Report - Parallelisation with MPI

Marco Di Bello

August 30, 2024

1 Introduction

1.1 Heat Diffusion Equation

The heat diffusion equation, is a fundamental partial differential equation that describes the distribution of heat (or temperature) in a given region over time. Mathematically, it is expressed as:

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{1.1.1}$$

Here, u(x,t) represents the temperature field, t is time, α is the thermal diffusivity of the material, and ∇^2 u is the Laplacian, which represents the spatial variation of temperature. The physical meaning of the heat diffusion equation lies in its ability to model the process of heat conduction, where thermal energy flows from regions of higher temperature to regions of lower temperature. The rate of diffusion is governed by the thermal diffusivity α which depends on the material's properties.

1.2 Serial implementation

The provided code is a C implementation of a serial 2D heat diffusion solver. It simulates the heat distribution in a two-dimensional grid over time based on the heat diffusion equation. The primary goal of this code is to calculate the temperature distribution in a grid where the initial temperature is highest in the center and zero at the boundaries, which remain constant throughout the simulation.

Let's see the first part of the code:

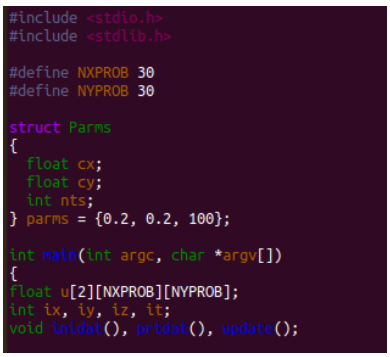

Figure 1: initial part

- `NXPROB and NYPROB:` These constants define the size of the grid (30x30 in this case).
- `Parms Structure:` This structure holds parameters for the simulation, specifically `cx`, `cy`, and `nts`, which represent the thermal diffusivity coefficients in the x and y directions and the number of time steps for the simulation, respectively.
- **inidat()** The grid is initialized with an initial temperature distribution using the inidat() function, and this initial state is saved in a file .

The program iterates through a set number of time steps (parms.nts), updating the temperature distribution in the grid at each step using the update() function.

```
/**************************************************************************
 *  subroutine update to update the temperature at every timestep
 **************************************************************************/
void update(int nx, int ny, float *u1, float* u2)
{
   int ix, iy;

   for (ix = 1; ix <= nx-2; ix++) {
      for (iy = 1; iy <= ny-2; iy++) {
         *(u2+ix*ny+iy) = *(u1+ix*ny+iy)  +
         parms.cx * (*(u1+(ix+1)*ny+iy) + *(u1+(ix-1)*ny+iy) -
         2.0 * *(u1+ix*ny+iy)) +
         parms.cy * (*(u1+ix*ny+iy+1) + *(u1+ix*ny+iy-1) -
         2.0 * *(u1+ix*ny+iy));
         }
      }
}
```

Figure 2: Update function

- `update()` The equation used in the `update` function represents a numerical approximation of the two-dimensional heat diffusion partial differential equation (PDE):

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{1.2.1}$$

The second spatial derivatives $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$ are approximated using central finite differences:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{1.2.2}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} \tag{1.2.3}$$

In the numerical equation in the code, these approximations are multiplied by the coefficients `cx` and `cy` (which correspond to $\alpha\frac{\Delta t}{\Delta x^2}$ and $\alpha\frac{\Delta t}{\Delta y^2}$ respectively) and added to the current temperature value to obtain the new temperature at the next time step. The temperature at a point (ix, iy) is updated based on the temperatures of the points (ix+1, iy), (ix-1, iy), (ix, iy+1), and (ix, iy-1).

```
/* Iterate over all timesteps and create output file */
printf("Iterating over %d time steps...\n",parms.nts);
iz = 0;
for (it = 1; it <= parms.nts; it++) {
   update(NXPROB, NYPROB, &u[iz][0][0], &u[1-iz][0][0]);
   iz = 1 - iz;
   }
printf("Done. Created output file: ");
prtdat(NXPROB, NYPROB, &u[iz][0][0], "final1.dat");
}
```

Figure 3: Calculation of new temperature

The code segment iterates over a set number of time steps to simulate heat diffusion across a 2D grid. The iteration process involves alternating between two layers of a 3D array ‘`u`‘ to store the current and updated temperature values. Here’s an explanation:

- The variable ‘`iz`‘ is initialized to ‘0‘ before the loop starts.
- At each iteration, the function ‘`update`‘ is called, which computes the new temperature distribution based on the current data stored in ‘`u[iz]`‘.
- The result of this update is stored in ‘`u[1-iz]`‘, which ensures that the new data does not overwrite the current state.
- After each time step, ‘`iz`‘ changes between ‘0‘ and ‘1‘, effectively swapping the roles of the old and new data arrays. This alternating mechanism allows the program to efficiently update and store temperature data over time without requiring additional arrays for each time step.

Finally, after all iterations are complete, the ‘`prtdat`‘ function is called to write the final temperature distribution to an output file ‘`final.dat`‘.

