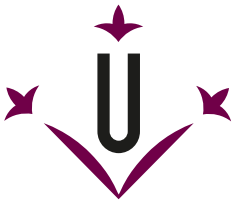
Universitat de Lleida
Màster en Enginyeria Informàtica
High Performance Computing
Prof. Francesc Gine De Sola
Prof. Jordi Ricard Onrubia Palacios



HPC Project

Heat Diffussion Equation

Jordi García Ventura
Christian López García

April 12, 2025

Contents

1 Parallelization

To parallelize the program we did parallelize the following three functions, which were the ones presenting loops.

1.1 initialize_grid

The function *initialize_grid* has one main loop with another loop inside. This function was the easier to parallelize, since we could simply do a single pragma command such as in **Code 1**. If you notice, we swapped the order of the loops, that is because we think that it will be better for cache locality, but we didn't test it.

Code 1 Paralellization of initialize_grid

```
...
#pragma omp parallel for private(j) collapse(2)
for (i = 0; i < nx; i++)
    for (j = 0; j < ny; j++)
...
```

1.2 solve_heat_equation

The *solve_heat_equation* function was pretty straight forward even though there were plenty loops, we parallelized the inner ones as in **Code 2** The first pragma command parallelizes the nested loops by collapsing them, and making the variable j private for each thread. The rest of the for loops are more straight forward as you can see in the code.

Code 2 Paralellization of solve_heat_equation

```
...
for (step = 0; step < steps; step++)
{
    #pragma omp parallel for private(j) collapse(2)
    for (i = 1; i < nx - 1; i++)
        for (j = 1; j < ny - 1; j++)
            // Compute the new grid

    #pragma omp parallel for
    for (i = 0; i < nx; i++)
        // Boundary conditions

    #pragma omp parallel for
    for (j = 0; j < ny; j++)
        // Swap the grids

    ...
}
```

1.3 write_grid

This last parallelizable function, *write_grid*, was the trickiest, since it contained thread-unsafe functions (*fwrite* and *fputc*) inside its loops, so to parallelize it, we had to recreate the same behaviour by replacing the thread-unsafe code that wrote the results into a file, to write the results into an intermediate cache (a matrix allocated in the heap). This code is not strictly needed, actually it may not be that good since the parallelized code is minimal and the implementation provides some overhead (we could calculate if it is worth it by doing more runs) by using the heap as well (by using *pixel_data*), but in this way we made it parallelizable, even if it doesn't provide efficiency.

Code 3 Paralellization of write_grid

```
int row_stride = ny * 3;
int padding = (4 - (row_stride % 4)) % 4;
int padded_row_size = row_stride + padding;
int total_size = nx * padded_row_size;
int i, j, p;

unsigned char *pixel_data = malloc(total_size);
if (!pixel_data) {
    fprintf(stderr, "Failed to allocate memory\n");
    exit(1);
}

#pragma omp parallel for private(j, p)
for (i = 0; i < nx; i++) {
    int row_index = nx - 1 - i; // BMP is bottom-to-top
    int row_offset = i * padded_row_size;

    for (j = 0; j < ny; j++) {
        unsigned char r, g, b;
        get_color(grid[row_index * ny + j], &r, &g, &b);

        int pixel_offset = row_offset + j * 3;
        pixel_data[pixel_offset + 0] = b;
        pixel_data[pixel_offset + 1] = g;
        pixel_data[pixel_offset + 2] = r;
    }

    // Zero out padding at the end of the row
    for (p = 0; p < padding; p++) {
        pixel_data[row_offset + row_stride + p] = 0;
    }
}

fwrite(pixel_data, 1, total_size, file);
free(pixel_data);
```

2 Results

To test the code, we decided to run it with *1*, *2*, *4* and *8* threads, we could remove the *1* and *8* thread execution, since they will be worse than the serial and the *4* thread execution, respectively, since our machines only have 4 cores.

