Work Assignment
Phase 2

João Pedro Dias Faria
Dept. de informática - UC de Computação Paralela
Universidade do Minho
Braga, Portugal
pg53939

Rui Pedro Guise da Silva
Dept. de informática - UC de Computação Paralela
Universidade do Minho
Braga, Portugal
pg54213

***Abstract*—his document serves as an auxiliary support for the Parallel Computing course project, which corresponds to the academic year 2022/2023.**

***Index Terms*—C, SeARCH, OpenMP, Parallelization.**

I. Introduction

The objective of this phase was to enhance the code improved in the first stage of the project, applying parallelization techniques to improve the execution time.

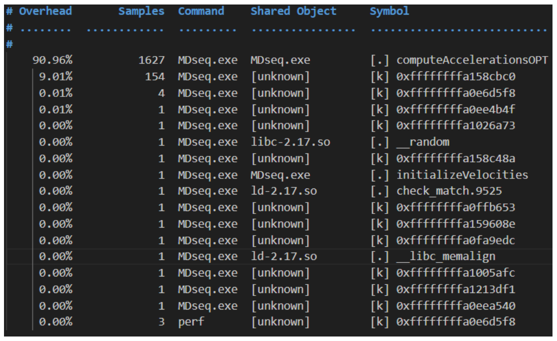
With this objective in mind, we will begin by checking the execution time of the sequential code for N=5000 and identify which function is the most computationally demanding. This initial analysis was crucial to identifying the most opportune sections for optimization.

Following this assessment, our attention shifted towards a detailed examination of the function's code to determine potential areas amenable to parallelization. Through a meticulous evaluation, we aimed to isolate segments that could benefit from concurrent execution, thereby expediting overall processing.

II. Key Blocks

In this second phase, we will assess the algorithm's performance primarily based on its execution time, as it provides a crucial indicator of the efficiency and effective management of CPU resources.

To identify the function that takes up the most execution time, we used *perf record* on the sequential code from the first phase.



```
# Overhead       Samples  Command    Shared Object      Symbol
# ........  ............  .........  .................  ...........................
#
    90.96%          1627  MDseq.exe  MDseq.exe          [.] computeAccelerationsOPT
     9.01%           154  MDseq.exe  [unknown]          [k] 0xffffffffa158cbc0
     0.01%             4  MDseq.exe  [unknown]          [k] 0xffffffffa0e6d5f8
     0.01%             1  MDseq.exe  [unknown]          [k] 0xffffffffa0ee4b4f
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa1026a73
     0.00%             1  MDseq.exe  libc-2.17.so       [.] __random
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa158c48a
     0.00%             1  MDseq.exe  MDseq.exe          [.] initializeVelocities
     0.00%             1  MDseq.exe  ld-2.17.so         [.] check_match.9525
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa0ffb653
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa159608e
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa0fa9edc
     0.00%             1  MDseq.exe  ld-2.17.so         [.] __libc_memalign
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa1005afc
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa1213df1
     0.00%             1  MDseq.exe  [unknown]          [k] 0xffffffffa0eea540
     0.00%             3  perf       [unknown]          [k] 0xffffffffa0e6d5f8
```

Fig. 1. Output obtained using *perf record*.

As anticipated, the most time-consuming function is 'computeAccelerationsOPT,' and it is this function that we will evaluate for the potential of parallelization.

III. Version number one

In addition to avoid potential concurrency issues related to variable sharing between threads, our approach to optimizing the code involves a careful consideration of variables within the loop. We have confined the declaration of variables used exclusively within the loop, thereby reducing the number of variables that require careful management to only two: 'Pot' and matrix 'a'. By restricting variable declarations within the loop, we reduce the overhead associated with private variables, avoiding the need for explicit use of OpenMP's private clause. This enables a more streamlined execution process. This approach not only simplifies the code but also leverages the inherent optimizations that compilers can apply when variables have a well-defined scope.

Recognizing that 'Pot' is a global variable, we took proactive steps to prevent thread conflicts by introducing an auxiliary variable 'P'. This auxiliary variable seamlessly replaces 'Pot' within the for loop, and upon completion of the 'Pot' function, it assumes the value of the auxiliary variable 'P'. This strategic substitution mitigates conflicts associated with global variables, ensuring smoother execution.

With these foundational adjustments implemented, our attention turned to the remaining variables: the newly introduced variable 'P' and the matrix 'a'. It became evident that both variables are written to by all threads, raising concerns about potential data races.

To fortify the program against such issues and ensure both efficiency and correctness, we leveraged the 'reduction' clause in OpenMP. Applying the + operator in conjunction with this clause proved instrumental. By utilizing the 'reduction' clause, each thread maintains a private copy of the reduction variable ('P' or 'a'). At the conclusion of the parallel region, these private copies harmoniously combine to yield the final result, effectively sidestepping data race complications.

IV. Version number two

For version two, we corrected an error made in phase 1 and reintroduced the 'computeAccelerations' function to be utilized in the situation where we were redundantly calculating P.

