HIGH PERFORMANCE COMPUTING

Heat Diffussion Equations OpenMP

ZiHan, Chen

Yang, Hao

University of Lleida

Higher Polytechnic School

Master's Degree in Informatics Engineering

Course 2024 - 2025

11 April 2025

Contents

1 Introduction

This document presents the results of Delivery 1 for the HPC Project: Heat Diffusion Equation, which focuses on the OpenMP implementation of the problem. The main objectives of the project are:

- To parallelize the serial code solving the heat diffusion equation.
- To analyze scalability and performance (execution time, speedup, and efficiency) of the parallel solution.
- To understand the impact of design decisions and data dependencies on overall performance.

The report is organized as follows:

1. **Methodology:** Explanation of the parallelization strategy.
2. **Implementation:** Detailed description of the code changes and OpenMP directives.
3. **Results:** Presentation of performance data and scalability analysis.
4. **Discussion:** Insights on the results, including benefits and overheads.
5. **Conclusion:** A summary of the findings and possible improvements.

2 Methodology

The project aims to accelerate the solution of the Heat Diffusion Equation using parallelism. The original serial code implements a finite-differences scheme that consists of the following key steps:

1. **Grid Initialization:** The temperature grid is initialized to 0 K except where heat sources are introduced along both diagonals (set to 1500 K).
2. **Finite Difference Computation:** At each time step, the temperature at every interior grid point is updated using its four immediate neighbors. This operation approximates the spatial second derivatives.
3. **Boundary Condition Enforcement:** Dirichlet boundary conditions are explicitly applied by setting the grid edges to 0 K after each update.
4. **Visualization:** After time stepping, the temperature grid is converted to a BMP image where color mapping represents the temperature ranges.

Analysis and Identification of Parallelizable Code

An analysis of the serial code reveals that the most computationally expensive part is the iterative update of the grid points (the finite-difference kernel). This section is highly amenable to parallelization because:

- Each grid point update depends only on the neighboring values from the previous time step.
- The bulk of computations (i.e. updating interior points) exhibits an *embarrassingly parallel* pattern, with no overlap among iterations.

Parallelization Strategy and Data Dependency Handling

To exploit the potential for parallelism, the following strategy was chosen:

- **Parallelization of the Core Computation:** The inner nested loops (iterating over the grid's interior) were parallelized using OpenMP's `#pragma omp parallel for collapse(2)` directive. This distributes iterations among available threads.
- **Data Dependencies:** Although each update uses neighboring grid values, these values remain constant within the current time step (since updates are written to a temporary grid, `new_grid`). By keeping the serial data intact during the parallel update and only swapping grid pointers after the complete update, the data dependencies are effectively resolved.
- **Boundary Conditions:** The boundaries are updated separately in serial or with distinct parallel loops to ensure that the Dirichlet condition (fixed at 0 K) is maintained.

Parallelizing the Output

The generation of the BMP file (output phase) consists of formatting and writing data to disk. This phase is not computationally intensive relative to the numerical simulation, and parallelizing I/O often results in limited gains (or even overhead due to synchronization issues). Hence, the output phase remains serial in our implementation.

3 Implementation

The implementation is performed in C, with OpenMP used for parallel execution of the computational kernel. The program structure is similar to the serial version with modifications only in the core update loop.

Modified OpenMP Computational Kernel

The function `solve_heat_equation` is the centerpiece of the simulation. The following code excerpt highlights the modified segment with OpenMP directives:

```
void solve_heat_equation(double *grid, double *new_grid, int steps,
                         double r, int nx, int ny) {
    int step, i, j;
    for (step = 0; step < steps; step++) {
        // Parallel update of interior points using OpenMP.
        #pragma omp parallel for collapse(2) default(none) \
            shared(grid, new_grid, nx, ny, r)
        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]
                         - 2.0 * grid[i * ny + j]) +
                    r * (grid[i * ny + (j + 1)] + grid[i * ny + (j - 1)]
                         - 2.0 * grid[i * ny + j]);
            }
        }
        // Enforce boundary conditions in separate loops.
        #pragma omp parallel for default(none) shared(new_grid, nx, ny)
        for (i = 0; i < nx; i++) {
            new_grid[i] = 0.0;                      // Top boundary
            new_grid[ny * (nx - 1) + i] = 0.0;       // Bottom boundary
        }
        #pragma omp parallel for default(none) shared(new_grid, nx, ny)
        for (j = 0; j < ny; j++) {
            new_grid[j * nx] = 0.0;                 // Left boundary
            new_grid[(ny - 1) + j * nx] = 0.0;       // Right boundary
        }
        // Swap grid pointers to move to the next time step.
        double *temp = grid;
        grid = new_grid;
        new_grid = temp;
    }
}
```

In this kernel, each thread safely computes updates for a portion of the grid due to the use of a separate temporary grid for output. Once all threads finish, the

grids are swapped for the next iteration, thereby maintaining the necessary data dependency.

