Documentation for MPI Heat Diffusion Simulation

Camilla Roselli

Contents

1 Introduction

This document provides a detailed explanation of the parallel implementation of a 2D heat transfer simulation implemented using the MPI (Message Passing Interface) framework, which is a standardized and portable framework designed to facilitate communication and coordination among processes in parallel computing environments. This review contains an explenation of the code written in the file `mpi_heat_diffusion.c` and the obtained results.

2 Program Overview

The program calculates the heat distribution on a 2D grid over a number of time steps and it distributes the computational workload across multiple processors via domain decomposition for efficient parallel computation. The main concept is that the tasks are divided into two different groups: the master, which has rank equal to zero, and the workers, that have rank different from zero. The master inizializes the data, distributes them among the other processes, it performs the heat evolution calculation on its portion of data and finally it collects all the data sent to him by the workers. The workers receve data from the master, perform theire part of the computation and than they send the results to the master. In this implementation the master also acts as a worker, but performs more actions.

3 Main

In the `main` functions different parts can be identified.
At the very beginning, the number of arguments, passed to the code via the command line, is checked by all the tasks. The passed arguments are:

- Number of rows in the global grid `rows`.
- Number of columns in the global grid `cols`.
- Specific heat capacity in the x direction `cx`.
- Specific heat capacity in the y direction `cy`.
- Number of time steps over which the grid is expected to evolve `nts`.

The program stops if the number of command-line arguments is different from five.

After that, the MPI environment is set up. `MPI_Init(&argc, &argv)` initializes the MPI environment, `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` determines the rank of each prosses: each process gets a unique rank, which identifies it in the parallel computation, and `MPI_Comm_size(MPI_COMM_WORLD, &numtasks)` tells each process how many other processes are participating in the computation.

Later on, the pointers `u`, `local_u1`, `local_u2`, are declared to `float` and initialized to `NULL`. These pointers point to the grids used for the calculation of the heat. `u` is the global grid with the number of rows and columns passed via the command line; it contains the initial temperature values and will contain the updated final temperature values obtained after the time iteration. Whereas `local_u1` and `local_u2` are grids defined for each task with the same number of colums of `u` but with a variable number of rows, depending on the number of involved procesor and the number of rows passed by argument.

Hereafter, we can distinguish the actions performed by the master (e.g. domain definition, domain decomposition and data collection), and the actions perfomed by the workers (e.g. Time iteration).

3.1 Domain definition

The master dynamically allocates the global `u` grid using the function `allocate_matrix`, initializes the data with the function `inidat` and prints them on file `initial.dat` using the function `prtdat`.

3.2 Domain decomposition

The `u` grid initialized by the master is divided into chunks, and each process is assigned a subset of rows. The master distributes the data using the MPI collective comunication routine `MPI_Scatterv`, which distributes distinct messages of variable size, from a single source task (in this case the master) to each task. Calling the same function the workers receve the data from the master.
`MPI_Scatterv(u, sendcounts, offsets, MPI_FLOAT, local_u1, local_rows  cols, MPI_FLOAT, 0, MPI_COMM_WORLD)`

This routine takes nine arguments:

- `u` is the address of the send buffer on the master process.
- `sendounts` is an array specifying the number of elements to send to each process. Each element of this array corresponds to the number of elements to be sent to a specific task in the communicator.
- `offsets` is an array specifying the displacement in the send buffer. Each element of this array contains the offset of each task involved in the communication.
- `MPI_FLOAT` is the datatype of the element in `u`.
- `local_u1` is the address of the receive buffer. Each process will receive its own data portion.
- `local_rows  cols` is the number of elements expected to be received.
- `MPI_FLOAT` is the datatype of the element in `local_u1`.
- `0` is the rank of the process that performs the scatter operation.
- `MPI_COMM_WORLD` is the communicator that defines the group of processes involved in the scatter.

```

    /*Calculate sendcounts*/
    sendcounts = malloc(numtasks * sizeof(int));
    remainder = rows % numtasks;
    minimum_rows = ((rows - remainder)/numtasks);

    for(int i = 0; i< numtasks; i++){
      chunks_dimension[i] = minimum_rows;
      }
    for(int i = 0; i< remainder; i++){
      chunks_dimension[i] = chunks_dimension[i]+1;
      }
    for(int i = 0; i< numtasks; i++){
      sendcounts[i]= chunks_dimension[i] * cols;
      }
```

