Phase 2: Parallel Programming Project Report

Diogo Gomes Rodrigues
Physical Engineering Student
Minho's University
Braga, Portugal
a103847@alunos.uminho.pt

Gonçalo Filipe Ribeiro Rodrigues
Physical Engineering Student
Minho's University
Braga, Portugal
a103849@alunos.uminho.pt

Vitor Fernandes Alves
Physical Engineering Student
Minho's University
Braga, Portugal
a103940@alunos.uminho.pt

I. Introduction

This report focus on optimizing and parallelizing the code identifying bottlenecks, restructuring the algorithm, and using OpenMP constructs.

II. Code Optimizations

We began by reordering the loops in the new lin_solve function to k, j, i and adjusted the summation order, replacing division with multiplication. We also enabled compiler flags: -O3 for optimization, -funroll-loops for loop unrolling, -ftree-vectorize and -mavx for vectorization, and -march=ivybridge for architecture-specific optimization.

III. Parallelization of the Code

a. *Identification of Bottlenecks*

Using gprof, we identified that the functions consuming the most time were lin_solve with 46.77% and advect with 27.89%. We then proceeded to parallelize these functions.

b. *Parallelization of the function lin_solve (par_l)*

The lin_solve function contains two sets of nested loops. A parallel region was created for both set with a barrier between them. While x array updates are independent, reduction(max:max_c) was used in max_c to ensure correctness , and old_x and change were declared private to avoid conflicts. The outermost loop (k) in each set was parallelized using #pragma omp for schedule(static) collapse(2) nowait, enabling efficient workload distribution by combining two outer loops into larger chunks and minimizing thread synchronization.

The results of the performance improvement achieved with this parallelization are presented in Table I.

c. *Parallelization of the function advect (par_l_a)*

The advect function contains three nested for loops with independent modifications to the d array across iterations. Parallelization was achieved using with #pragma omp and #pragma omp for schedule(static) collapse(3) nowait to efficiently distribute the workload. Additionally, _mm_prefetch was used to preload data from the d0 array.

The results of this parallelization are show in Table I. Gprof revealed that project takes 44.34% of execution time.

d. *Parallelization of the function project (par_l_a_p)*

This function includes two sets of three nested loops. To optimize, we reordered the loops to k, j, i for better spatial locality and precomputed MAX(M, MAX(N, O)). For parallelization, we created two parallel regions, one for each set of loops. The first region updates div and p, while the second handles u, v, and w, with independence between iterations. Both regions use #pragma omp parallel for schedule(static) to efficiently distribute work across threads.

The parallelization results are shown in Table I. Gprof revealed that the set_bnd with 25.23% is a new bottlenecks.

e. *Parallelization of the function set_bnd (par_l_a_p_b)*

In this function, we began by reordering the loops to improve spatial locality. Then, we created a parallel region to cover all three sets of nested loops. The positions of the array x that are modified are distinct, preventing data races. To distribute the workload across threads, we used #pragma omp for schedule(static), ensuring efficient load balancing with static scheduling. The results are presented in Table I.

f. *Parallelization of the function vel_step*

As mentioned in the earlier phase, while this function can be parallelized, doing so would involve parallelizing calls to already parallelized functions, worsening performance. Therefore, we chose not to parallelize it.

g. *Timing Measurement Results*

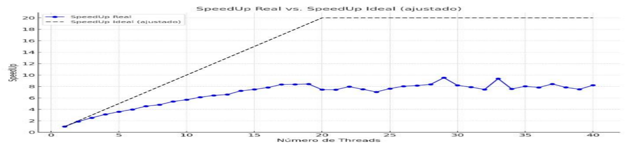
TABLE I. The results of the time measurements were obtained using the k-best measurement heuristic to ensure replicability, with k=5 and 20 measurements, using 18 threads.

	T(s)		T(s)
original	25.7±0.2	par_l_a_p	4.2±0.1
par_l	23.1±0.1	par_l_a_p_b	2.9±0.1
par_l_a	8.4±0.1	seq	21.7±0.1

Analyzing the table, we see that parallelization reduced execution time, but the final speedup of 7.5 falls short of the expected 18 (with 18 threads and cores). This is due to remaining sequential code and the lin_solve function, which, despite being parallelized, remains time-consuming due to extensive float operations.

IV. Performance Speedup Graph

In order to analyze the ideal number of threads, we made a graph comparing the speedup time compared to the already optimized sequential version.



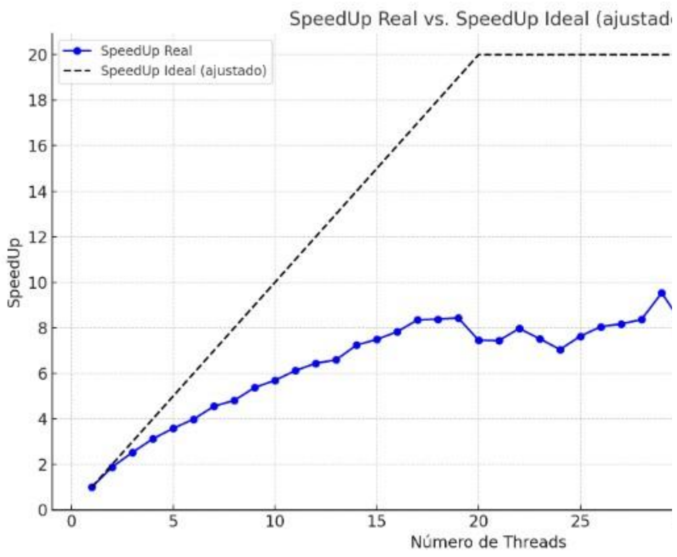
Graph I- Scalability Graph comparing the SpeedUp with the number of threads used.

We can clearly see that speedup is pratically linear up to 19 threads, wich is to be expected since the cluster has 20 cores, and from 20 threads there is an inconsistent increase and decrease in execution time, reaching a variable peak around 29 threads, wich can be explained by the processor´s support for Hyper-Threading. The real speedup is much lower than the theoretical one, wich may be due to the fact that there is an overhead in thread management and frequent context switching between threads, wich is quite costly when several threads compete for the same cores.

References

[1] Slides of theoritical parallel programming classes

ANNEXES



Graph I- Scalability Graph comparing the SpeedUp with the number of threads used