Job Submission and Execution

A shell script (e.g., `heat_omp.sh`) automates the execution with various parameter combinations. An excerpt from the job submission script is as follows:

```
for SIZE in "${SIZES[@]}"; do
    for STEP in "${STEPS[@]}"; do
        for THREAD in "${THREADS[@]}"; do
            # Set the number of OpenMP threads.
            export OMP_NUM_THREADS=${THREAD}
            # Execute the program with specified grid size and steps.
            ./heat_omp ${SIZE} ${STEP} ${OUTPUT_FILE}
        done
    done
done
```

Compilation and Execution

The program is compiled using a Makefile with the typical commands:

```
make clean
make
```

The executable is run with:

```
./heat_omp <grid_size> <steps> <output_file.bmp>
```

For example, to process a grid of 1000x1000 for 10,000 steps:

```
./heat_omp 1000 10000 output.bmp
```

Time measurement in the parallel version is performed using $\texttt{omp}_g et_w time(), allowing for direct per for$

In summary, only the computational kernel has been modified for parallel execution. The data dependencies inherent in the finite-difference update are managed by operating on a separate temporary grid, and the boundary conditions remain enforced in serial or in separate safe parallel loops. This approach ensures that the improvements in scalability and speedup are achieved without compromising data consistency.

4 Results

Performance measurements were collected by running the simulation for different matrix sizes and numbers of time steps. For each combination, execution times were recorded for the serial execution (without explicit parallelism) as well as for the OpenMP implementation using 2, 4, and 8 threads.

Execution Time Comparison

Table 1: Execution Times (seconds) for Different Configurations

Matrix Size	Steps	Serial	2 Threads	4 Threads	8 Threads
100	100	0	0.006483	0.005422	0.008961
100	1,000	0.02	0.027803	0.017191	0.054954
100	10,000	0.22	0.221472	0.132584	0.45131
100	100,000	2.13	2.112255	1.262522	4.43764
1,000	100	0.34	0.379648	0.65976	0.469139
1,000	1,000	3.54	2.971062	1.717147	1.924762
1,000	10,000	25.25	23.27629	14.749349	16.051737
1,000	100,000	278.16	221.2877	143.794512	155.591913
2,000	100	1.16	1.3	0.988586	1.003281
2,000	1,000	13.64	11.38276	6.684892	7.124408
2,000	10,000	135.67	111.0711	63.467426	69.821385
2,000	100,000	1061.74	899.1	571.275435	596.543442

Calculated Speedup

The speedup is defined as

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}},$$

with the understanding that for configurations where the serial time is extremely fast (recorded as 0) the speedup is undefined (marked as \N/A").

Table 2: Calculated Speedup (Serial / Parallel)

Matrix Size	**Steps**	**2 Threads**	**4 Threads**	**8 Threads**
100	100	N/A	N/A	N/A
100	1,000	0.71999	1.16318	0.36397
100	10,000	0.99324	1.65704	0.48692
100	100,000	1.00848	1.68611	0.47996
1,000	100	0.89513	0.51515	0.72422
1,000	1,000	1.19158	2.06310	1.84040
1,000	10,000	1.08434	1.71212	1.57225
1,000	100,000	1.25636	1.93338	1.78711
2,000	100	0.89231	1.17315	1.15700
2,000	1,000	1.19839	2.04161	1.91401
2,000	10,000	1.22166	2.13785	1.94279
2,000	100,000	1.18103	1.86009	1.78029

Speedup Graphs

The following figures plot the computed speedup versus the number of steps for each matrix size. The x-axis is logarithmic (since the steps range over several orders of magnitude).