`sendcounts` is dynamically allocated and each element `sendcounts[i]` stores the total number of data assigned to the task with rank *i*. In order to compute `sendcounts` differents steps are perfomerd. First of all the minimun number of rows (`minimum_rows`) each task will handle is calculated by dividing the total number of rows evenly among all tasks. The minimum number of rows is assigned to the array `chunks_dimension` of which elements contain the number of rows of each task. However, sometimes the rows do not divide evenly and this is why the `remainder` is considered; the remaining rows are distributed one by one to the tasks, and the elements `chunks_dimension[i]` are updated. This procedure allows the workload to remain as balanced as possible. At this point, the number of rows owned by the task with rank *i* is the integer `local_rows` that is defined as the element `chunks_dimension[i]`.
In the end `sendcounts[i]` is obtained multipling the number of rows assigned to each task `chunks_dimension[i]` by the number of columns.

```c
    /*Calculate offsets*/
    offsets = malloc(numtasks * sizeof(int));
    offsets[0] = 0;
    for(int i = 1; i<numtasks;i++){
    offsets[i] = offsets[i-1] + sendcounts[i-1];
    }
```

`offsets` is dynamically allocated and each element `offsets[i]` stores the starting position of the chunk of data assigned to the task with rank *i*. The offset of the first task (rank 0) is set to zero, whereas the other offsets are calculated by taking the starting point of the previous task and adding the size of the data chunk assigned to that previous task.

3.3 Heat computation

This part is the core of the heat diffusion simulation and it is perfomed by all the tasks and the master acts as a worker. To calculate the heat diffusion on the two-dimensional grid is used a `for` loop that iterates over the number of time steps `nts` passed as command-line argument. Because the total grid is distributed across multiple processors, as described in section 3.2, communication between neighboring processes is necessary to maintain consistency at the boundaries. For this reason in the loop all the tasks call the collective communication functions `send_forwards` and `send_backwards`. `send_forwards` sends the bottom row of the local grid to the next processor and receives the top row from the previous processor, while `send_backwards` sends the top row of the local grid to the previous processor and receives the bottom row from the next processor. All tasks wait for the communication to be done before moving on, and once the informations are passed, the function `update` is called and the temperature on the grid is updated considering also the values of the grid elements of the neighboring tasks. In the end the grid pointers are swapped to prepare for the next iteration. When the loop ends, the udated data are collected by the master.

3.4 Data Collecion

To collect data after the time iteration, the MPI collective comunication routine `MPI_Gatherv` is used. This function acts in the opposite way with respect to `MPI_Scatterv`: it gathers distinct messages, of different size, from each task to a single destination task, in this case the master.
`MPI_Gatherv(local_u1, local_rows  cols, MPI_FLOAT, u, sendcounts, offsets, MPI_FLOAT, 0, MPI_COMM_WORLD)`

This routine takes nine arguments:

- `local_u1` is the address of the send buffer on each process.
- `local_rows * cols` is the number of sent elements from each process.
- `MPI_FLOAT` is the datatype of the element in `local_u1`.
- `u` is the address of the receive buffer where the receved elements will be stored on the root process.
- `recevcounts` is the array in which each element specifies the number of elements that the roots expects to receve from each process. In this case it is called `sendcounts` because is the same array already used for data distribution.
- `offsets` is an array specifying the displacement in the receve buffer.
- `MPI_FLOAT` is the datatype of the receved element.
- `0` is the rank of the process that performs the ghater operation.

- MPI_COMM_WORLD is the communicator that defines the group of processes involved in the ghater.

The master calls this function to collect the data from the workers, while the workers call this function to send the data to the master. After data collection the master prints the updated temperature on the file `final.dat` calling the function `prtdat`.

4 Subroutines

In this section all the funtions mentioned in the previous ones are explained in details.

4.1 inidat()

This function initializes the elements of a 2D array stored as a 1D array. It takes three arguments:

- `nx` is the number of rows in the conceptual 2D matrix.
- `ny` is the number of columns in the conceptual 2D matrix.
- `u` is a pointer to a 1D array that will hold the data of the 2D matrix.

