in the first phase). This introduction explores the fundamental concepts of **OpenMP**, emphasizing its role in breaking down program complexities, distributing tasks among multiple threads and ultimately improving overall execution efficiency. In this report, we will explain all the steps we took so we could create a faster program without damaging its simplicity (hot-spots, code analysis, adopted strategies, etc.).

II. Code Analysis and Modifications

The first step was to measure the execution time of the program and identify the hot-spots, portions of the code with high computation time. Identifying these blocks of code is an important step, once they have a lot of potential to be optimized. So, we obtained this feedback using the commands **perf stat**, **perf record** and **perf report**.

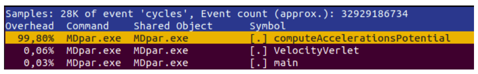


```
     76836544400      cycles
     93631762701      instructions              #    1,22  insn per cycle

   121,516663994 seconds time elapsed

    30,729949000 seconds user
     0,005955000 seconds sys
```

Fig. 1. Code statistics



```
Samples: 28K of event 'cycles', Event count (approx.): 32929186734
Overhead  Command     Shared Object       Symbol
  99,80%  MDpar.exe   MDpar.exe           [.] computeAccelerationsPotential
   0,06%  MDpar.exe   MDpar.exe           [.] VelocityVerlet
   0,03%  MDpar.exe   MDpar.exe           [.] main
```

Fig. 2. Hot-spots

With the presented hot-spots, we inspected the code and realized there were some loops that could be distributed among threads. In first place, we understood very fast that the double loop, in **computeAccelerationsPotential()**, was the heaviest portion of code inside that function. Though, we saw that there was some potential of optimization in the first loop of that function, where all the accelerations were being set to zero. With this first analysis, we studied several **OpenMP** options of optimization that could lead to a good improvement in the execution time of the program. As an initial and simple step, we added a **#pragma omp parallel for** in the first loop of the function. This allowed us to make a small optimization of the execution time of the code with a simple modification. In a second approach, we implemented a **#pragma omp parallel for**, in the double loop, where all the used variables were **private**, except the **Pot** and the matrix of accelerations, **a**. With the variable **Pot**, once it was being increased in every iterations of the loop, we decided to apply a **reduction(+)**. After this decision, we had to control the access to the matrix of accelerations, once it was being updated by all the threads, without any control, an action that could lead to wrong results. In a first (and not right, in terms of execution time) decision, we decided to add **#pragma omp critical** instructions before the updates of the matrix. With this algorithm, the threads could only update the matrix's values one at a time, resulting in correct results. Although, this resulted in a very high computation time, because the threads had to wait a lot of time for one another, so they could access the critical zone, modify the value and exit the same zone. So, we had to think on another solution that could make possible the action of controlling the access to the matrix, where all the threads didn't have to wait for one another while they were, after making their calculations, updating the matrix's values with those results. With a lot of tests, we decided to add the matrix **a** to the **reduction** section, once it was, as well, being increased with the results of some previous calculations. This action resulted in a really good execution time and correct results. After this step, we looked for some instructions that could improve the execution time. With a lot of testing, we decided to implement a **schedule(dynamic, 40)** instruction, giving the property of splitting the code iterations in a dynamic way between threads.

III. Scalability Analysis

In order to ensure the effectiveness of the code modifications, an extensive testing phase was conducted to assess the scalability of the optimized code. The primary objective was to enhance the execution time across varying thread counts. Testing was performed for all program phases, employing one, two, four, eight, twenty, and forty threads to cover a broad spectrum of scenarios.

A. *Consistency Across Thread Counts*

- The testing process aimed to identify a configuration that demonstrated improved execution time consistently, irrespective of the number of threads in use.
- Results were scrutinized to ensure that the code modifications not only optimized performance under specific thread counts but exhibited scalability across a range of parallelization levels.
- Each phase of the program was individually tested with different thread counts to ensure that the scalability improvements were not phase-specific.

