Parallel Computing
Work Assignment Phase 2

Ana Rita Poças (*pg53645)* e Orlando Palmeira (*pg54123)*

***Abstract*—At this phase of the practical assignment, we will use the optimized code from the first phase (simple molecular dynamics simulation). The main goal is to explore shared memory parallelism techniques, using the OpenMP tool, in order to reduce the code's execution time.**

***Index Terms*—Shared memory parallelism, threads, OpenMP, execution time**

I. Introduction

The main goal of this phase of the practical assignment is to apply parallelism techniques to a single-threaded program in order to reduce its overall execution time. To achieve this goal, we used profiling tools to detect hot spots (in terms of computing), then we used these results to know where we should apply parallelism techniques and, in the end, measured the performance of the adopted solution through scalability analysis.

II. Changes regarding the first phase

In this phase of the practical assignment, the input size has been changed. In the first phase, we had $N = 2160$, and now $N = 5000$. This change resulted in a deterioration in the overall execution time (as expected) of the sequential code, making it approximately 23 seconds.

III. Code analysis

A. *Hotspots detected in our code*

Just like in the first phase, we used the *gprof* tool to detect hotspots consuming CPU time. Thus, we generated the following graph produced by *gprof2dot*:

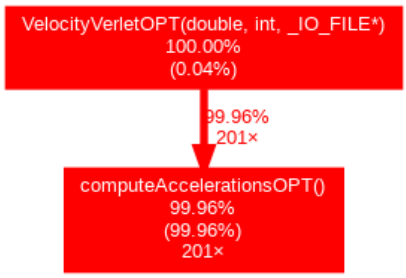

Fig. 1. Hotspot detected with *gprof*

As we can see, the function *computeAccelerations* is the heaviest function, occupying nearly 100% of the execution time. Therefore, this function will be the main focus for applying parallelism techniques.

In the *computeAccelerations* function, there are two nested *for* loops where all the mathematical calculations of this function are performed, with the rest being simple operations for initializing variable values. Therefore, it will be in these *for* loops that we will apply parallelism techniques.

IV. Implementation of parallelism techniques

As mentioned earlier, we will apply parallelism techniques to the two nested *for* loops of the *computeAccelerations* function, and this application will take place in the outer loop. We have chosen to implement parallelism in the outer loop in order to reduce the granularity of tasks executed by each thread, which may result in performance improvement (small tasks introduce more(relative) overhead caused by creation and management of threads).

A. *Parallelism exploration and approaches chosen*

Because of the reasons mentioned earlier, we will use the `#pragma omp parallel for` directive to parallelize the loop. However, there are variables in the loop whose usage should be exclusive to each thread (since their values won't be used outside the context of the parallelized section of the function). To address this, we have used "private" within the directive to declare variables that need to be copied for each thread to avoid data races.

While most variables in the function are protected using "private", two of them cannot be treated this way (Pot and matrix 'a'), as their values are used outside the context of the parallelized loop. Thus, we can employ some techniques to solve this issue. The first option is to safeguard calculations associated with these variables with directives such as "#pragma critical" or "#pragma atomic". This approach introduces some overhead due to the need for thread synchronization and, furthermore, contributes to reducing the portion of parallelized code, making it a less favorable option.

The second option is to use "reduction", where a copy of the 'Pot' and 'a' variables is made for each thread, and the values of these copies are aggregated into the original variables at the end of the loop. This technique is evidently better as it avoids the need for heavier synchronization mechanisms like "critical" and "atomic", contributing to improved performance.

Taking all the previous explanation into account, the OpenMP directive used in the outer loop will be the following: `#pragma omp parallel for reduction(Pot,a[:N][:3]) private(variables of the loop)`. However, there are still other options that

we can explore, which involve choosing the task scheduling approach for the threads.

For the scheduling of thread tasks, we decided to explore static scheduling and dynamic scheduling. Static scheduling has the advantage of not introducing much overhead in thread management, as the blocks of iterations assigned to each thread are predefined and do not change over time. On the other hand, dynamic scheduling allows load balancing between threads, enabling a thread that finishes its block of iterations to request more ”work” to reduce the load on threads performing heavier tasks. However, dynamic scheduling introduces overhead in thread management.

With this, we anticipate that the use of dynamic scheduling can contribute to better code performance. The reason for this is that the number of iterations in the inner loop (executed by each thread) directly depends on the value of the i variable from the outer loop, which may result in unbalanced thread loads. For example, with N=5000 and i=0, the inner loop will have 4999 iterations. If i=4500, then the inner loop will have 499 iterations. There is an evident imbalance in the load distribution, and this can be minimized with the use of dynamic scheduling.

B. *Scalability analysis*

To confirm our predictions, we conducted a comparison of the two approaches through a scalability analysis:

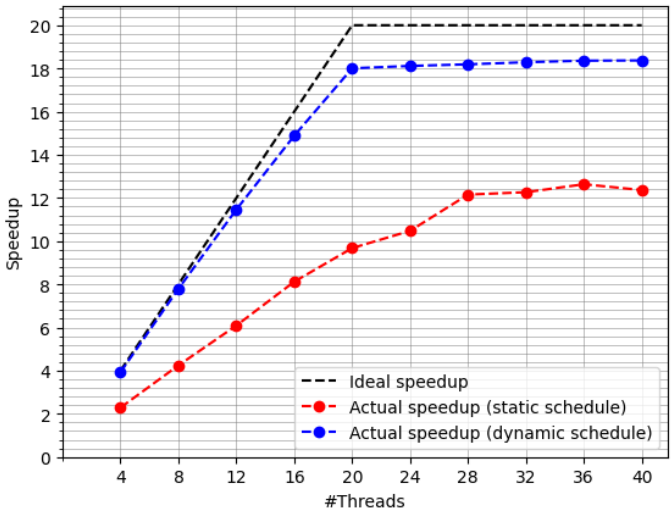

Fig. 2. Comparison dynamic vs static scheduling

As shown in the graph above, the version with dynamic scheduling performs significantly better (it’s very close to the ideal speedup). Due to the uneven load that would exist between threads with the static scheduling approach, the overhead imposed by thread management in dynamic scheduling is easily offset by the load balancing advantages.

