13 APRIL 2025

HIGH PERFORMANCE COMPUTING

OPENMP PRACTICE

Oriol Vilalta Arrondo
Joel Lloret Vidal

Escola Politècnica Superior
Universitat de Lleida

Table of Contents

MAIN DECISIONS **2**

Initial version 2
Revised version 2

SCALABILITY AND RESULTS **3**

Execution times 3
Speed-ups 4
Efficiencies 5
Thread scaling 6

Table of figures

Plot 1: Speed-up (4 threads) 4
Plot 2: Speed-up (8 threads) 4
Plot 3: Efficiency (4 threads) 5
Plot 4: Efficiency (8 threads) 5
Plot 5: Thread scaling 6

Our code can be accessed in this repository.

Main decisions

The parallelization of this implementation of the *Heat Diffusion Equation* problem has mainly been centred around the *solve_heat_equation* function. Here, the program performs all the steps of the equation, calculating the entire grid in each of them.

Initial version

The first important thing to notice at this point is that steps cannot be distributed among threads, as each step relies entirely on the previous one. With that in mind, the only option is to parallelize the calculation of the matrix within each step; by dividing the matrix in equal parts and assigning one to each thread. The simplest way to do this is by applying the directive *#pragma omp parallel for* on the first loop of the matrix, having *j* as private variable. The loops for the boundary conditions can be handled similarly.

However, it has been noticed that this implementation may introduce a significant amount of overhead; as threads will be created/destroyed every time the directive is used, which would be several times within each step. This version refers to the file *OMP/heat_ompver1.c* of the repository.

Revised version

This implementation refers to the *OMP/heat_omp.c* of the repository. The version developed is an alternative in which the *solve_heat_equation* call is located inside an *#omp parallel* region, and each thread calculates its part of the matrix based on their *tid*, done only once before looping through the steps. Below is the pseudocode for the steps distribution:

```
base_steps = (nx-2) / number_of_threads;

remainder = (nx-2) % number_of_threads;

from = thread_id * base_steps + (thread_id < remainder ? thread_id : remainder)+1;

to = from + base_steps + (thread_id < remainder ? 1 : 0);
```

To ensure that threads don't move on to the next step before the entire matrix is calculated, a barrier is used for synchronization right after finishing their part of the matrix. The boundary conditions operations are performed only once (by one thread) each step, by using the *#pragma omp single* directive. After that, *grid* and *new_grid* are swapped, and the execution can safely move to the next step.

This implementation only introduces the overhead of creating threads once, and it has been tested and determined that it provides better performance than the version initially mentioned.

In addition to this, the *initialize_grid* function has also been parallelized, by applying *#pragma omp parallel for* on the outmost loop of the function. As this function is purely computational, and each cell is independent of the other, there aren't any apparent drawbacks to parallelization.

Regarding the *write_grid* function, which prints the grid into the file, we have opted to not parallelize it because *fwrite* and *fputc* operate on a shared resource, and writing from multiple threads may cause race

conditions or corrupted output unless you synchronize access (which may introduce considerable overhead).

Scalability and results

Execution times

Below are tables with the execution times of our implementation, using 1, 4 and 8 threads:

Serial execution times				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,11	1,02	10,2
1000x1000	1,14	12,11	109,47	1.073,03
2000x2000	4,52	47,73	476,43	4.316,25

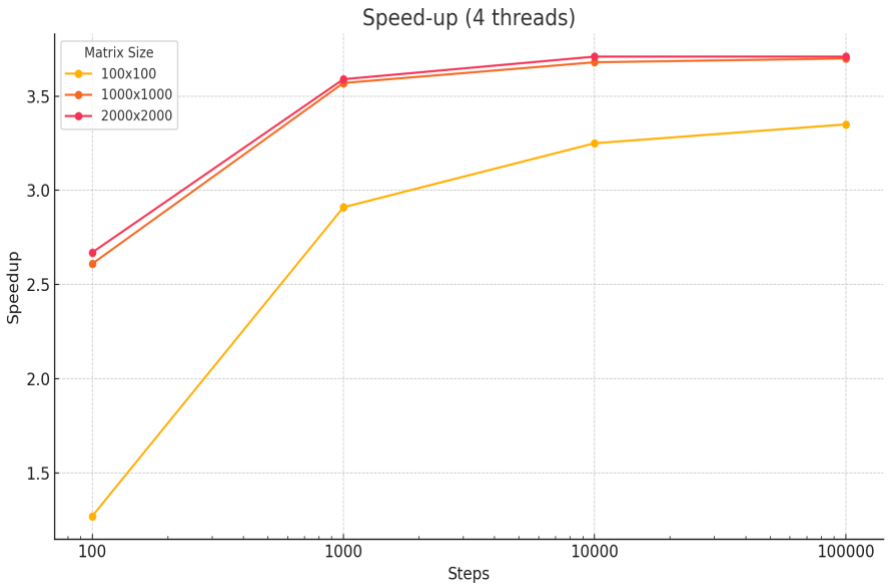
Concurrent execution times (4 threads)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,04	0,31	3,05
1000x1000	0,44	3,39	29,71	289,84
2000x2000	1,69	13,29	128,44	1.162,82

