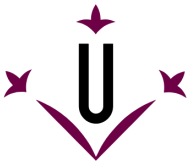


Universitat de Lleida

HPC Project:
Heat Diffussion Equation
High Performance Computing

High Performance Computing:
Delivery 1:
Implementation with OpenMP

Gerard Jaimejuan Guardia
Yanis Amrane
12/04/2025

1. Introduction

The goal of this project was to solve the 2D heat diffusion equation using parallel programming techniques. This first delivery focused on using OpenMP to parallelize a serial version of the code and evaluate how performance changes when executed on multiple threads. The main objectives were to optimize computation time, analyze scalability, and justify the design choices made during development.

2. Parallelization Strategy

2.1 Serial Code Analysis

The original serial program initializes a 2D grid and simulates the spread of heat over time using a finite difference method. Key parts of the code include:

- A nested loop in `solve_heat_equation()` for updating grid values.
- Two separate grids (`grid` and `new_grid`) to avoid overwriting data during updates.
- Initialization that applies a heat source along both diagonals.
- Dirichlet boundary conditions applied at each time step.
- Writing the final result to a BMP file for visualization.

2.2 Parallelization Decisions

We identified the double loop that updates the grid as the main parallelization target. We parallelized this computation using OpenMP with the following directive:

```
#pragma omp parallel for private(j) collapse(2)
for (i = 1; i < nx - 1; i++) {
    for (j = 1; j < ny - 1; j++) {

        new_grid[i * ny + j] = grid[i * ny + j]
        + r*(grid[(i + 1) * ny + j] +grid[(i - 1) * ny + j] -2*grid[i * ny + j])
        +r*(grid[i * ny + j+1] +grid[i * ny + j-1] -2*grid[i * ny + j]);
    }

}
```

We used the `collapse(2)` clause here, which merges the two nested loops into a single iteration space. This provides finer granularity for OpenMP, distributing the total number of iterations across the threads more evenly. In our tests, this approach produced stable results and worked as expected for 1, 2, and 4 threads.

The boundary conditions were also handled in parallel using two additional loops:

```
#pragma omp parallel for
for ( i = 0; i < nx; i++) {
    new_grid[0*ny+i] = 0.0;
    new_grid[ny*(nx-1)+i] = 0.0;
}
#pragma omp parallel for
for (j = 0; j < ny; j++) {
    new_grid[0+j*nx] = 0.0;
    new_grid[(ny-1)+j*nx] = 0.0;
}        // Swap the grids
```

We also optionally parallelized the grid initialization using:

```
void initialize_grid(double *grid, int nx, int ny,int temp_source) {
    int i,j;
    #pragma omp parallel for collapse(2)
    for(i = 0; i < nx; i++) {
        for (j = 0; j < ny; j++) {
            if(i==j) grid[i * ny + j] = 1500.0;
            else if(i== nx-1-j) grid[i * ny + j]=1500.0;
            else grid[i * ny + j]=0.0;
        }
    }

    }
```

Since this part executes very quickly, it doesn't significantly affect performance.

2.3 Output Parallelization

The output writing part (BMP file generation) was not parallelized. Its computational cost is negligible, and parallelizing it would add unnecessary complexity.

3. Performance Evaluation

3.1 Test Conditions

We tested grid sizes ranging from 100x100 to 2000x2000 with a number of steps between 100 and 100000. The program was executed using 1, 2, and 4 threads.

3.2 Execution Time Summary (s)

	Steps			
Matrix Size	100	1000	10000	100000
100x100	0.009962	0.041997	0.388400	3.707282
1000x1000	0.487245	4.088999	36.685650	367.450552
2000x2000	1.889420	16.070740	157.055077	1448.453027