The matrix is stored in a single contiguous block of memory, and the function will initialize its elements based on the indices i and j as if it were a 2D array. The function uses two nested `for` loops to iterate over the rows and columns of the matrix, and calculates an index in the 1D array `u` that corresponds to the element in row i and column j in the conceptual 2D matrix.

```c
void inidat(int nx, int ny, float *u) {
    for (int i = 0; i < nx; i++) {
        for (int j = 0; j < ny; j++) {
            u[i * ny + j] = (float)(i * (nx - i - 1) * j * (ny - j - 1)
                );
        }
    }
}
```

4.2 prtdat()

This function writes the contents of a 2D grid (stored as a 1D array) into a file in a formatted way. It takes four arguments:

- `nx` is the number of rows of the grid.
- `ny` is the number of columns of the grid.
- `u` is the pointer to a 1D array that represents the grid.
- `fname` is the name of the file where the grid should be written.

The function creates the file in write-mode, uses two nested loops to access the elements in the 1D array and it writes them in the file. The data are floating-point with 3 decimal places, each value is padded to occupy 8 characters, and they are printed in a matrix-like format.

```c
void prtdat(int nx, int ny, float *u, char *fnam) {
    FILE *fp = fopen(fnam, "w");
    for (int i = 0; i < nx; i++) {
        for (int j = 0; j < ny; j++) {
            fprintf(fp, "%8.3f", u[i * ny + j]);
```

```c
                if (j != ny - 1) fprintf(fp, " ");
            }
            fprintf(fp, "\n");
        }
        fclose(fp);
}
```

4.3 send_forwards() and send_backwards()

These functions handle data communication between neighboring processes in a distributed computing setup using MPI. They allow processes to exchange rows of data in both forward and backward directions.

```c
void send_forwards(int rank, int numtasks, int local_rows, int cols,
    float *local_u, float *prv_msg, float *flw_msg) {
     if (rank != numtasks - 1) {
         MPI_Send(&local_u[(local_rows - 1) * cols], cols, MPI_FLOAT,
            rank + 1, 0, MPI_COMM_WORLD);
     }
     if (rank != 0) {
         MPI_Recv(prv_msg, cols, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
     }
}
```

`void send_forwards` sends the last row of data from a process to the next higher-ranked process, if such a process exists, and receives the last row of data from the previous lower-ranked process, if such a process exists. More in detail, if the process is not the last, it sends the last row of its local data to the next process, calling the routine `MPI_Send`. If the process is not the first, it receves the last row of the previus process and store the data in the array `prv_msg`, calling the routine `MPI_Recv`.

```c
void send_backwards(int rank, int numtasks, int local_rows, int cols,
    float *local_u, float *prv_msg, float *flw_msg) {
     if (rank != 0) {
         MPI_Send(&local_u[0], cols, MPI_FLOAT, rank - 1, 1,
            MPI_COMM_WORLD);
     }
     if (rank != numtasks - 1) {
         MPI_Recv(flw_msg, cols, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
     }
}
```

`void send_backwards` sends the first row of data from a process to the previous lower-ranked process, if such a process exists, and receives the first row of data from the next higher-ranked process, if such a process exists. More in detail, if the process is not the first, it sends the first row of its local data to the previus process, calling the routine `MPI_Send`. If the process is not the last, it receves the first row of the next process and store the data in the array `flw_msg`, calling the routine `MPI_Recv`.

`MPI_Send` takes six arguments:

- `&local_u[i]` is the starting address of the data to send.
- `cols` is the number of elements to send.
- `MPI_FLOAT` is the data type of elements to send.
- `rank` $\pm$ `1` is the rank of the destination process.
- `0` is a tag to identify the message.

- MPI_COMM_WORLD is the communicator, which defines the group of processes that participate in communication.

MPI_Recv takes seven arguments:

- ...msg is the starting address of the buffer where the received data will be stored (e.g., prv_msg or flw_msg)
- cols is the number of elements to receve.
- MPI_FLOAT is the data type of elements to receive.
- rank is the rank of the source process.
- 0 A tag that matches the specific message.
- MPI_COMM_WORLD is the communicator, which defines the group of processes that participate in communication.
- MPI_STATUS_IGNORE is a parameter that identifies the status of the message.

