Work Assignment — Phase 2

João Luís Ferreira Magalhães
MS Computer Engineering
University of Minho
Braga, Portugal

Jorge Nuno Gomes Rodrigues
MS Computer Engineering
University of Minho
Braga, Portugal

Rodrigo Ferreira Gomes
MS Computer Engineering
University of Minho
Braga, Portugal

***Abstract*—The second phase of the work assignment consists of exploring parallelism using OpenMP on the new version of the solver. In this phase, the program should be parallelized using an optimal number of threads.**

***Index Terms*—Parallelism, OpenMp, Threads**

I. Analysis Of The Solver

For this phase, it was required a change in the size of the grid (84) plus a new solver, more suitable for parallelization. From the previous phase, we chose to keep some optimizations regarding the solver, in which were also changed in the new version. For the first profiling, using the previous level of optimization, we got an execution time of 13.37 seconds, with the *hot_spot* being the function *lin_solve*.

II. Data Races And Parallelization Analysis

The function *lin_solve* implements a Red-Black approach that heavily encourages the parallelization of the code. This approach also handles the shared *x* and *x0* arrays, making sure that the same memory position isn't being written at the same time if each phase is sequential. The function *set_bnd* is outside the phases, which means is a point of synchronization. Finally, the convergence check implemented through *max_c* requires special attention because the changes need to be safely observed by each thread. For the rest of the code, we came to the conclusion that most of it could be indeed parallelized due to the lack of data races. The only requirement was to keep the instructions in sequence and understand what parts of the code should only be executed by one thread.

Taking into account the data races exposed, we can now parallelize the code, leveraging *OpenMP* directives to distribute computations across multiple threads. The distribution occurs at the outermost loop, using *pragma omp parallel for*, so that each thread can process a subset of the grid. We also opted to add the *shared(x, x0)* clause for readability, as the arrays would be shared by the threads anyway. It is important to note that each phase received the same treatment and that we kept the original sequence of the code. To handle the convergence optimally, we opted for the clause *reduction (max:max_c)* . This clause makes each thread keep a copy of the variable *max_c* to later be reduced to only one maximum value. Finally, we must add the clause *private(old_x, change)* for the mandatory local copy required by the algorithm. Alternatively, we could change the declaration to the for loop.

For the functions *add_source* and *advect*, we used the clause *pragma omp parallel for*, respecting each shared variables. For the functions *set_bnd* and *project*, we followed the same logic. However, we must handle the private copy for *i* and *j* in the first case, and we have sequentially between loops that must be preserved for the second case.

III. Scalability Analysis And Discussion

After implementing the parallelization, we ran the program for $1 \leq N \leq 40$, so we could analyze the optimal number of threads. The speedups obtained can be found in Fig 1

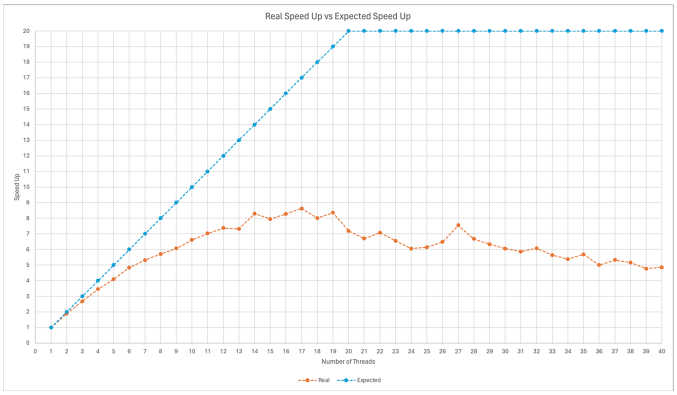

Fig. 1. Speedup of the program for the different number of threads, with the expected ideal speedup for the "cpar" partion.

As seen in the graph, the experimental speedup increases only up to 17 threads, decreasing after that with the increase of threads. The Cache misses were expected to increase with the number of threads. However, in reality, the cache misses were stable. Yet, the memory loads started to increase with the number of threads. What really impacted the execution time was the memory bandwidth that isn't sufficient to deliver the data to every core simultaneously, resulting in a delay on the thread's tasks. This can be easily observed in the reports when *gomp_team_barrier_wait_end* is what takes the most time in an ambient with 25 or more threads. It's important to mention that exists an overhead with routines used, like the *reduction*, even if it's the best option, verifiable by the increase in *CPI*. Finally, we must consider the work balance of each thread can vary, which may be decremental to performance. The data for the analysis can be found in Table I on the *Annexes*.

