18 MAY 2025

HIGH PERFORMANCE COMPUTING

MPI PRACTICE

Oriol Vilalta Arrondo
Joel Lloret Vidal

Escola Politècnica Superior
Universitat de Lleida

Table of contents

Our code can be accessed in this repository.

Main decisions

The chosen approach is *strategic mapping of tasks*.

In the previous OMP implementation, we defined a strategy in which each thread would compute its part of the work. In this assignment, we haven't been able to use the same OMP implementation, as the mentioned behavior has been applied to MPI ranks. To do that, each rank defines the size of the sub-grid it will work on through the function *compute_local_rows*.

Each rank adds two halo rows, on for the top of the sub-grid and one for the bottom. These rows aren't part of the work that the rank must compute, but copies of the border rows belonging to the neighboring ranks; which will be needed to solve the heat equation for the top and bottom rows of the rank's range.

With that in mind, each rank initializes its sub-grid and computes its workload in the *solve_heat_equation* function, where all the steps of the simulation are performed. The first thing to do in every step is the data exchange of the rows that belong to the neighboring ranks. This process is done in the *exchange_halos* function in the following way:

- *MPI_Isend*
    - To send the upper interior row (not the halo) to the previous rank. This is not executed by rank 0, as it already computes the upmost row of the entire grid and has no neighbor above it.
    - To send the lower interior row (not the halo) to the next rank. This is not executed by the last rank, as it already computes the downmost row of the entire grid and has no neighbor below it.
- MPI_Irecv
    - To receive the values from the last row computed by the previous rank, storing it in the upper halo row of the local grid. This is not executed by rank 0, as it already computes the upmost row of the entire grid and has no neighbor above it.
    - To receive the values from the first row computed by the following rank, storing it in the lower halo row of the local grid. This is not executed by the last rank, as it already computes the downmost row of the entire grid and has no neighbor below it.

These data exchange operations are non-blocking, so that all the send/recv calls can be issued as fast as possible. With that in mind, we must ensure that every send and receive operation is completed before moving on, so we don't risk continuing with corrupted or incomplete data. To do that, the MPI_Waitall operation is employed, and the execution can move on to computing the new values of the heat equation for the current step.

To compute the new grid with the exchanged information, the array is iterated in the same manner as in the serial or OMP implementations. However, in this case, we account for the invalid halo rows in the first and last ranks. OMP parallelization has been applied to the

outmost loop for iterating the grid, with a dynamic schedule to avoid idle threads and implement better load balancing.

After each rank finishes executing *solve_heat_equation*, rank 0 receives through *MPI_Recv* the local grids computed by the other ranks, which send the grid using *MPI_Send.* At this point, rank 0 has all the grid information and can print it in the *bmp* file.

MPI + OMP Testing

In this section, we aim to evaluate the performance impact of varying thread and process counts in parallel computations. By systematically comparing different combinations of OpenMP threads and MPI processes, we seek to identify the most efficient configuration for different grid sizes and problem scales. This analysis will help highlight the trade-offs between computation speed, scalability, and resource utilization, providing insights into optimizing parallel execution strategies. These are the results of this analysis:

OMP Threads	MPI Processes			
	2	4	6	8
2	0,0484	3,32878	3,44328	3,955316
4	3,236673	10,376433	9,870983	10,3417
8	0,464052	0,074326	0,130647	0,2258

Table 1. Results for 100 x 100 and 1000 steps.

OMP Threads	MPI Processes			
	2	4	6	8
2	1,5971	4,670613	4,194363	4,086657
4	5,47945	11,86924	10,612981	11,14735
8	2,074516	1,732878	2,462836	0,948052

Table 2. Results for 1000 x 1000 and 1000 steps.

OMP Threads	MPI Processes			
	2	4	6	8
2	6,1666	9,284303	7,695724	6,55392
4	9,80015	15,32597	12,834182	12,868016
8	6,72219	6,651531	8,765137	4,217445

Table 3. Results for 2000 x 2000 and 1000 steps.