4.4 update()

```
void update(int local_rows, int cols, float *u1, float *u2, int rank,
    int numtasks, float *prv_msg, float *flw_msg, float cx, float cy) {
     for (int i = 0; i < local_rows; i++) {
         for (int j = 0; j < cols; j++) {
             float up = (i == 0 && rank != 0) ? prv_msg[j] : (i > 0 ? u1
                 [(i - 1) * cols + j] : u1[i * cols + j]);
             float down = (i == local_rows - 1 && rank != numtasks - 1)
                 ? flw_msg[j] : (i < local_rows - 1 ? u1[(i + 1) * cols +
                  j] : u1[i * cols + j]);
             float left = (j == 0) ? u1[i * cols + j] : u1[i * cols + j
                 - 1];
             float right = (j == cols - 1) ? u1[i * cols + j] : u1[i *
                 cols + j + 1];
             u2[i * cols + j] = u1[i * cols + j] + cx * (left + right -
                 2.0 * u1[i * cols + j]) + cy * (up + down - 2.0 * u1[i *
                  cols + j]);
         }
     }
}
```

The heat evolution is computed using a time stepping algorithm and each element of temperature is calculeted accorging the formula:

$$u_2[i,j] = u_1[i,j] + c_x \cdot (\text{left} + \text{right} - 2 \cdot u_1[i,j]) + c_y \cdot (\text{up} + \text{down} - 2 \cdot u_1[i,j])$$

The update function calculates the next state of a grid, stored in u2, based on its current state, stored in u1. It takes the following arguments:

- local_rows is the number of rows of the grid handled by the current process.
- cols is the umber of columns.
- u1 is the current state of the grid.
- u2 is the array in which the updated grid values will be stored.
- rank is the rank of the current process in the MPI communicator.
- numtasks is the total number of processes.
- prv_msg is the buffer that contains the boundary row received from the previous process.
- flw_msg is the buffer that contains the boundary row received from the next process.

- cx and cy are the coefficients for the calculation along the x and y directions.

The function performs an update of the grid considering each element and its neighbors: above, below, to the left, and to the right, and two coefficients, cx and cy, which weigh the contributions of the cells from the horizontal and vertical directions. Because the grid is distributed across multiple processes, some neighbors might belong to adjacent processes, and the function accounts for this by incorporating data received from neighboring processes (prv_msg for the top boundary and flw_msg for the bottom boundary).
First, the function identifies neighbor values via two nested for loops. For points in the middle of the grid, neighbors come from the corresponding positions in the local grid. For points on top (if i == 0), and the process is not the first, the function takes the value from prv_msg[j]. For points below (if i == local_rows - 1) and the process is not the last, the function takes the value from flw_msg[j]. Left (if j == 0) and right (if j == cols - 1) edges are set default to the current cell value. After the calcolation the new value is stored in the corresponding position in u2.

4.5 allocate_matrix()

```c
float *allocate_matrix(int rows, int cols) {
    return malloc(rows * cols * sizeof(float));
}
```

This function is designed to dynamically allocate memory for a two-dimensional matrix of floating point numbers. The function takes as arguments the number of rows and the number of columns, and returns a pointer to a float.

5 Output Files

The program generates two files: initial.dat, that contains the initial temperature distribution and the file final.dat, containing the final temperature distribution after the time iteration. The data of these two files are plotted in the following graphs that show the initial and final temperature. In this case, the results were obtained using a matrix 100 $\times$ 100.

