

Introduction

Objective

This project aims to solve the *Heat Diffussion Equation* problem by means of MPI, OpenMP and CUDA programming models. Specifically, the main aim is to parallelize the given problem and analyse the scalability of the proposed solutions and the effects of design decisions according to the characteristics of the problem.

Material to Deliver

You should deliver the source code of different sections of the project: “*OpenMP program*”, “*Hybrid program: OpenMP and MPI*” and “*CUDA program*”; according to the deadlines given below. The code will be delivered using a *github repository*. The code should contain explanations of the main decisions and library functions used to parallelize the problem. For each delivery, you should also do a brief PDF report (maximum 7 pages) with the following items:

- I. **Main decisions taken to parallelize the problem:** You should explain how you have parallelized the problem from the serial version.
- II. **Scalability of the Program:** You should show the *Execution Time*, together with the *Speedup and Efficiency* for different number of processes/cores and problem size. It is better if you use figures instead of tables.
- III. **Discussion of the obtained Results:** You should give an explanation of the results for the main metrics (*Speedup and Efficiency*) and the reasons for the obtained results.

Software and delivery Instructions

In order to do the project, you need to have an user account in the EPS teaching cluster (*moore.udl.cat*) and install in your local computer a *ssh* client to connect to the cluster. If you connect from outside of the UdL network, you need a VPN connection. In order to do this, you need to install a FortiClient in your laptop and the *Authenticator* app in your mobile for a 2FA authentication process. A manual with the instructions to install both can be found in the folder “/Resources/VPN” of the virtual campus.

In the cluster, you will have all the additional software that you need to do the project.

Likewise, to develop and test the code on your local computer, you will need to install any software that implements OpenMP, MPI and CUDA. For MPI, you can use whatever you want, but we recommend the open software MPICH2, which is portable and very popular. You can download and find information about its installation and its use at the following address:

<http://www.mcs.anl.gov/research/projects/mpich2/>

To run the serial code for this practice, there are no special requirements. You can find the source code in the front-end of the *moore.udl.cat* cluster, folder “/share/apps/files/heat/code” and also in the Project folder that you can find in Resources section of the virtual campus (/Resources/Project_HPC).

You should deliver reports throughout the Sakai virtual campus, by means of the corresponding *Activity Section*. A **github** reference to the code must be included in the corresponding report. **All the activities can be done by groups of two students.**

Deadlines for the deliveries and Percentage of the final mark:

- Section 1 (OpenMP Program): 13th of April (25%)
- Section 2 (Hybrid Program): 11th of May (35%)
- Section 3 (CUDA program): 8th of June (30%)

Note:

In order to make the report understandable and easily evaluated or reviewed by any reader, you must follow the rules of good practices to prepare a report. The report must contain the title of the activity and the authors. Use the editing techniques properly: tables, figures, cross-references, table of contents, etc. The report should be complete, justifying the design decisions, making a proper analysis of the performance and the comparison of the effects of different alternatives to reach the more appropriate decision. You can insert some code pieces for helping to explain your solutions. The figures must be correctly formatted considering aspects such as titles, axis labels, legend, type of lines, symbols, colours, etc. The text in the figures, labels and axis must be legible. The axis must be correctly dimensioned. The type of chart must be selected correctly depending on that you want to explain. Figures must be accompanied by a brief explanation.

Some common mistakes that you should avoid:

- Vague explanations without justification.
- Explanations of the implementation without justify the structure and the decisions you took.
- Lots of tables with data. You have to identify the most important data and summarize your explanations. Be concrete. Add value with your contributions.
- The figures are not properly created, bad axis ranges, no labels, blurry, or added as screenshot.
- The figures are not representative, lots of figures that can be summarized, etc.

Description of the Problem

Most problems in parallel computing require communication among tasks and a number of common problems require communication with neighbor tasks. This is the case of the *heat Diffusion equation*.

The heat diffusion equation describes the temperature change over time, given initial temperature distribution and boundary conditions. This equation is as follows:

$$\frac{\partial \mu}{\partial t} = \alpha \left(\frac{\partial^2 \mu}{\partial x^2} + \frac{\partial^2 \mu}{\partial y^2} \right);$$

where:

- $\mu = \mu(x, y, t)$ is the temperature at the point (x, y) at the instant t and
- α is the thermal diffusivity of the material.

In two spatial dimension, the solution of the heat equation represents the temperature μ (at any position x, y and any time t) in a surface. Because the rate at which heat flows through the surface depends on the material that makes up the surface, the constant α which is related to the thermal diffusivity of the material is included in the heat equation.