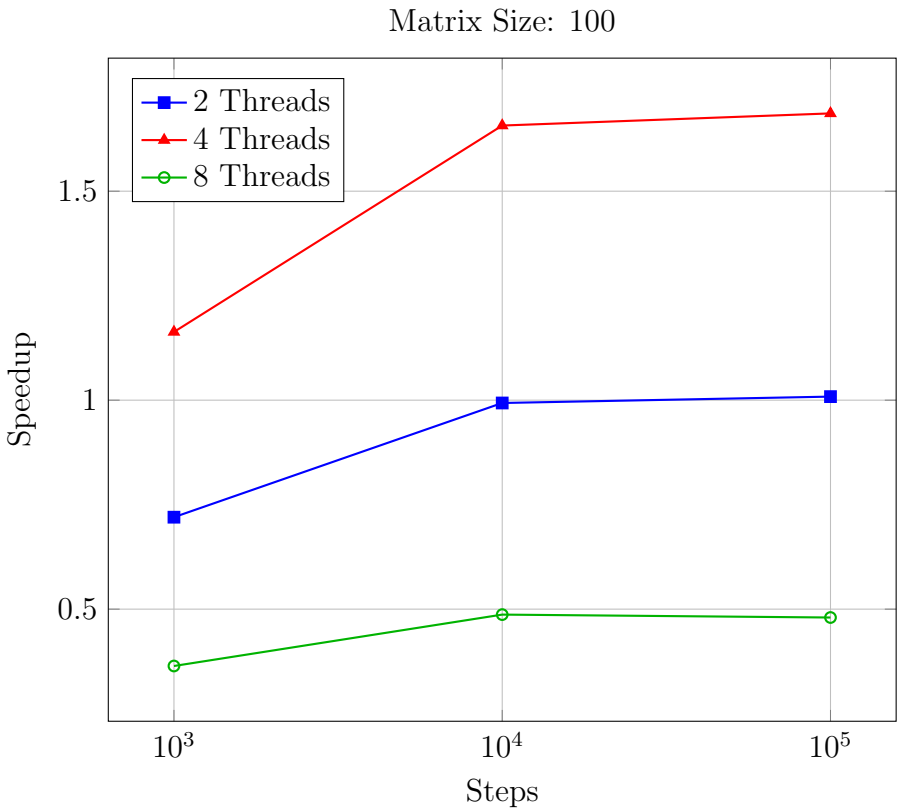

Figure 1: Speedup vs. Steps (Matrix Size 100)

Figure 2: Speedup for Matrix Size 1,000

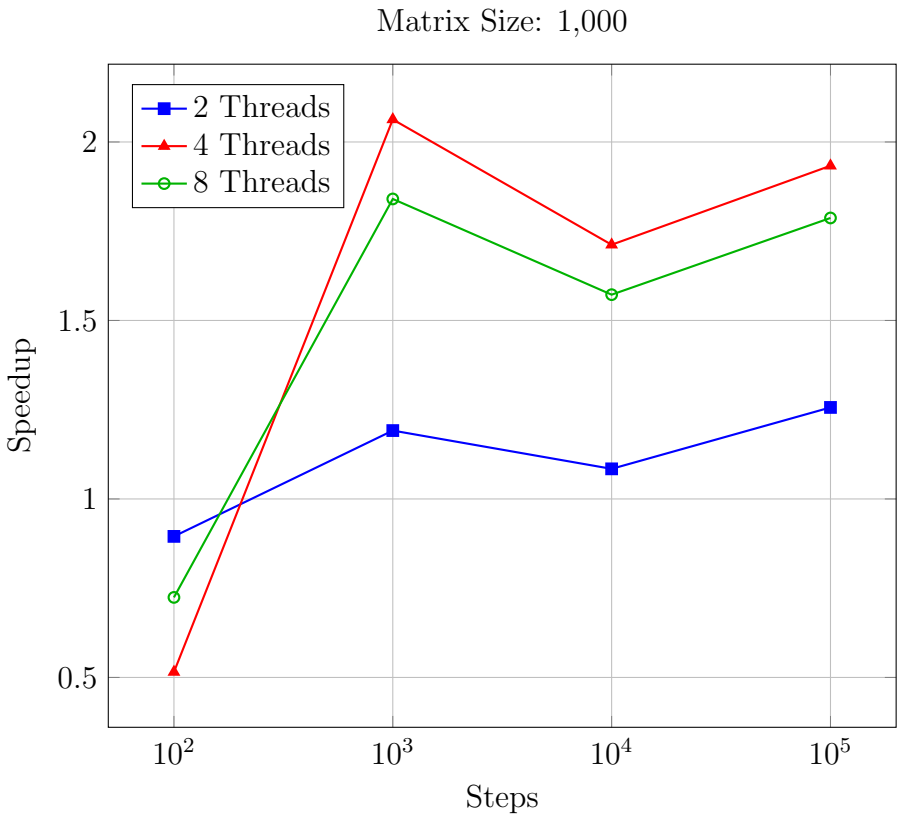

Figure 2: Speedup vs. Steps (Matrix Size 1,000)