IV. Annexes

A. Other Data Obtained

TABLE I
PERFORMANCE METRICS BY NUMBER OF THREADS

Threads	$T_{\text{Execution Time}}$	CPI	Memory loads	L1-dcache-loads
1	13.37 s	0.50	0	31,036,051,605
2	7.10 s	0.50	0	31,109,765,135
3	4.99 s	0.50	0	31,216,676,735
4	3.87 s	0.60	0	31,236,658,879
5	3.26 s	0.60	0	31,314,091,465
6	2.77 s	0.60	0	31,133,993,718
7	2.52 s	0.60	0	31,405,856,180
8	2.34 s	0.60	1,483	31,498,863,595
9	2.20 s	0.60	53	31,722,649,670
10	2.02 s	0.60	0	31,719,390,833
11	1.90 s	0.70	0	31,829,716,735
12	1.81 s	0.70	0	31,749,132,688
13	1.83 s	0.70	1,294	31,996,448,207
14	1.61 s	0.70	1,216	31,937,539,835
15	1.68 s	0.80	0	32,042,985,451
16	1.61 s	0.80	0	32,343,212,555
17	1.55 s	0.80	0	32,330,879,189
18	1.67 s	0.90	0	32,377,803,830
19	1.60 s	0.90	279	32,747,018,030
20	1.86 s	1.00	16,583	33,537,125,622
21	2.00 s	1.10	29,129	33,914,882,225
22	1.89 s	1.10	215,127	34,303,545,381
23	2.04 s	1.20	249,779	34,353,370,282
24	2.21 s	1.40	302,383	34,581,431,947
25	2.18 s	1.40	207,044	34,471,252,798
26	2.06 s	1.40	412,112	34,564,771,613
27	1.77 s	1.30	112,917	34,458,149,491
28	2.00 s	1.50	627,363	34,128,070,365
29	2.11 s	1.60	616,128	34,341,498,082
30	2.21 s	1.70	625,951	34,794,249,225
31	2.28 s	1.80	646,231	34,723,560,926
32	2.20 s	1.80	766,961	34,793,075,859
33	2.37 s	2.00	653,783	34,848,826,103
34	2.49 s	2.10	784,301	35,266,469,346
35	2.35 s	2.00	1,761,849	35,369,211,147
36	2.68 s	2.40	1,447,214	35,313,665,275
37	2.51 s	2.30	1,370,538	35,442,166,975
38	2.59 s	2.40	1,854,607	35,129,440,223
39	2.81 s	2.70	1,376,675	35,721,842,820
40	2.75 s	2.60	3,431,356	35,983,817,724

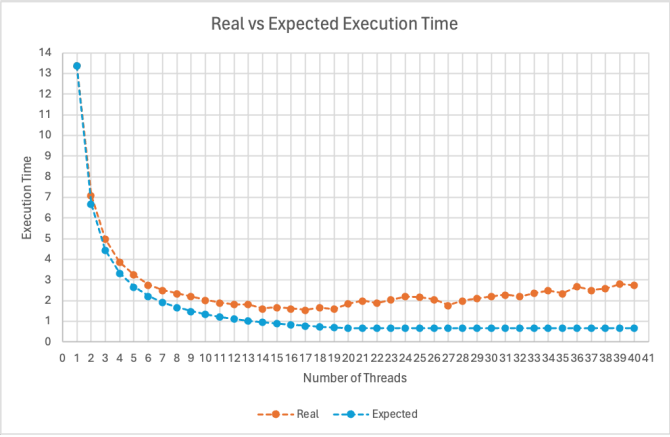

Fig. 2. Execution time of the program for the different number of threads with the expected ideal time.

B. Iterative Analysis Method

The use of an iterative method for the analysis of the code was very much important to the results we obtained. The method can be summarized in this manner: we run the code with *perf report* in a multi thread environment and find the current *hot spot* of the code that requires the most attention. From this setup, we managed to comprehend that many sections of the code could be furthered parallelize. A quick example is the two functions *lin_solve*, the first *hot spot*, and *advect*. After finishing parallelizing *lin_solve*, the report showed us that now *advect* is the function that is taking the most and so on.

C. Exploring other Alternatives

For the optimal parallelization of the function *lin_solve*, we also considered other clauses that we verified not to be optimal. These alternatives use the *critical* clause, which is the least efficient option, and use the *atomic* that was not a success. Starting with the critical clause, we used the same *pragma omp parallel for shared(x, x0) private(old_x,change)* on the outside loop, with the deference now being the use of *pragma omp critical* for the *if* condition to choose the maximum value. This is very much inefficient because promotes a very strong barrier for each thread that must stop on that block and execute sequentially. This version is worse than a non parallelized code. For the *atomic* clause, we could not code a version that ensures the calculation of the maximum value. Our version was producing the proper end result, but by analyzing the code, we could not guarantee it would always work for every run of the code. So, we considered this version to not be suitable for this solver.