Concurrent execution times (8 threads)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,05	0,44	4,32
1000x1000	0,45	3,57	30,75	297,72
2000x2000	1,72	13,75	135,73	1.181,73

Serial runtime grows substantially with both matrix size (100×100, 1000×1000, and 2000×2000) and the number of steps (100 up to 100,000). When switching to parallel implementations (4 threads or 8 threads), the runtime drops significantly. This is especially noticeable for larger matrices and more steps, where parallelism delivers a higher overall benefit.

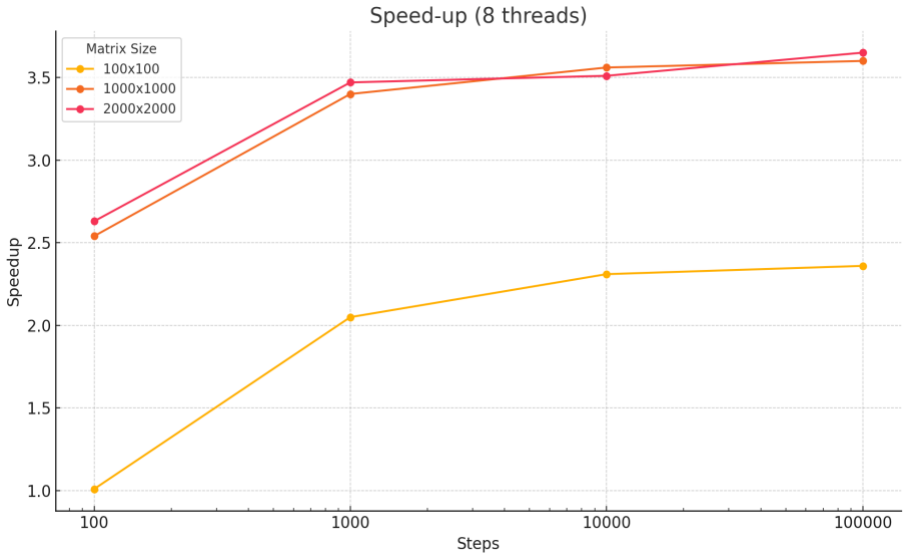
Speed-ups

Below are the speed-ups computed from the execution times above:



Plot 1: Speed-up (4 threads)

The speed-up for 4 threads on large problems (e.g., 1000×1000 or 2000×2000 with 100,000 steps) often exceeds 3.5x. This is relatively close to the ideal speed-up of 4x, indicating that the parallel implementation scales well up to 4 threads on larger data sets and step counts.

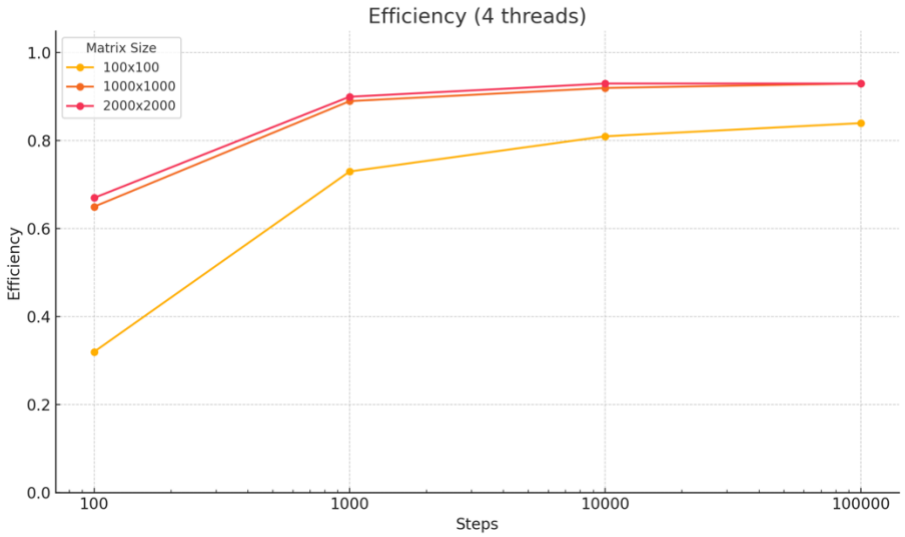


Plot 2: Speed-up (8 threads)

The speed-up for 8 threads is around 3.5–3.7x, which is still good but notably below the theoretical maximum of 8x. As the number of threads doubles from 4 to 8, the speed-up does not fully double, due in part to overheads and possibly limited memory bandwidth.