To solve this equation, the spatial domain can be discretized into a grid and use finite differences:

- Time derivate: $\frac{\partial \mu}{\partial t} \approx \frac{(\mu_{xy}^{n+1} - \mu_{xy}^n)}{\Delta t}$
- Spatial derivates:
 - $\frac{\partial^2 \mu}{\partial x^2} \approx \frac{(\mu_{x+1y}^n - 2\mu_{xy}^n + \mu_{x-1y}^n)}{\Delta x^2}$
 - $\frac{\partial^2 \mu}{\partial y^2} \approx \frac{(\mu_{xy+1}^n - 2\mu_{xy}^n + \mu_{xy-1}^n)}{\Delta y^2}$

Thus, the grid update rule on each time step becomes:

$$\mu_{xy}^{n+1} = \mu_{xy}^n + r \cdot (\mu_{x+1y}^n - 2\mu_{xy}^n + \mu_{x-1y}^n) + r \cdot (\mu_{xy+1}^n - 2\mu_{xy}^n + \mu_{xy-1}^n),$$

where $r = \frac{\alpha \Delta t}{\Delta x^2}$ (Assuming $\Delta x = \Delta y$).

As we can see in Figure 1, the calculation of an element is dependent upon neighbor element values.

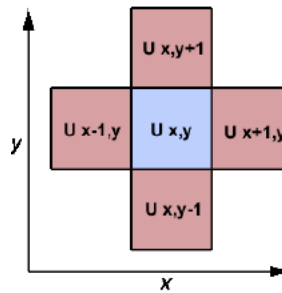


Figure 1. Dependency upon neighbor element values.

In order to calculate these values, we must take into account the following requirements:

- The element of a 2-dimensional array represents the temperature at point on the square.
- The initial temperature is zero on the boundaries and the **heat source is at both diagonals of the square**.
- The boundary temperature is held at zero.
- A time stepping algorithm is used.

Serial Code

The main function of the serial code (*heat_serial.c*) is shown in the Figure 2.

```

139 // Main function
140 int main(int argc, char *argv[]) {
141     clock_t time_begin, time_end;
142     char car;
143     double r; // constant of the heat equation
144     int nx,ny; // Grid size in x-direction and y-direction
145     int steps; // Number of time steps
146     //double DT;
147     if (argc!=4)
148     {
149         printf("Command line wrong\n");
150         printf("Command line should be: heat_serial size steps name_output_file.bmp. \n");
151         printf("Try again!!!!\n");
152         return 1;
153     }
154     nx=ny=atoi(argv[1]);
155     r= ALPHA * DT / (DX * DY);
156     steps=atoi(argv[2]);
157     time_begin=clock();
158     // Allocate memory for the grid
159     double *grid = (double *)calloc(nx * ny, sizeof(double));
160     double *new_grid = (double *)calloc(nx * ny, sizeof(double));
161
162     // Initialize the grid
163     initialize_grid(grid, nx, ny, T);
164
165     // Solve heat equation
166     solve_heat_equation(grid,new_grid,steps,r,nx, ny);
167     // Write grid into a bmp file
168     FILE *file = fopen(argv[3], "wb");
169     if (!file) {
170         printf("Error opening the output file.\n");
171         return 1;
172     }
173
174     write_bmp_header(file, nx, ny);
175     write_grid(file,grid,nx,ny);
176
177     fclose(file);
178     //Function to visualize the values of the temperature. Use only for debugging
179     // print_grid(grid, nx, ny);
180     // Free allocated memory
181     free(grid);
182     free(new_grid);
183     time_end=clock();
184     printf("The Execution Time=%fs with a matrix size of %dx%d and %d steps\n", (time_end-time_begin)/(double)CLOCKS_PER_SEC, nx, nx, steps);
185     return 0;
186 }

```

Figure 2. Main function of the *heat_serial* program.

The key features of the serial code are the following:

1. Arguments: The heat equation serial program has three different arguments:

heat_serial nx steps name_output

- *nx*= Size of the quadratic grid: Given that the number of rows and columns are equal, only the number of rows/columns is needed.
- *steps*=Time steps: This is the number of time steps of the equation.
- *Name of the output file*: This is the name of the “bmp” file, where the heat map is stored.

An example of execution could be: *heat_serial 1000 10000 output.bmp*.

2. Constants: The program defines the constants shown into the Figure 3.

```

1 #define BMP_HEADER_SIZE 54
2
3 #define ALPHA 0.01 //Thermal diffusivity
4 #define L 0.2 // Length (m) of the square domain
5 #define DX 0.02 // grid spacing in x-direction
6 #define DY 0.02 // grid spacing in y-direction
7 #define DT 0.0005 // Time step
8 #define T 1500 //Temperature on °k of the heat source
    
```

Figure 3. Constants of the program.