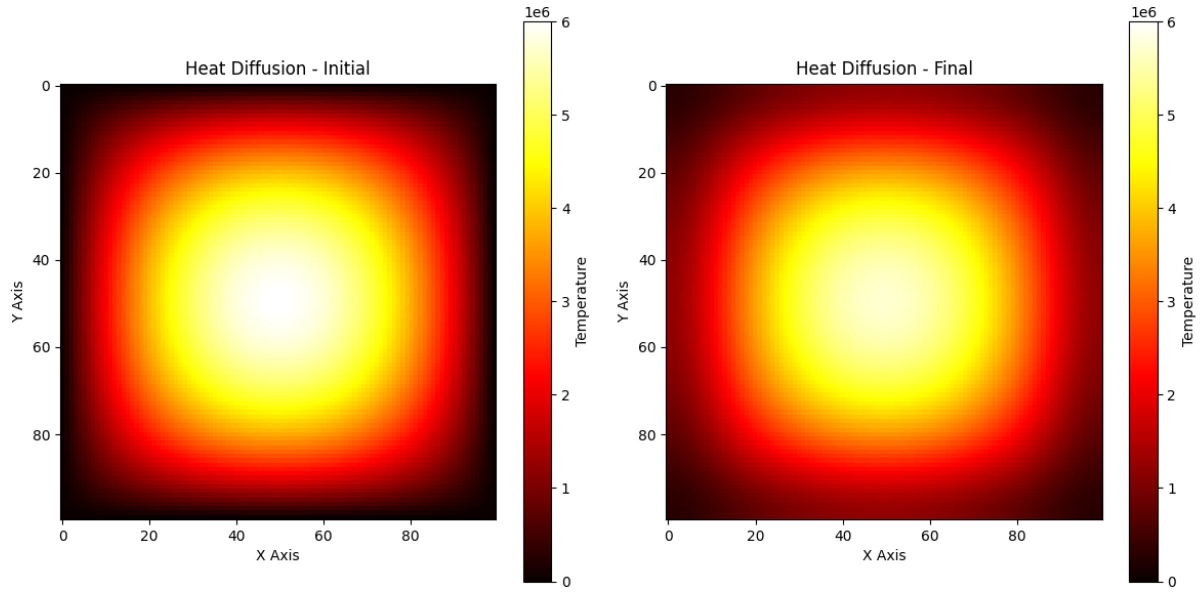

Figure 1: The plot on the left shows the initial temperature of the grid wherease the ones on the right shows the temperature after 150 time steps and using a heat coefficient equal to 0.2 in both x and y directions.

6 Docker Container for Compilation and Execution

The `Dockerfile` creates a container environment to compile and run the parallel heat diffusion simulation using MPI and then, visualize the results with a Python script. The container will create a folder called `heat_diffusion` in your home directory. The `initial.dat` and `final.dat` filed and the image `heat_diffusion_comparison.png` will be saved in this folder.

To build and run the container:

```
cd <software_project>

./run_container <4> <100> <100> <0.2> <0.01> <500>
```

The simulation parameters and the number of processors used can be changed by modifying:

```
./run_container <number_of_processes> <rows> <cols> <cx> <cy> <timesteps>
```

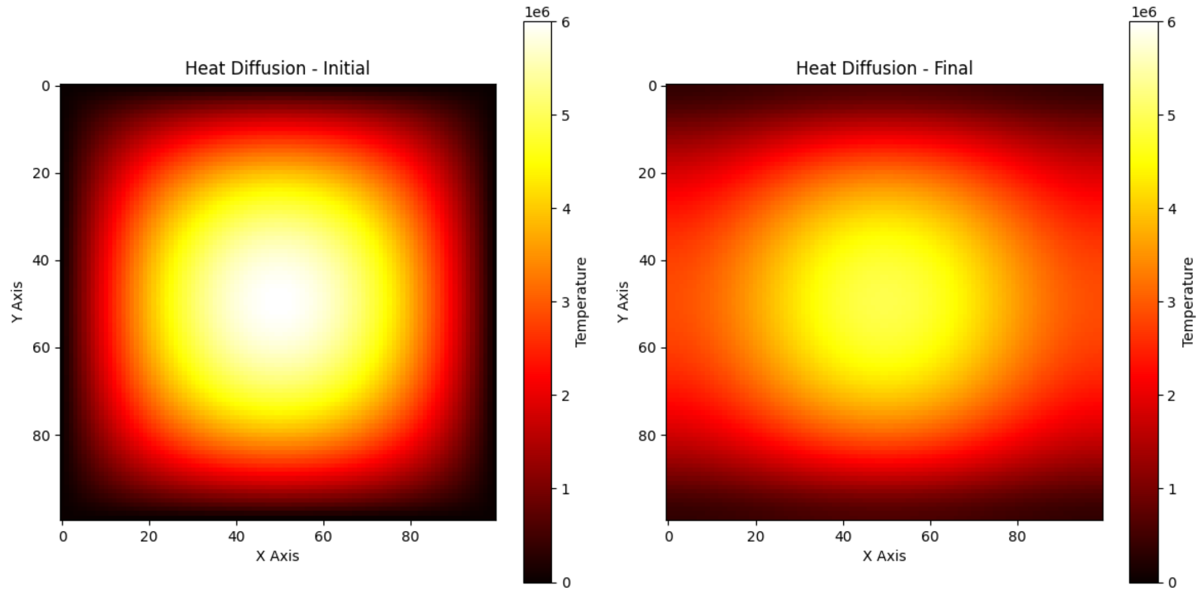

Figure 2: The plot on the left shows the initial temperature of the grid wherease the ones on the right shows the temperature after 1100 time steps and using a heat coefficient equal to 0.2 along x and equal to 0.01 along y direction.