3.3 Figures

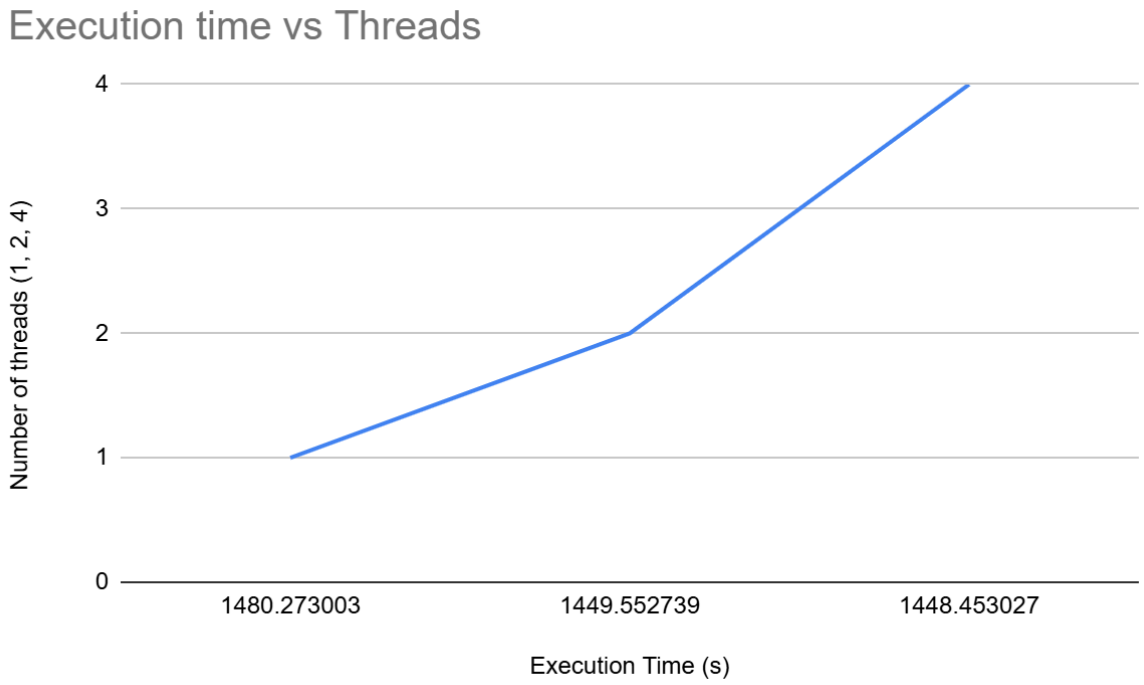


Figure 1: Execution Time vs Threads (2000x2000, 100000 steps)