To calculate the speedup and efficiency we used the following formulas:

$$S = \frac{T_1}{T_n} \tag{1}$$

$$E = \frac{S}{n} = \frac{T_1}{n < T_n} \tag{2}$$

IMPORTANT: For the calculation of efficiency, as we are evaluating the threading scalability, we will use as n the number of used threads instead of the ones that the machine has. This means that in the execution of *8* threads, we will use *8* instead of *4*.

2.1 *100 steps* results

First, we will show the executions using *100 steps* in *Figure 1a*, *Figure 1b* and *Figure 1c*. Notice the logarithmic Y-axis in the time graph and that the speedup and efficiency graph contains a column for 'Theoretical speedup' which is [1, 4] since the machine we will run the code on has 4 cores.

2.2 *1000 steps* results

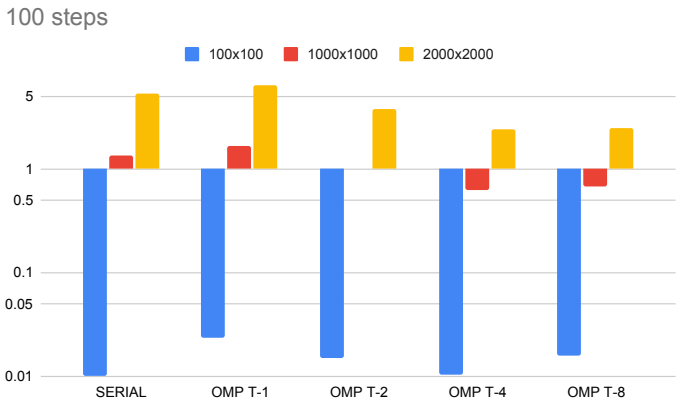
Executions using *1000 steps*, *Figure 2a*, *Figure 2b* and *Figure 2c*.

2.3 *10000 steps* results

Executions using *10000 steps*, *Figure 3a*, *Figure 3b* and *Figure 3c*.

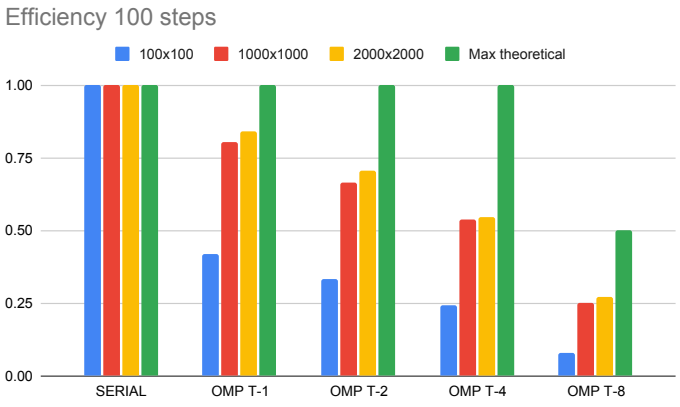
2.4 *100000 steps* results

Executions using *100000 steps*, *Figure 4a*, *Figure 4b* and *Figure 4c*.



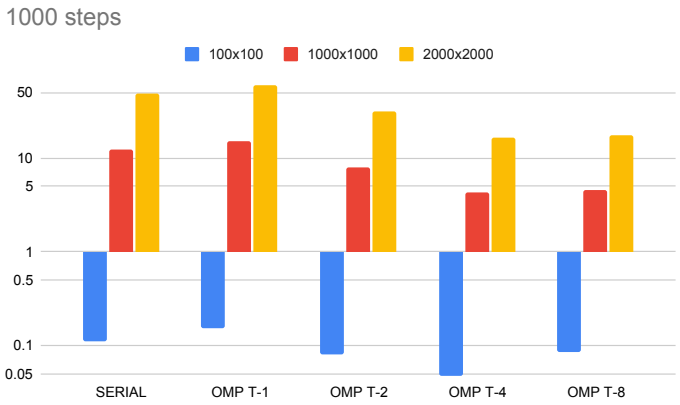
(a) Execution time comparison between different matrix sizes