2 Mpi implementation

2.1 General Initial part and initial Master part

The program begins by setting up the necessary MPI (Message Passing Interface) environment and initializing variables essential for the 2D heat diffusion simulation. Here's a step-by-step explanation:

- '`MPI_Init(&argc,&argv)`': Initializes the MPI environment, setting up the parallel execution environment for the program.
- '`MPI_Comm_size(MPI_COMM_WORLD, &numtasks)`': Determines the total number of tasks available (i.e., the number of processes participating in the computation) and stores this in the variable '`numtasks`'.
- '`MPI_Comm_rank(MPI_COMM_WORLD, &taskid)`': Identifies the unique rank of each task within the MPI communicator. The rank (stored in '`taskid`') determines the role of each process, such as whether it is the master or a worker.
- '`chunksize`': This variable is calculated by dividing the total number of rows ('`NXPROB`') by the number of tasks ('`numtasks`'). It represents the number of rows each process will handle.
- '`extra`': This variable accounts for any remaining rows that cannot be evenly divided among the processes. It is computed as the remainder of '`NXPROB`' divided by '`numtasks`'.
- MPI Tags ('`tag1`', '`tag2`', '`tag_lower`', '`tag_upper`'): Tags are used to label messages sent between processes. These tags help distinguish between different types of messages, such as those carrying different portions of the grid or boundary conditions.

In the master's section:

- The initial temperature distribution of the grid is created using the function '`inidat`'.
- Boundary conditions of the grid are set: The temperature at the edges is initialized to predefined values, ensuring that they remain constant throughout the simulation.
- The initial state of the grid is saved to a file named '`initial.dat`' using the '`prtdat`' function.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define NXPROB 34
#define NYPROB 34
#define MASTER 0

struct Parms
{
  float cx;
  float cy;
  int nts;
} parms = {0.2, 0.2, 100};

int main(int argc, char *argv[])
{
 float u[2][NXPROB][NYPROB];
 float u_local[2][NXPROB][NYPROB];
 int i,j,ix, iy, iz, it,numtasks,taskid,chunksize,offset,tag1,tag2,tag_lower,tag_upper;
 int source,dest,send_chunk,recv_chunk,extra;
 void inidat(), prtdat(), update();


 // required variable for receive routines

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

 printf("Starting a parallel version of 2D heat example...\n");
 printf("Using [%d][%d] grid.\n",NXPROB, NYPROB);

 chunksize=NXPROB/numtasks;
 extra= NXPROB % numtasks;
 //useful tags
 tag1=1;
 tag2=2;
 tag_lower=10;
 tag_upper=11;

 ////////////////////////////////////////////////////////////////////////////MASTER SECTION//////////////////
 if (taskid == MASTER) {

  printf("Initializing grid and creating input file:");
  inidat(NXPROB, NYPROB, u);/*riempe griglia con valori inziali*/
  prtdat(NXPROB, NYPROB, u, "initial.dat");

/*initialize borders of the grid*/

   for (ix = 0; ix <= NXPROB-1; ix++) {
     u[1][ix][0] = u[0][ix][0];
     u[1][ix][NYPROB-1] = u[0][ix][NYPROB-1];
   }
   for (iy = 0; iy <= NYPROB-1; iy++) {
     u[1][0][iy] = u[0][0][iy];
     u[1][NXPROB-1][iy] = u[0][NXPROB-1][iy];
   }
```

Figure 4: MPI initial setting

2.2 Master sends portions of data

```
/* send each worker its portion of array*/
j = 1;
for (dest=1; dest<numtasks; dest++) {
   int send_chunk = chunksize;
   if (j <= extra ){
     send_chunk =send_chunk+1;
   }
   MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
   MPI_Send(&u[0][offset][0],send_chunk*NXPROB, MPI_FLOAT, dest,tag2,MPI_COMM_WORLD);
   //offset=offset+send_chunk;
   printf("Sent %d elements to task %d with offset= %d\n",send_chunk,dest,offset);
   offset=offset+send_chunk;
   j++;

}
```

Figure 5: Master sends data

In this section, the `Master` process is responsible for distributing portions of the grid to each worker process. The grid is divided into chunks, with each chunk representing a section of the grid that a worker process will handle. Here's a detailed breakdown:

- `offset = chunksize;`: Initializes the `offset` variable, which keeps track of the starting row for the chunk of data being sent to each worker. Initially, it is set to `chunksize`, as the `Master` process itself will handle the first chunk.
- The loop `for (dest=1; dest<numtasks; dest++)`: Iterates over all worker processes, starting from rank 1 (since rank 0 is the `Master`).
- `if (j <= extra) { send_chunk = send_chunk+1; }`: This condition handles the distribution of extra rows. If there are any extra rows that couldn't be evenly divided among all workers, one extra row is added to the chunk size for the first `extra` workers.
- `MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);`: Sends the `offset` value to each worker, which indicates the starting row for the chunk that the worker will process.
- `MPI_Send(&u[0][offset][0], send_chunk*NXPROB, MPI_FLOAT, dest, tag2, MPI_COMM_WORLD);`: Sends the actual chunk of the grid (a 2D slice of `send_chunk` rows) to each worker.
- `offset = offset + send_chunk;`: Updates the `offset` to point to the next chunk of the grid for the subsequent worker.
- `j++;`: Increments the counter `j`, which tracks how many extra rows have been distributed. Once all extra rows have been assigned, subsequent workers will receive only `chunksize` rows.

This section of code ensures that the grid is distributed evenly among the worker processes, with extra rows being distributed one by one until all have been assigned. This approach helps balance the computational load across all processes.

2.3 Worker receive portions