3. Grid Initialization:

- The 2D grid is initialized to zeros, with a heat source (1500.0°k) at both diagonals of the grid. The heat source initialization in a grid 10x10 is shown in Figure 4.

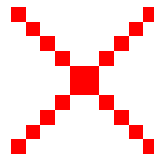


Figure 4. Heat source initialization with a grid 10x10 (*heat_serial 10 1 output.bmp*).

4. Heat equation computation:

- Each grid point is updated using the finite difference method, based on its neighbours.
- The simulation progresses through the specified number of steps (steps), which is an argument of the program. The time step is a constant of the program.
- Two grids (*grid* and *new_grid*) are used to avoid overwriting during updates. At each step, they are swapped.

5. Visualization:

- The final output of the equation can be visualized by means of a colour map stored into the “*output.bmp*” file. This output is

implemented by means of three functions: “*write_bmp_header*”, “*get_color*” and “*write_grid*”. The distribution of the colours in relation to the temperature is as follows:

Temperature (°k)	Color
≥ 500	Red
≥ 100	Orange
≥ 50	Lilac
≥ 25	Yellow
≥ 1	Blue
≥ 0.1	Cyan
< 0.1	White

In addition, there is an optional function “*print_grid*” to visualize the values of the grid. This option can be used for debugging and verifying the output of your parallel programs.

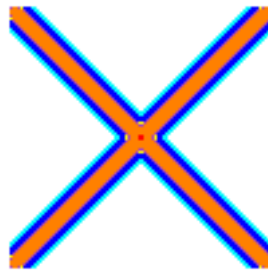


Figure 5. Output of *heat_serial* with a matrix 100x100 and 100steps (*heat_serial* 100 100 output.bmp).

The compilation and running is as follows:

1. Compile:

```
./gcc heat_serial.c -o heat_serial -lm
```

2. Run:

```
./heat_serial nx steps output.bmp
```

Remember that is **totally forbidden to execute the code in the front-end** of the cluster. Doing this you can collapse the server or produce a server downfall. By this, it is important to use the front-end queuing system even for serial programs.

For executing the serial code, you can use the same script used with the serial NAS parallel benchmarks together with the *qsub* command.

In order to have serial execution times as a reference for the future parallelization

versions, you should calculate the execution time of the *heat_serial* program for the arguments given on Table 1.

Matrix Size	Steps			
	100	1000	10000	100000
100x100				
1000x1000				
2000x2000				

Table 1. Execution times of the *heat_serial*.

An example of the output with a matrix of 100x100 and 100 steps is shown in Figure 5.

For executing the OpenMP and MPI implementation, you must use scripts provided into the Benchmarking activity.

Project Statement

Delivery 1: Implementation with OpenMP (Deadline 13th of April)

In order to obtain the best performance of an application implemented with a parallel programming model, firstly, we should start by optimizing the application at node-level. According to this, the first delivery will focus on studying the operation of the sequential version, analysing the pieces of code liable to be parallelized following a work decomposition model. Next, we will apply the OpenMP directives to parallelize the corresponding code according to the hardware and the suitable work decomposition model at node level.

When developing this version of the application, it is necessary to take into account the dependencies and their synchronization needs.

In this section you should elaborate a report discussing about the following points:

- Analyse the serial code and detect the code susceptible to be parallelized.
- Justify the strategy used to parallelize the serial code. You should point out how has solved the data dependencies.
- Analyse if it is possible to parallelize the output and justify the answer.
- If you have tested different strategies, you can describe briefly each of them and justify the reason of your choice. It is not needed to test for all the test-bed of data.
- Analyse scalability and speedup in relation to the number of threads and size problem. Consider on your discussion the effects of the dependences between data, in case it was needed, in your solution.

In order to measure the time in the OpenMP programs, you must use the *omp_get_wtime()* function, as Figure 6 shows.

```
double start_time, elapsed_time;
int x=100,y=25,z;
start_time=omp_get_wtime();
z=x*y;
elapsed_time=omp_get_wtime()-start_time;
```

Figure 6. Example of `omp_get_wtime()` function.

Delivery 2: Hybrid Implementation with OpenMP-MPI (Deadline 11th of May)

In this activity, we should take as starting point the previous OpenMP implementation. Thus, we will focus on the implementation using the MPI programming model. It means that we should analyse which is the best way to decompose the data between nodes. You can choose to develop one of the following two different versions, which work as follows:

a) Static mapping of tasks

It consists in dividing the region into a fixed number of fragments, which are processed in parallel by different processors/cores. Note that the allocation is done at the beginning of the execution of the program and cannot be modified. It can be classified into quadrants, rows, columns, etc.