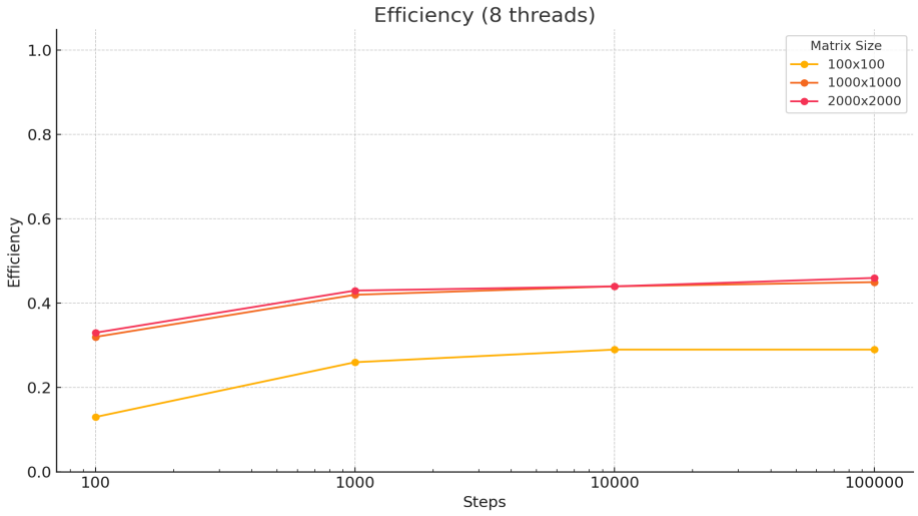
Efficiencies

Parallel efficiency is essentially the speedup divided by the number of threads. An efficiency near 1.0 indicates near-perfect scaling.



Plot 3: Efficiency (4 threads)

With 4 threads, efficiency often stabilizes around 0.8–0.9 for large matrix sizes.

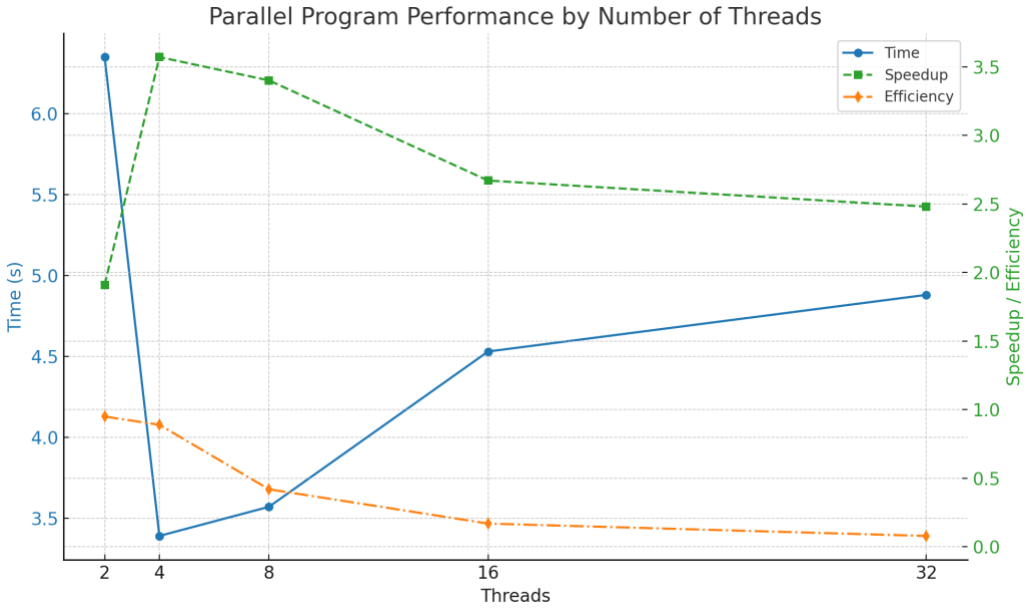


Plot 4: Efficiency (8 threads)

With 8 threads, efficiency falls to around 0.4–0.5 under similar conditions. This drop reflects that, while total speed-up is still good, each added thread contributes less incremental performance.

Thread scaling

To test performance across a wider number of threads, a 1000 step 1000x1000 size execution has been performed with 2, 4, 8, 16 and 32 threads. Although this is not as exhaustive as the previous metrics taken, this will give us a better understanding of how the implementation scales with thread count. To make this as representative as possible, we have used an intermediate problem and matrix size.



Plot 5: Thread scaling

The additional data for 2, 4, 8, 16, and 32 threads shows that:

- Speed-up climbs well up to 4 or 8 threads (speed-up of around 3.5).
- Beyond 8 threads, speed-up actually decreases (around 2.67 for 16 threads, around 2.48 for 32 threads), and efficiency collapses (0.17 or 0.08).

This result is common in parallel programs when the problem size is not large enough to keep many threads busy, or when shared-memory overhead undermines scaling.