Parallel Computing
Work Assignment Phase 2

Gonçalo Costa
PG55944
Universidade do Minho
Braga, Portugal

Marta Rodrigues
PG55982
Universidade do Minho
Braga, Portugal

Pedro Lopes
PG55992
Universidade do Minho
Braga, Portugal

***Abstract*—The program simulates fluid dynamics in 3D using Jos Stam's stable fluid solver. In this paper, we analyzed and implemented a parallelized version of this algorithm using OpenMP.**

***Index terms*—Parallelism, OpenMP, 3D Fluid Solver**

I. Introduction

This phase aimed to explore shared memory parallelism with OpenMP directives to minimize the execution time.

II. Code Profiling

We increased the data size from SIZE = 42 to SIZE = 84 and replaced the *lin_solve* function with a more efficient red-black solver, modifying it to align as closely with the approach used in the first phase. Using the *perf* command, we generated a *report* and concluded that the hotspots were the following, with the associated overhead:

Function	Overhead
lin_solve	68,87%
project	16,79%
advect	13,09%

Table I: Results of the command perf

III. Implementations

We analysed and optimized the identified functions abo using *#pragma* directives to reduce overhead and improve performance:

- ***lin_solve***: Parallelised the red and black loops found in this solver and, as both loops are nested, we decided to use the *collapse(2)* directive. We used 2 instead of 3, because the inner loop is dependent on the outer loops, violating an important premise of this directive: the independence between the loops, to linearize them.

  To ensure there were no data races with *max_c*, we applied the *reduction(max:max_c)* directive, which creates private copies of this value for each thread, to prevent any risk of threads reading wrong values.

  Similarly, *old_x*, *IDX*, and *change* were declared inside loops to ensure thread-specific access, which could also be achieved using the *private* directive.

  We also used the directive *schedule(static)* to balance the workload of the linearized iterations across the threads available.
- ***advect***: We only used the for *collapse(3) schedule(static)* directive to parallelize the function. This time, we figured that value 3 was the best by testing different values and checking that all loops were independent.
- ***project***: We only parallelized the loops through the same strategy mentioned in the previous *advect* function, leaving the function calls outside the loops sequential because *lin_solve* was already parallelized and *set_bnd* was problematic, so we ended up leaving this last function as is.
- ***add_source***: Though less frequently called, we used *parallel for schedule(static)* for maximum parallelization, even if it didn't improve the performance as much as the other functions.

Other functions remained unchanged due to parallelization issues (e.g., *set_bnd*) or insignificant performance improvements.

IV. Performance (Strong Scalabity) Analysis

We evaluated the performance of the code across multiple threads and calculated the speed-up, as shown in the Appendix. We also compared it against the ideal speed-up using Amdahl's Law, resulting in the graph provided.

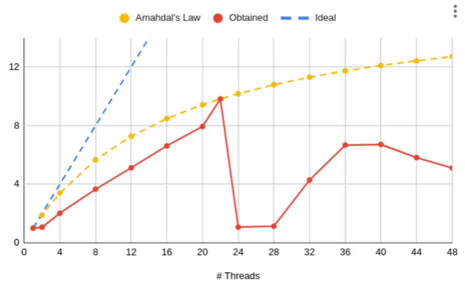

Fig. 1: Speed-Up Graph using *srun* with *cpus-per-task* = *threads*

Our implementation achieved a faster version than the sequential code, with a mostly linear speed-up. However, the parallelization fell short of the ideal speed-up for a fully parallelizable algorithm. Notably, there was a sharp drop in performance between 24 and 28 threads before returning to normal behaviour, which we found unusual. However, running the script *run.sh* through every thread with 48 CPUs per task produced a better result.

V. Appendix

A. *Results for Parallelization Scalability*

#T	TExec	SpeedUp	#I$\times10^9$	#CC$\times10^9$	#M$\times10^6$	CPI
1	14.4301	1	81.99	45.49	1.89	0.6
2	13.5818	1.06	82.33	83.65	1.92	1.0
4	7.1313	2.02	82.73	85.16	1.92	1.0
8	3.9483	3.65	83.21	88.23	1.92	1.1
16	2.1808	6,62	84.65	95.59	1.95	1.1
20	1.818	7,94	85.46	99.26	1.99	1.2
22	1.6776	9,82	85.72	100.64	1.99	1.2
32	3.369	4,28	133.83	285.18	2.10	2.1
48	2.8302	5,1	115.69	370.76	2.11	3.2

Table II: Results obtained using different number of threads

- #T - Number of threads
- TExec - Execution Time
- #I - Number of instructions $\times$ 10^9
- #CC - Number of clock cycles $\times$ 10^9
- #M - Number of Misses $\times$ 10^7
- CPI - Cycles per Instruction

B. *Amahdal's Law*

The speed-up for P processes (S(P)) is calculated using Amdahl's Law:

$$S(p) = \frac{1}{f + \frac{1-f}{P}}$$

Where:

- P - Number of processes (threads)
- f - serial fraction of work
- S(p) - Speed-Up for p processes

To determine f, we used the formula:

$$f = \frac{\frac{1}{S} - \frac{1}{P}}{1 - \left(\frac{1}{P}\right)}$$

Here, $S = \dfrac{Tseq}{Tpar}$, with:

$Tseq$ = execution time for the sequential program

$Tpar$ = execution time for the parallelized program with P=22 threads, chosen because it showed optimal performance with minimal overhead.

This gave f=0.0591.

With these calculations, we obtained the ideal speed-up based on Amahdal's Law for the different number of threads:

#T	1	2	4	8	16	20	22	32	48
SpeedUp	1	1,89	3,41	5,71	8,6	9,58	9,99	11,53	13

Table III: Ideal Speed-Up for different number of threads

C. *Speed-up Graph using script*

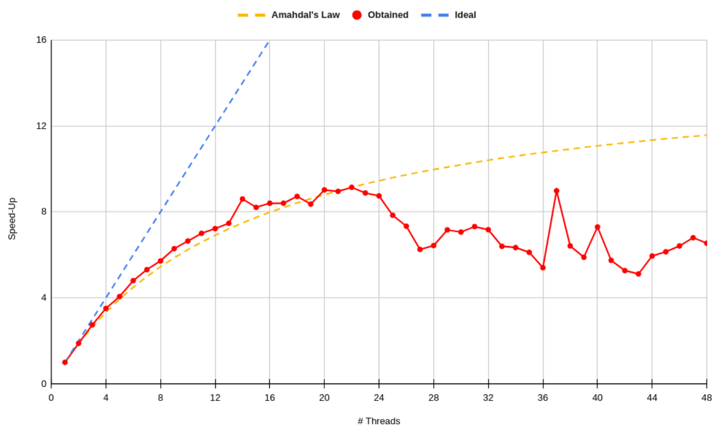

Fig. 2: Speed-Up Graph using the script through 48 threads