(b) Speedup comparison of the different executions

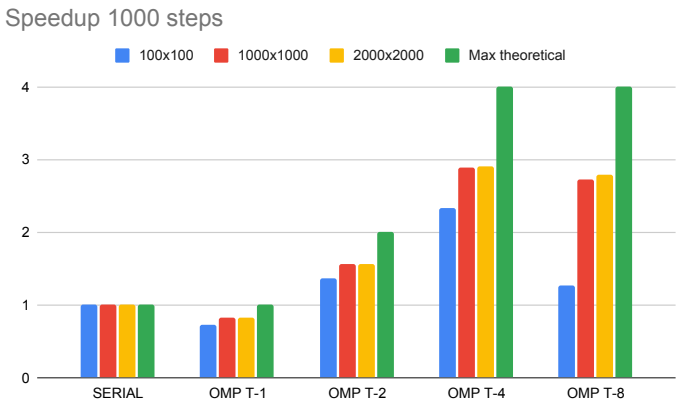


(c) Efficiency comparison of the different executions

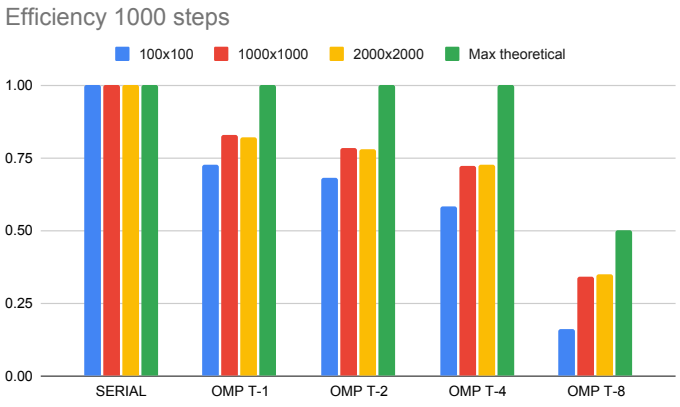
Figure 1: Time, speedup and efficiency graphs of *100 steps* execution



(a) Execution time comparison between different matrix sizes

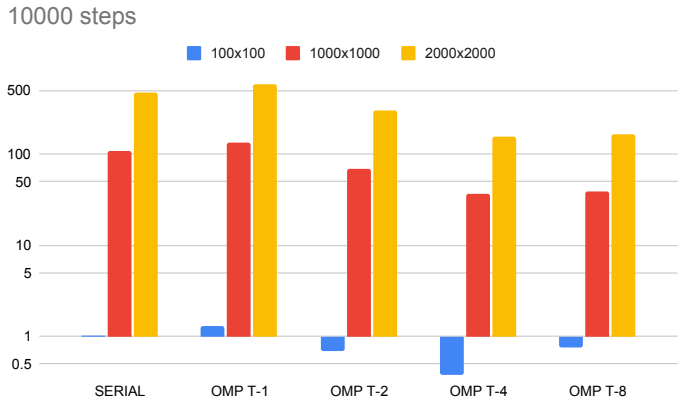


(b) Speedup comparison of the different executions



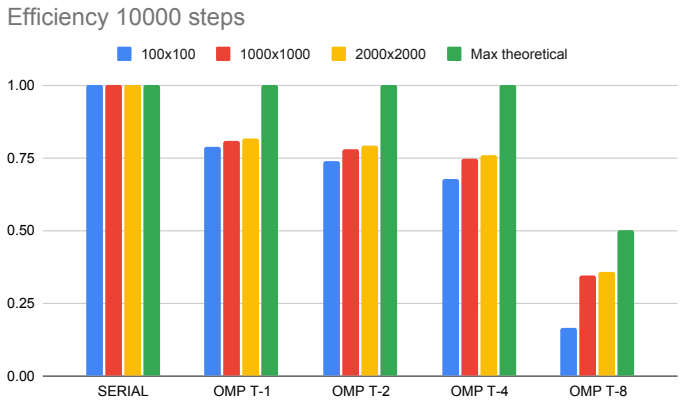
(c) Efficiency comparison of the different executions