Furthermore, to further parallelize the program, we incorporated the *schedule(dynamic, N)* OpenMP directive to regulate the distribution of loop iterations among threads dynamically, each handling a chunk of N iterations. This decision was based on the varying workload of loop iterations, aiming to ensure load balancing and responsiveness across, given that the number of particles may vary in future use cases.By dynamically allocating tasks, the directive adapts to unpredictable workloads, potentially reducing idle time among threads and enhancing overall efficiency.

While contemplating parallelization, we explored the option of parallelizing the loop responsible for initializing the matrix ’a’ with zeros. However, our testing phase revealed that this approach failed to yield improvements in execution time, leading us to abandon this particular optimization strategy.

It’s noteworthy that the determination of the number of threads to be utilized in the parallelized sections of our code is dynamic and defined during program execution. This adaptability is facilitated by the command: export OMP_NUM_THREADS=N, where ’N’ represents the desired number of threads.

V. Tests

Considering that execution time is the most suitable metric for evaluating the efficiency and effectiveness of CPU resource utilization, it will be the main metric considered in this phase.

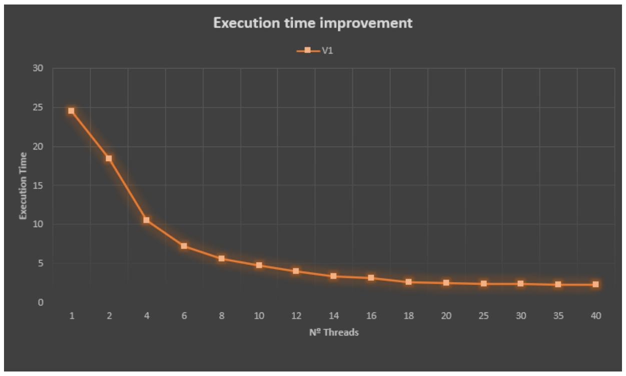


Fig. 2. Version Number One

The final execution time considered for each thread is obtained through the average of three execution times (see appendix A).

As we can observe, the code from version 1 reaches the minimum execution time with 40 threads. After testing different chunk sizes for our case (see appendix B), we compare these of the first version with those obtained in version 2 to determine which version we would choose.

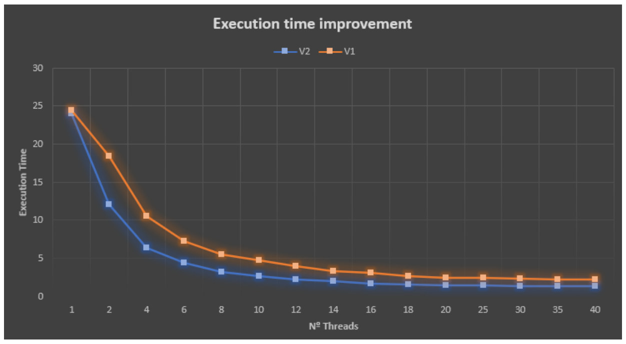


Fig. 3. Version Number Two

As we can see from the graph, the execution time obtained with this new version is slightly faster than the first one tested.

In addition to the execution time, we also examine the speedup concerning the number of threads,using the second version. To provide a basis for comparison with the results obtained, we opted to include the ideal speedup graph. Ideal speedup grows linearly up to 20 threads, as the *SeARCH* has 20 physical cores.

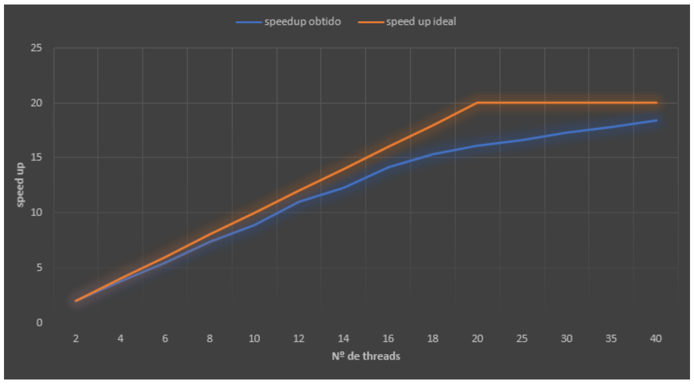


Fig. 4. speedup

As expected, the achieved speedup does not match the ideal speedup, mainly due to incomplete parallelization of the code. The disparity in speedup can also be attributed to the overhead by parallelization, due to the creation and synchronization of threads, and the competition for memory access among the threads.