And these are the plotted results

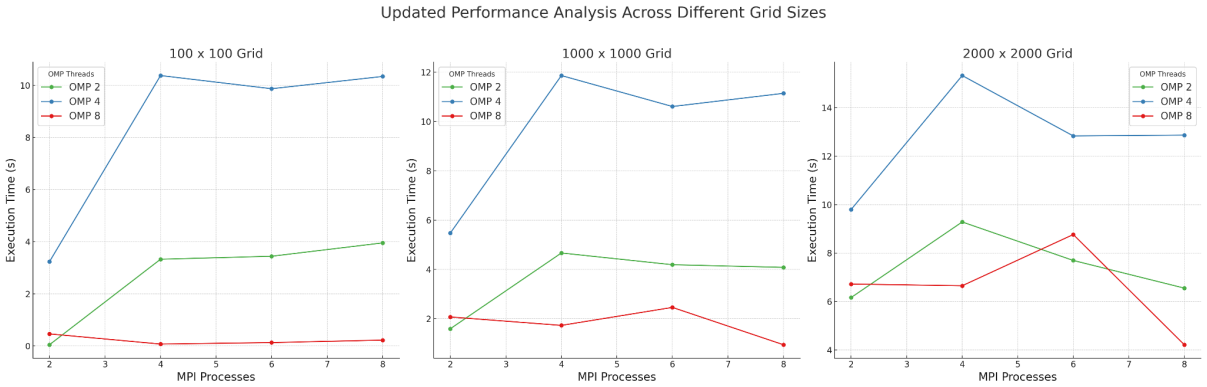

Image 1. Plotted results of the previous analysis.

This study gives us insights in which is the best option to start taking mesures. The results seem to be 8 threads and 8 processes, this approach is not the best with an smaller matrix since the overhead takes more time than the execution. For larger sizes, this configuration is better since it is way faster.

Comparing OMP and OMP + MPI

In this section, we present a comprehensive comparison of the performance across different parallelization strategies, including pure OMP (analyzed on the previous assigment), combined OMP + MPI, and serial execution. The aim is to assess the impact of adding MPI processes to the existing OMP-based parallelization, highlighting the improvements in execution time, speedup, and scalability. By examining the results from both approaches, we aim to identify the most efficient configuration for various grid sizes and problem complexities, while also quantifying the trade-offs in communication overhead and load balancing.

SERIAL				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,11	1,02	10,2
1000x1000	1,14	12,11	109,47	1073,03
2000x2000	4,52	47,73	476,43	4.316,25

Table 4. Serial Results.

CONCURRENT OMP (4 threads)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,04	0,31	3,05
1000x1000	0,44	3,39	29,71	289,84
2000x2000	1,69	13,29	128,44	1.162,82

Table 5. Results with 4 threads.

CONCURRENT OMP + MPI (8 threads and 8 processes)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,029	0,221	1,070	10,639
1000x1000	0,251	4,720	10,843	105,144
2000x2000	0,909	4,462	105,485	354,609

Table 6. Results with 8 threads and 8 processes.

With those results, we can get the speed-up to compare the results.

SPEED UP (8 Threads and 8 Processes)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,34	0,50	0,95	0,96
1000x1000	4,54	2,57	10,10	10,21

2000x2000	4,97	10,70	4,52	12,17

Table 7. Speed Up of concurrent OMP and MPI.

With this results we can say that on small matrix sizes have too little computation per parallel task, making the overhead of thread/process management more dominant. For medium matrices benefit from parallelization once the problem size justifies the overhead, leading to near-linear scaling at the higher step counts. And finally, for larger sizes clearly benefit from parallelism but may require careful tuning to avoid imbalance or communication overhead.

We can also compare those results with the OMP Results obtained in the previous assignment.

SPEED UP (OMP 4 Threads)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	1,27	2,91	3,25	3,35
1000x1000	2,61	3,57	3,68	3,70
2000x2000	2,67	3,59	3,71	3,71

Table 8. Speed Up of concurrent with OMP

The results gives us an improvement on almost all the values except for the lower matrix sizes.

Conclusions

Effective parallelization requires careful consideration of both matrix size and computational workload. For future implementations, focusing on minimizing overhead and improving workload distribution will be critical to achieving higher parallel efficiency.

The combined OMP + MPI approach consistently outperformed pure OMP and serial versions, achieving the fastest execution times across all tested configurations. For instance, the 2000 x 2000 matrix at 100,000 steps was solved in 354.609 s, a substantial reduction from the serial time of 4,316.25 s.

The hybrid approach demonstrated better scalability, effectively utilizing available computational resources.

We've created a faster version than the previous ones, achieving significantly lower execution times and better scalability. This improvement was made possible by combining OMP and MPI to efficiently distribute the computational load across multiple threads and processes. However, while this approach reduced overall execution time, it also introduced parallel overhead and load balancing challenges, highlighting the need for further optimization to fully exploit the available computational power.