Figure 2: Time, speedup and efficiency graphs of *1000 steps* execution



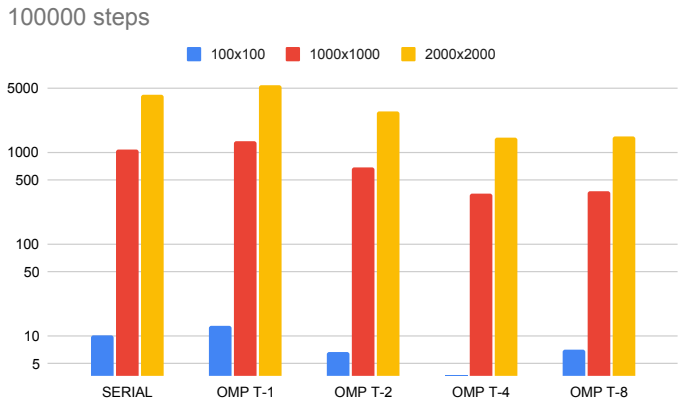
(a) Execution time comparison between different matrix sizes

(b) Speedup comparison of the different executions

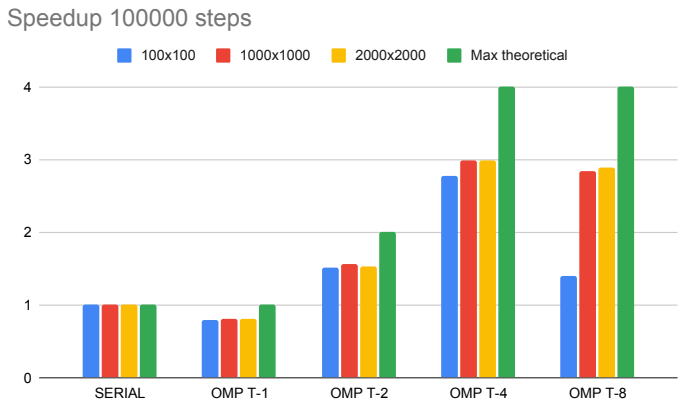


(c) Efficiency comparison of the different executions

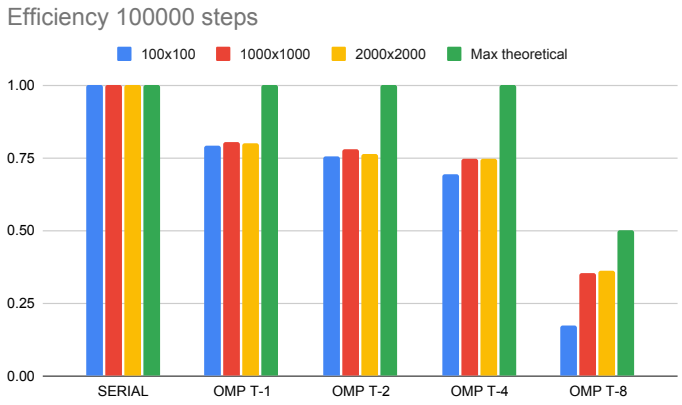
Figure 3: Time, speedup and efficiency graphs of *10000 steps* execution



(a) Execution time comparison between different matrix sizes



(b) Speedup comparison of the different executions



(c) Efficiency comparison of the different executions

Figure 4: Time, speedup and efficiency graphs of *100000 steps* execution

3 Conclusions

From the graphs of the previous *Section 2*, we can extract some conclusions.