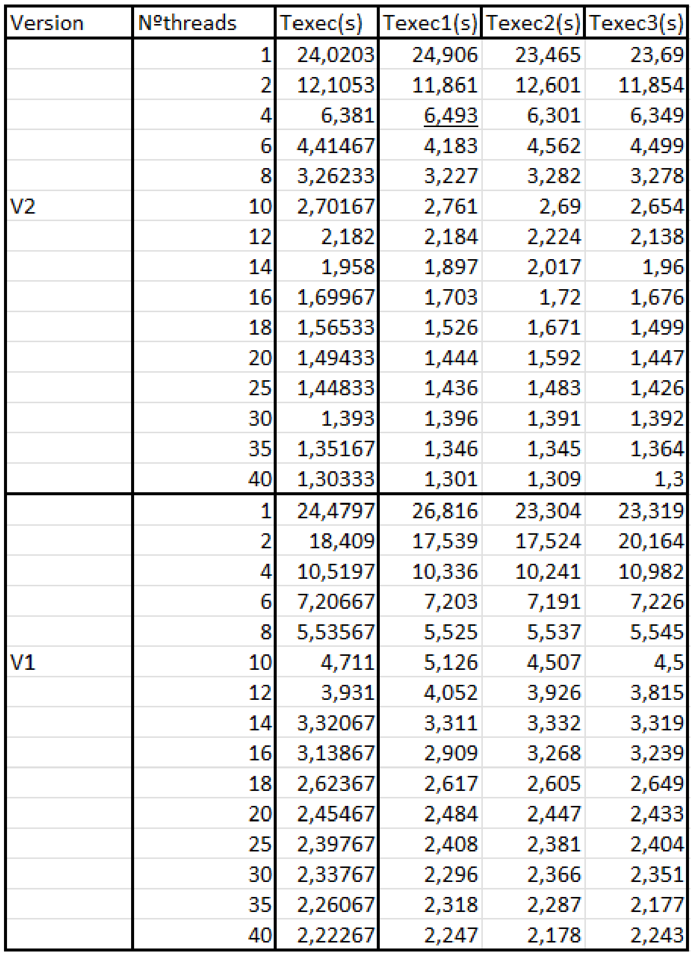
VI. Conclusion

Through this second phase of the project, our team has significantly strengthened our knowledge on the theoretical and practical components of parallel programming within a shared memory environment using C and OpenMP.

Our team is very satisfied with the the results achieved on this project. By effectively harnessing parallelization techniques, we have reduced the sequential execution time of the code from the initial phase. This achievement was made possible by distributing computational workloads across available resources, resulting in significantly improved performance and execution times.

Appendix

A. Appendix A



Version	Nºthreads	Texec(s)	Texec1(s)	Texec2(s)	Texec3(s)
	1	24,0203	24,906	23,465	23,69
	2	12,1053	11,861	12,601	11,854
	4	6,381	6,493	6,301	6,349
	6	4,41467	4,183	4,562	4,499
	8	3,26233	3,227	3,282	3,278
V2	10	2,70167	2,761	2,69	2,654
	12	2,182	2,184	2,224	2,138
	14	1,958	1,897	2,017	1,96
	16	1,69967	1,703	1,72	1,676
	18	1,56533	1,526	1,671	1,499
	20	1,49433	1,444	1,592	1,447
	25	1,44833	1,436	1,483	1,426
	30	1,393	1,396	1,391	1,392
	35	1,35167	1,346	1,345	1,364
	40	1,30333	1,301	1,309	1,3
	1	24,4797	26,816	23,304	23,319
	2	18,409	17,539	17,524	20,164
	4	10,5197	10,336	10,241	10,982
	6	7,20667	7,203	7,191	7,226
	8	5,53567	5,525	5,537	5,545
V1	10	4,711	5,126	4,507	4,5
	12	3,931	4,052	3,926	3,815
	14	3,32067	3,311	3,332	3,319
	16	3,13867	2,909	3,268	3,239
	18	2,62367	2,617	2,605	2,649
	20	2,45467	2,484	2,447	2,433
	25	2,39767	2,408	2,381	2,404
	30	2,33767	2,296	2,366	2,351
	35	2,26067	2,318	2,287	2,177
	40	2,22267	2,247	2,178	2,243

Fig. 5. execution times

B. Appendix B

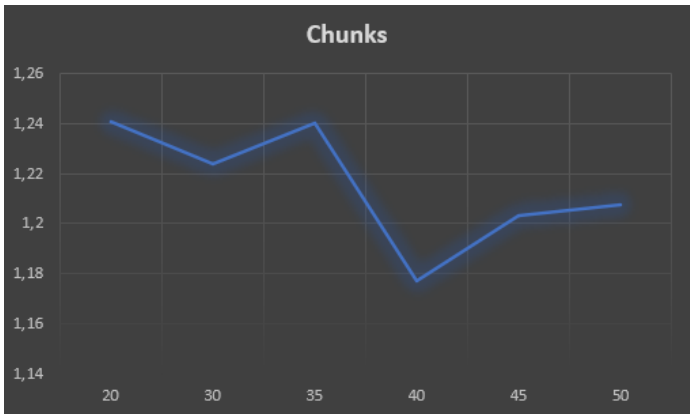


Fig. 6. execution time per chunks