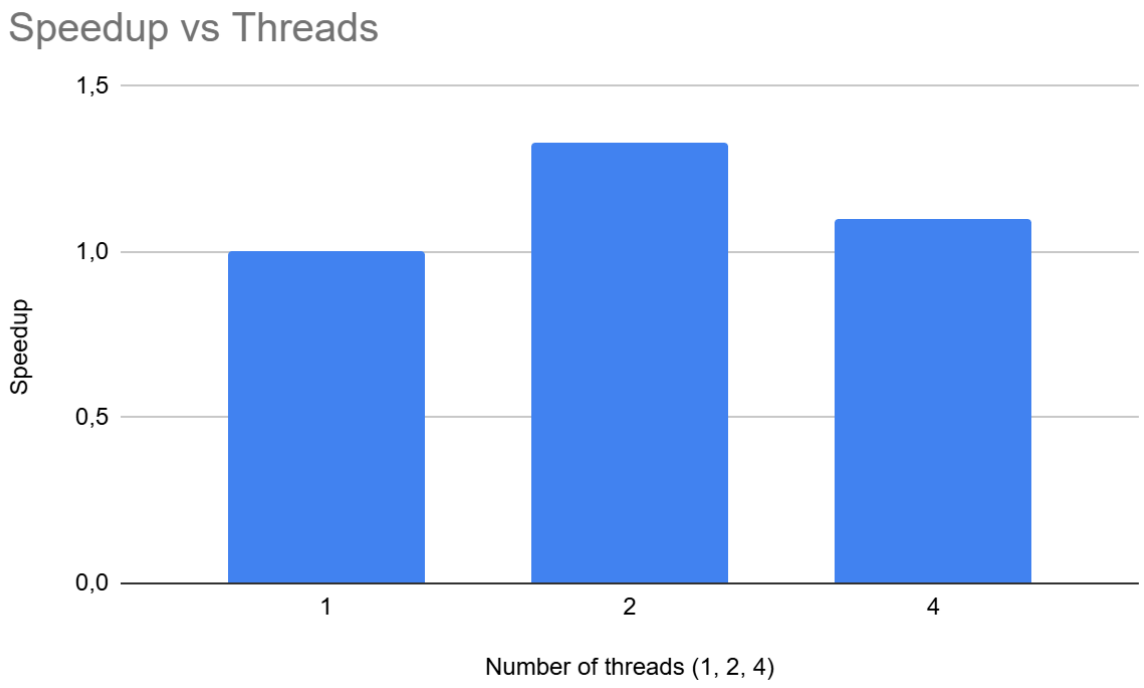


Figure 2: Speedup vs Threads (100x100, 100 steps)

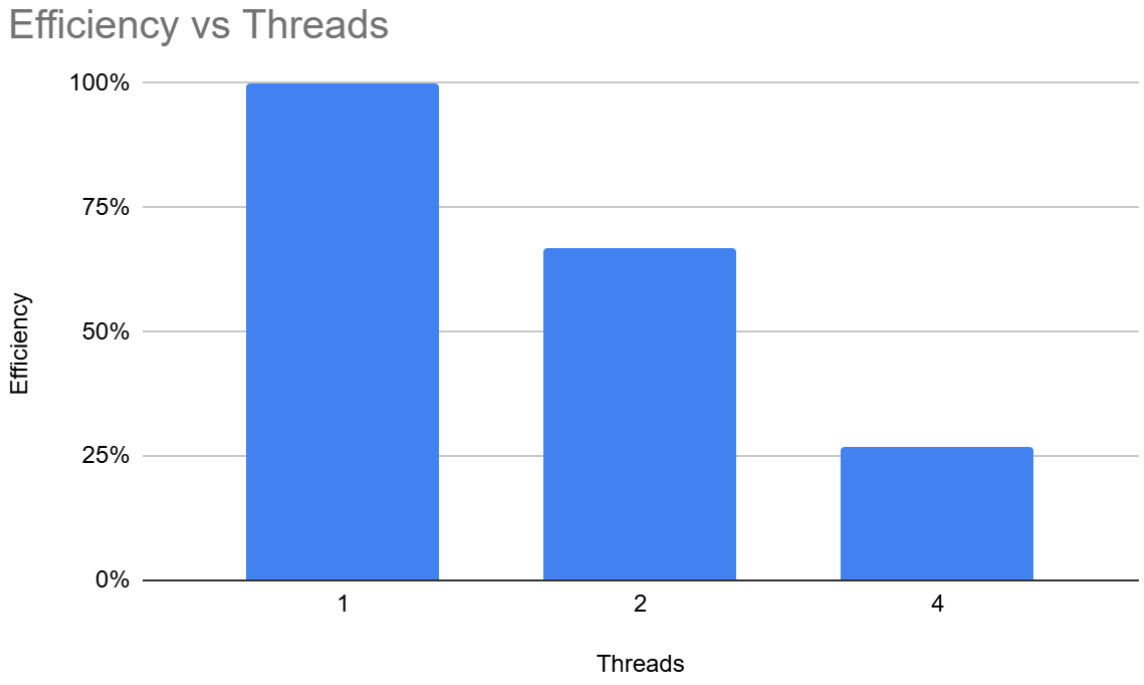


Figure 3: Efficiency vs Threads (100x100, 100 steps)

4. Discussion of Results

4.1 Speedup and Efficiency

The program does not show a noticeable speedup beyond 1 thread. For 2 and 4 threads, the speedup remains very close to 1 and efficiency drops significantly. This indicates that thread creation and synchronization overhead cancels out the potential parallel gains.