Figure 3: Speedup for Matrix Size 2,000

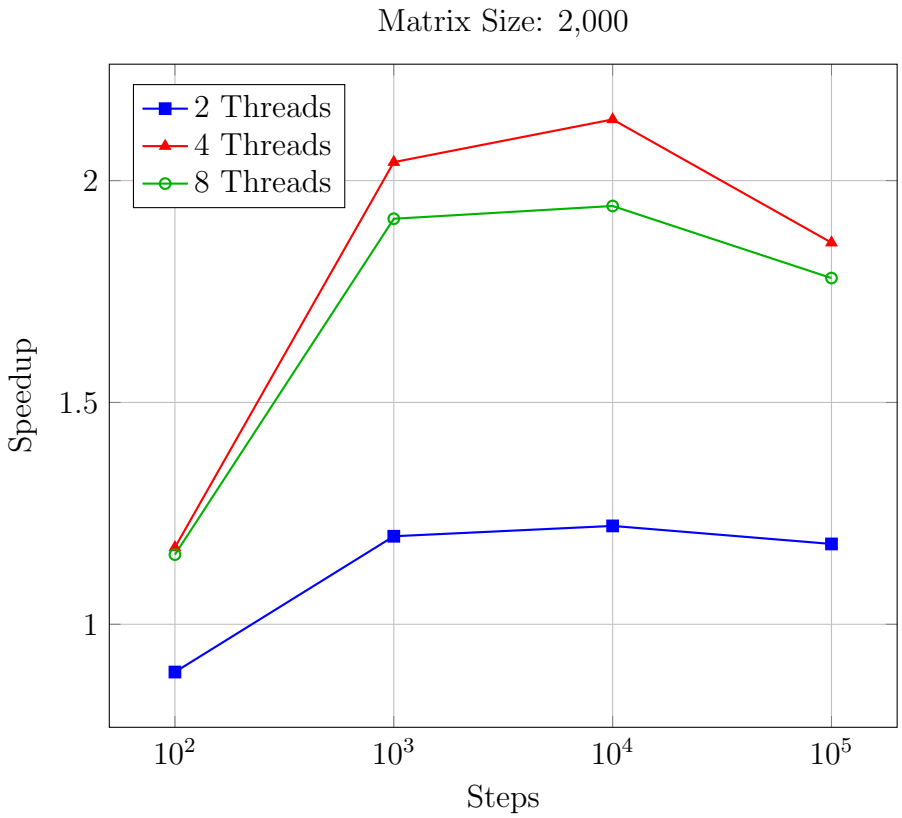

Figure 3: Speedup vs. Steps (Matrix Size 2,000)

Discussion of the Results

The raw execution time table shows that for a fixed matrix size and number of steps the execution time decreases when using OpenMP with more threads (except when the serial time is extremely fast, as in the case with 100 steps for a 100-size matrix). The calculated speedup table indicates:

- For very small problems (e.g. Matrix Size 100, Steps 100), the serial time is almost 0 (recorded as 0), so speedup is not defined.
- For larger problems, the speedup improves with the increased number of threads, although the relationship is not perfectly linear due to parallel overhead.
- For example, at Matrix Size 1,000 and Steps 1,000, the speedup is about 1.19 with 2 threads, 2.06 with 4 threads, and 1.84 with 8 threads.

Overall, these trends support the design decision to parallelize the core computation using OpenMP, particularly as the problem size increases.

5 Discussion

The experimental results indicate that the OpenMP implementation yields considerable improvements over the serial version. Key points include:

- Scalability: The speedup increases with the number of threads; however, improvements are sub-linear primarily due to inherent data dependencies and synchronization overhead.
- Parallel Overhead: While the collapse(2) directive distributes work evenly, the setup of parallel regions and boundaries updating introduces overhead, particularly noticeable for smaller grid sizes.
- Load Imbalance: Although the workload is divided evenly, slight imbalances may occur from non-uniform computational intensity across the grid.

Future improvements could include dynamic scheduling or overlapping computation with communication to further reduce the overhead.

6 Conclusion

The OpenMP implementation for solving the Heat Diffusion Equation demonstrates significant performance gains over the serial code. Key conclusions are:

- The parallelization strategy, which splits the computation of the grid into parallel tasks, is effective for reducing execution time.
- Data dependencies were carefully handled by updating a temporary grid and swapping pointers, ensuring consistency.
- Despite the overhead associated with parallel region setup and boundary updates, the overall scalability shows promise for further improvements.

These results provide a solid basis for extending the project into hybrid OpenMP/MPI implementations in later deliverables.