B. *Comparison Between MDpar.exe and MDseq.exe*

- Performance results from the modified code in the **MDpar.cpp** file were systematically compared with the original sequential version (**MDseq.cpp**).
- This comparative analysis aimed to validate that the parallelized code not only improved execution times but outperformed the sequential version across diverse scenarios.

IV. Results

In this section of the code, we will present the results of the optimizations. We used a script that executed the code five times with each number of threads: two, four, eight, twenty and forty. This made possible the analysis of the execution time and respective speedup (in comparison to the sequential execution). All these results were rounded to three decimal places and the values in the **#I** and **Cycles** columns of the table correspond to numbers in scientific notation (1E10). Is important to refer that the group made tests with fifty threads, but the results were worse in comparison to the forty threads test.

TABLE I
Results

Threads	Time (s)	#I	Cycles	Speedup (x)
2	14,026 +- 0,344	9,605	6,976	8,664
4	7,061 +- 0,169	9,617	7,005	17,210
8	3,603 +- 0,085	9,650	7,083	33,727
20	1,569 +- 0,034	9,840	7,557	77,400
40	1,458 +- 0,012	10,152	13,772	83,345
Sequential	121.517 +- 4.200	9.363	7.684	-

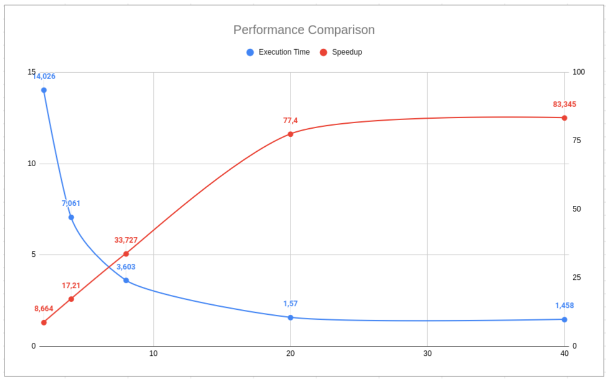


Fig. 3. Performance comparison

V. Conclusions

In conclusion, the exploration of shared memory parallelism, using **OpenMP**, has proven to be a valuable strategy for improving the execution time of the provided program. The identification of hot-spots through performance analysis tools, such as **perf**, allowed us to target specific areas of the code for optimization. The focus was primarily on the **computeAccelerationsPotential()** function, where careful analysis and strategic **OpenMP** directives were applied. The adoption of **OpenMP** parallelization strategies, including **#pragma omp parallel for**, allowed for the efficient distribution of computational workloads among multiple threads. The use of a **reduction(+)** clause and the inclusion of the matrix **a** in the reduction section contributed to both correct results and improved execution times. The dynamic scheduling with **schedule(dynamic, 40)** further optimized the distribution of code iterations among threads. The scalability analysis demonstrated that the code modifications consistently improved execution times across a range of thread counts, from one to forty threads. This scalability was essential to ensure that the optimized code could adapt to varying computational resources. Additionally, the comparison between **MDpar.exe** and **MDseq.exe** confirmed the superior performance of the parallelized version, reinforcing the effectiveness of the implemented optimizations. In summary, the successful application of OpenMP directives and careful consideration of code optimizations resulted in a faster and more scalable program without compromising its simplicity. The iterative testing and refinement process, coupled with thorough performance analysis, contributed to a well-balanced and efficient parallel implementation. This work showcases the potential of shared memory parallelism in enhancing the performance of computational programs.

References

[1] Sergey Dubovyk, "Introduction to the OpenMP with C++ and some integrals approximation" Medium, Accessed on November 21, 2023.
[2] "An introduction to OpenMP" University College London, Accessed on November 21, 2023.
[3] "Introduction to Parallel Programming with OpenMP in C++" GeeksforGeeks, Accessed on November 23, 2023.