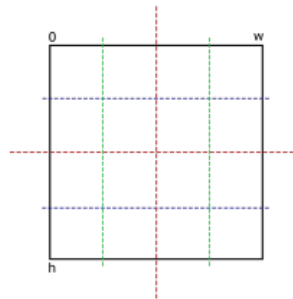


Figure 7. Static decomposition.

b) Dynamic mapping of tasks

In this case, the fragments will be mapped to different processors/cores as soon as they finish with the previous calculations.

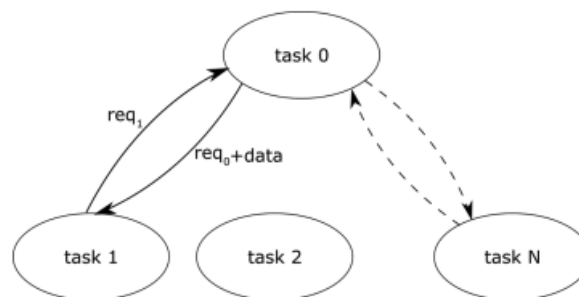


Figure 8. Dynamic decomposition.

For the chosen strategy, you must to analyse the performance of each implementation at the following points:

- Speedup compared to the number of nodes/cores.
- Identify if there is imbalanced.
- Quantify the additional cost (overhead) of parallelization.

You can optionally develop both strategies. Obviously, we will assess it positively. In such a case, explain the strengths and weaknesses of each implementation and discuss some scenarios where you think it would be more convenient to use and why.

In order to measure the time in the MPI programs, we recommend using the `MPI_Wtime` function, as the Figure 9 shows.

```
float x, y, z;  
double start, elapsed;  
start = MPI_Wtime();  
z=x*y;  
elapsed = MPI_Wtime() - start;
```

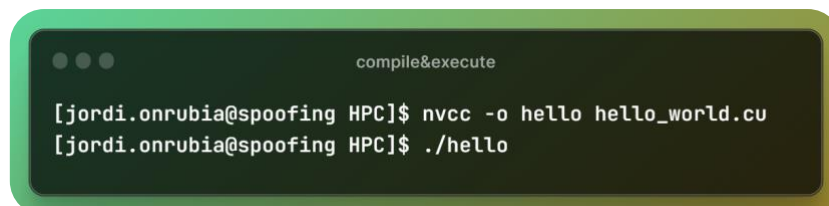
Figure 9. Example of `MPI_wtime()` function.

Delivery 3: CUDA implementation (Deadline 8th of June)

In this activity, you should propose a new implementation using CUDA C++, you should try to use different Grid and Thread sizes configurations as well as different image sizes. Compare the performance and scalability of the problem, and compare it as far as possible, with the results obtained in the previous deliveries.

At the end of the document, write a brief explanation about CPU computing vs GPU, this explanation must be about your experience, how hard is the programming, cost-benefit...

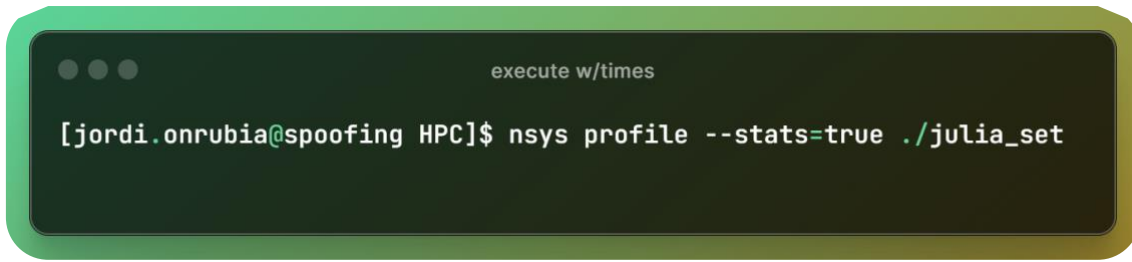
Compile and Execute:



```
compile&execute  
[jordi.onrubia@spoofing HPC]$ nvcc -o hello hello_world.cu  
[jordi.onrubia@spoofing HPC]$ ./hello
```

Figure 10. Example of compiling and executing with CUDA.

Get Times:



```
execute w/times  
[jordi.onrubia@spoofing HPC]$ nsys profile --stats=true ./julia_set
```

Figure 11. Example of how to obtain execution time with CUDA.

You can also use the classic way of computing the time = end_time-start_time, with the time library