- Since we added some overhead, as we predicted, our executions of the code with OMP with 1 thread are worse than the serial code.
- Since the machines the code ran on have 4 cores, the execution of the code with OMP with 8 threads is worse than the execution with 4 threads.
- The execution time starts to improve when using >2 threads (obviously), and it's further improved when the problem size is bigger, with bigger matrix sizes and more steps.
- The efficiency is not that good in any execution, but they are quite similar (excluding the 100 steps execution), so we can 'ignore' it and, in the future, decide to just use the best speedup, which is the 4 threads.

4 Extra work

4.1 Testing

Before doing all these parallelizations, we decided that it would be nice to have some way of testing that, when changing the code, we wouldn't break it. So we decided to do some automatic tests.

These test consist in the comparison of the resulting heat image generated of the code executions of the same size and steps (we assume that the serial code results in a correct image). The implementation of these tests is in the *run-tests.sh* file and can be run by running *Code 4*, which will result in an output similar to *Figure 5*

Code 4 Run tests

```
1 $ make test
```

4.2 Batch running

Before running these tests, we need a way of easily generating those images. To do so, we made 'run-multiple-omp.sh', which implementation sets up all the variable combinations possible and submits them to the queue. To run these executions do it by running *Code 5*

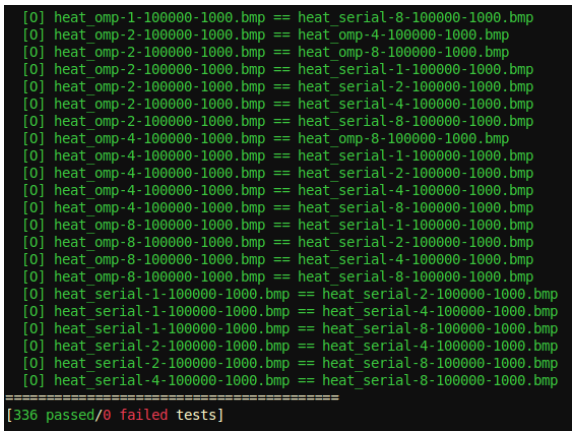


Figure 5: Snippet of the resulting logs of running the tests

Code 5 Run code

```
$ make [all | serial | omp]
```

Once these executions end, we can check it using 'qstat', we can run the tests.

4.3 Resulting time extraction

Once the execution ends and the tests are passing, we have **PLENTY** of log files with the different resulting times and some other logs. To facilitate the extraction of time, we made 'run-extract-time.sh', which can be called by running *Code 6*. Which will generate a 'results.csv' with the resulting times and the information of the different executions to process further.

Code 6 Run time extraction

```
$ make time
```

A Time results

100 STEPS	***Execution***				
Matrix Size	***SERIAL***	***OMP T-1***	***OMP T-2***	***OMP T-4***	***OMP T-8***
100x100	*0.01*	*0.023784*	*0.014906*	*0.010339*	*0.015886*
1000x1000	*1.34*	*1.667607*	*1.007961*	*0.621634*	*0.670591*
2000x2000	*5.33*	*6.344592*	*3.766311*	*2.432052*	*2.452237*

Table 1: 100 steps execution time results

1000 STEPS	***Execution***				
Matrix Size	***SERIAL***	***OMP T-1***	***OMP T-2***	***OMP T-4***	***OMP T-8***
100x100	*0.11*	*0.151429*	*0.08076*	*0.047042*	*0.08615*
1000x1000	*12.37*	*14.936403*	*7.899948*	*4.287486*	*4.538504*
2000x2000	*48.66*	*59.242891*	*31.113219*	*16.738927*	*17.468413*

Table 2: 1000 steps execution time results

10000 STEPS	***Execution***				
Matrix Size	***SERIAL***	***OMP T-1***	***OMP T-2***	***OMP T-4***	***OMP T-8***
100x100	1.02	1.296468	0.691538	0.375628	0.765333
1000x1000	109.6	135.606736	70.097145	36.68247	39.480683
2000x2000	476.73	583.12579	300.843573	157.136858	166.913589