V. Performance analysis

From the graph presented in Figure 2, we can observe that our solution failed to achieve the ideal speedup. It was expected that the solution would not reach optimal performance, and there are some reasons for this to occur.

A. *Reduction overhead*

Firstly, we must consider the overhead imposed by the use of the ‘reduction‘ clause, particularly on the matrix ‘a‘. At the creation of threads, we are making multiple copies of the matrix ’a’ (with dimensions 5000×3). At the end of the parallelized loop execution, the sum of all elements of the matrix is calculated to obtain the result. All these operations impose additional load on the execution.

B. *Memory wall*

In terms of memory bandwidth, we believe this factor has some impact on the performance of our solution. As seen in Appendix A, the increase in the number of threads contributes to a slight increase in CPI. This is a possible indicator (without absolute certainty) that the algorithm might be somewhat constrained by memory bandwidth, since the CPU might be waiting for data from the memory (there are multiple memory accesses, mainly in matrix ’a’). Furthermore, we analyzed the arithmetic intensity (Appendix B, evolution of #I/LLC.Miss with the number of threads) and we’ve realized that the arithmetic intensity decreases when the number of threads increases. This might tell us that the algorithm of our program is memory bounded, which contributes to deteriorate the performance.

C. *Load imbalance*

When it comes to load imbalance, we acknowledge the possibility that our solution may be affected by this factor. In fact, we attempted to minimize this issue with the use of ’schedule(dynamic)’, but this mechanism may not be perfect. Additionally, dynamic scheduling imposes thread management overhead, which contributes to not reach the ideal speedup.

D. *Task granularity*

We believe that the solution’s performance is not affected by task granularity, as the parallelism approach adopted in our solution already has the maximum task size possible to achieve in the optimized function.

E. *Synchronization overhead*

Finally, we also believe that performance may not be affected by synchronization overhead since no synchronization directives like ’atomic’ and ’critical’ are used anywhere in the code.

VI. Conclusion

At the end of this phase of the practical work, we have managed to develop a program faster than its sequential version, thanks to the implementation of parallelism. However, the adopted solution did not reach the ideal performance, although we believe it came quite close.

One aspect we would like to highlight is the fact that the adopted approach proved to be quite effective, achieving a very satisfactory performance, relatively close to the ideal, with the use of only one OpenMP directive, without the need for synchronization directives.

Finally, it is worth mentioning that, after various tests, we determined that the optimal number of threads to achieve the best performance in our solution is 40.

APPENDIX A
CPI EVOLUTION

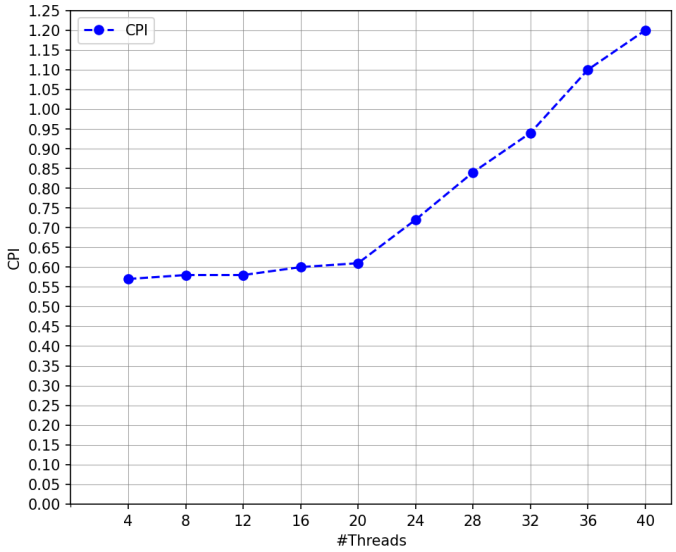

Fig. 3. CPI evolution with the number of threads

APPENDIX B
ARITHMETIC INTENSITY (#I/LLC.MISS)

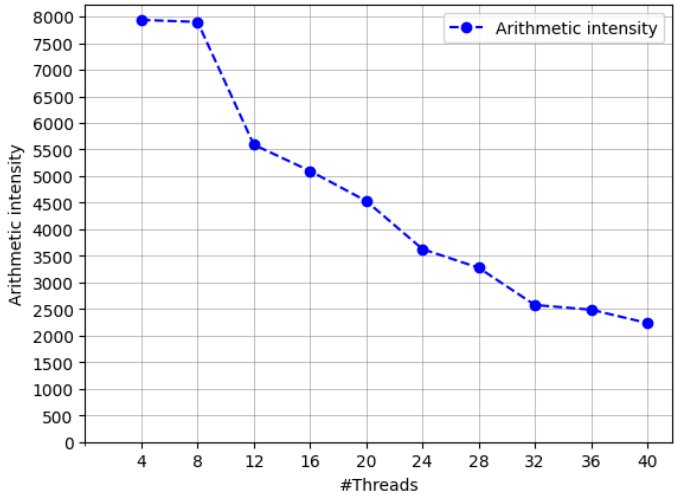

Fig. 4. Arithmetic intensity with the number of threads