4.2 Interpretation

There are several reasons for these results:

- The computation per loop iteration is relatively light, so the potential gain is limited.
- OpenMP thread management adds overhead.
- Memory bandwidth may become saturated.
- The workload per thread is too small to benefit from parallelism.

5. Conclusion

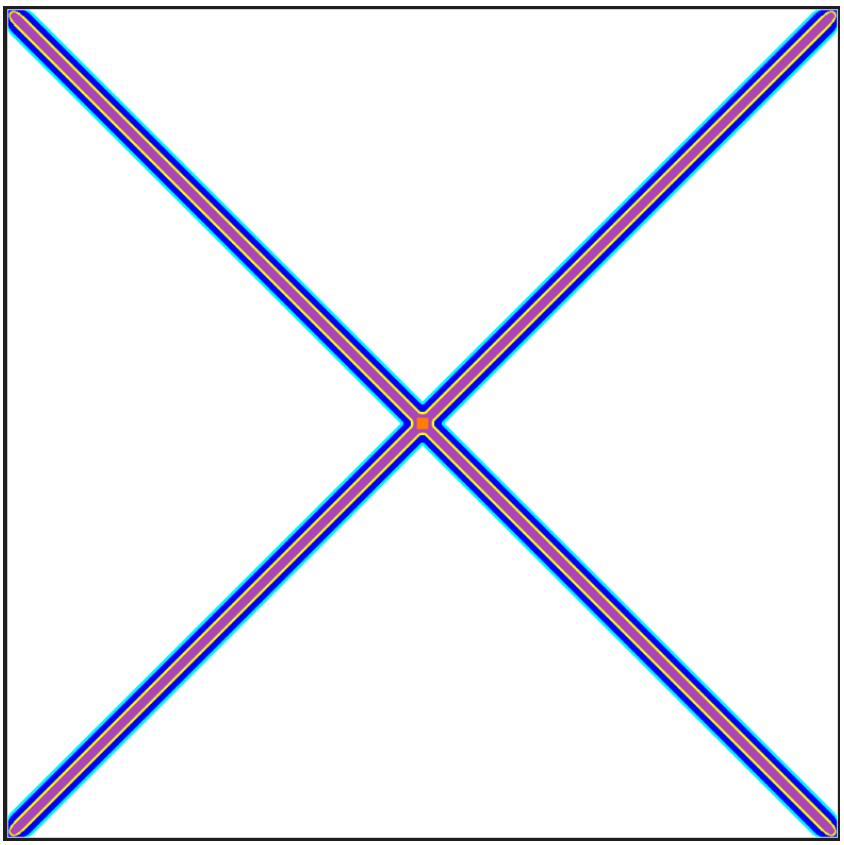
The OpenMP implementation of the heat diffusion simulation works correctly with up to 4 threads. However, performance gains are very limited. This project shows that parallelization is not always effective, especially when the workload per thread is small. Possible future improvements could include tiling, better memory access patterns, or using a hybrid OpenMP + MPI approach.

6. Annexes

- run_heat_omp.sh

```
#!/bin/bash

## Specifies the interpreting shell for the job.
#$ -S /bin/bash

## Specifies that all environment variables active within the qsub utility be exported to the context of the job.
#$ -V

## Specifies the parallel environment
#a$ -pe smp 4

## Execute the job from the current working directory.
#$ -cwd

## The  name  of  the  job.
#$ -N HeatOpenMP

##send an email when the job ends
#$ -m e

##email addrees notification
#$ -M ya9@alumnes.udl.cat

##executer avec 4 theads
#$ -v OMP_NUM_THREADS=4

##Se placer dans le bon répertoire de soumission(c'est une verification en plus)
cd $SGE_O_WORKDIR

##compiler le code si pas encore compilé
gcc -fopenmp heat_openmp.c -o heat_openmp -lm

## In this line you have to write the command that will execute your application.
./heat_openmp 100 100 output_100_100_4.bmp
```

- Output of heat_openmp with a matrix 1000x1000 and 1000 steps



- Output of heat_openmp with a matrix 2000x2000 and 1000 steps