Table 3: 10000 steps execution time results

100000 STEPS	***Execution***				
Matrix Size	***SERIAL***	***OMP T-1***	***OMP T-2***	***OMP T-4***	***OMP T-8***
100x100	10.18	12.819621	6.75271	3.657432	7.254863
1000x1000	1072.98	1331.956969	687.403504	358.492834	377.120077
2000x2000	4310.41	5374.222172	2820.720803	1440.691108	1493.628449

Table 4: 100000 steps execution time results

B Speedup results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.4204507232*	*0.803546639*	*0.8400855406*	*1*
OMP T-2	*0.6708707903*	*1.329416515*	*1.415177876*	*2*
OMP T-4	*0.9672115292*	*2.155609249*	*2.191564983*	*4*
OMP T-8	*0.6294850812*	*1.998237376*	*2.173525642*	*4*

Table 5: 100 steps speedup results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7264130385*	*0.8281779756*	*0.8213643726*	*1*
OMP T-2	*1.362060426*	*1.565833092*	*1.563965464*	*2*
OMP T-4	*2.338335955*	*2.885140616*	*2.906996368*	*4*
OMP T-8	*1.276842716*	*2.72556772*	*2.785599356*	*4*

Table 6: 1000 steps speedup results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7867529318*	*0.8082194383*	*0.8175423008*	*1*
OMP T-2	*1.474973176*	*1.563544421*	*1.584644123*	*2*
OMP T-4	*2.715452522*	*2.987803166*	*3.033852185*	*4*
OMP T-8	*1.332753194*	*2.776041134*	*2.856148519*	*4*

Table 7: 10000 steps speedup results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7940952389*	*0.8055665648*	*0.802052811*	*1*
OMP T-2	*1.507542898*	*1.560917269*	*1.52812359*	*2*
OMP T-4	*2.78337369*	*2.993030539*	*2.991904355*	*4*
OMP T-8	*1.403196725*	*2.845194582*	*2.885864957*	*4*

Table 8: 100000 steps speedup results

C Efficiency results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.4204507232*	*0.803546639*	*0.8400855406*	*1*
OMP T-2	*0.3354353951*	*0.6647082576*	*0.7075889378*	*1*
OMP T-4	*0.2418028823*	*0.5389023123*	*0.5478912457*	*1*
OMP T-8	*0.07868563515*	*0.249779672*	*0.2716907053*	*0.5*

Table 9: 100 steps efficiency results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7264130385*	*0.8281779756*	*0.8213643726*	*1*
OMP T-2	*0.681030213*	*0.7829165458*	*0.7819827322*	*1*
OMP T-4	*0.5845839888*	*0.7212851541*	*0.7267490921*	*1*
OMP T-8	*0.1596053395*	*0.340695965*	*0.3481999195*	*0.5*

Table 10: 1000 steps efficiency results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7867529318*	*0.8082194383*	*0.8175423008*	*1*
OMP T-2	*0.7374865879*	*0.7817722106*	*0.7923220617*	*1*
OMP T-4	*0.6788631305*	*0.7469507915*	*0.7584630463*	*1*
OMP T-8	*0.1665941492*	*0.3470051417*	*0.3570185649*	*0.5*

Table 11: 10000 steps efficiency results

	Max theoretical			
SERIAL	*1*	*1*	*1*	*1*
OMP T-1	*0.7940952389*	*0.8055665648*	*0.802052811*	*1*
OMP T-2	*0.7537714488*	*0.7804586344*	*0.764061795*	*1*
OMP T-4	*0.6958434224*	*0.7482576346*	*0.7479760887*	*1*
OMP T-8	*0.1753995906*	*0.3556493228*	*0.3607331196*	*0.5*

Table 12: 100000 steps efficiency results