```
////////////////////////////////////////////////////////////WORKER SECTION*////////////////////////////////////////////////////////////



}else if(taskid>MASTER)  {

  /*initialize local grid to 0*/
 for (ix = 0; ix < NXPROB-1; ix++) {
          for (iy = 0; iy < NYPROB; iy++) {
              u_local[0][ix][iy] = 0.0;
              u_local[1][ix][iy]= 0.0;
          }
 }

  /*receive data portions from master*/
 source=MASTER;
 MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 int recv_chunk = chunksize;
 if (taskid <= extra) {
    recv_chunk += 1;
 }
 MPI_Recv(&u_local[0][offset][0], recv_chunk*NXPROB, MPI_FLOAT, source, tag2, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

 printf("Receive %d elements from task %d with offset= %d\n",recv_chunk,source,offset);
```

Figure 6: Worker receives data

In this section, the worker processes (`taskid > MASTER`) are responsible for receiving their assigned portions of the grid from the `Master` process. Here's a detailed explanation:

- `if(taskid > MASTER)`: This condition ensures that the following operations are only executed by worker processes, excluding the `Master` process.
- `/* initialize local grid to 0 */`: Each worker initializes its local grid arrays `u_local[0]` and `u_local[1]` to zero. This step prepares the grid for receiving data and processing:
- `/* receive data portions from master */`: This section manages the reception of grid data from the `Master` process:
  - `source = MASTER;`: Identifies the source of the incoming data as the `Master` process.
  - `MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`: Receives the `offset` value from the `Master`.

The `offset` indicates the starting row of the grid portion that the worker will process.

- `int recv_chunk = chunksize;`: Initializes the size of the chunk of data that each worker expects to receive.
- `if (taskid <= extra) { recv_chunk += 1; }`: If the worker is among the first `extra` workers, it receives one additional row to handle the extra rows that could not be evenly divided.
- `MPI_Recv(&u_local[0][offset][0], recv_chunk*NXPROB, MPI_FLOAT, source, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`: This line receives the grid data from the `Master`. The amount of data received is `recv_chunk` rows, each containing `NXPROB` elements.

The use of `MPI_Recv` in this code is an example of blocking communication, meaning the worker process will wait until the data is fully received before continuing execution. This ensures that each worker receives its complete portion of the grid before starting its computations, which is crucial for maintaining the correctness of the simulation. Blocking communication simplifies synchronization between the `Master` and worker processes, as it guarantees that data is available when the worker proceeds to the next step.

2.4 Communication

2.4.1 Master Communication

```
//Master works and communicates
printf( "Iterating over %d time steps...for %d \n", parms.nts,taskid);
iz=0;
offset=0;
for (it = 1; it <= parms.nts; it++) {

   MPI_Send(&u[iz][offset+chunksize - 1][0], NYPROB, MPI_FLOAT,  1, tag_lower,MPI_COMM_WORLD);
   MPI_Recv(&u[iz][offset+chunksize][0], NYPROB, MPI_FLOAT,  1, tag_upper, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

   update(chunksize,NYPROB,&u[iz][offset+1][0],&u[1-iz][offset+1][0]);

   iz=1-iz;
   MPI_Barrier(MPI_COMM_WORLD);
  }
```

Figure 7: Master communication

In this section, the `Master` process not only works on its portion of the grid but also manages communication with the first worker process to ensure the correct handling of boundary conditions. Here's a detailed explanation:

- `iz` is initialized to 0, and it is used to alternate between the current and the previous state of the grid at each time step.
- `offset` is set to 0, indicating that the `Master` works from the beginning of the global grid (from row 0).

- `for (it = 1; it <= parms.nts; it++) {` : This loop iterates over all time steps (`parms.nts`).
- `MPI_Send(&u[iz][offset+chunksize - 1][0], NYPROB, MPI_FLOAT, 1, tag_lower, MPI_COMM_WORLD);`:
  - The `Master` sends the last row (`offset + chunksize - 1`) of its portion of the grid to the first worker process (`taskid = 1`).
  - The tag `tag_lower` is used to identify this message.
- `MPI_Recv(&u[iz][offset+chunksize][0], NYPROB, MPI_FLOAT, 1, tag_upper, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`:
  - The `Master` then receives the first row (`offset + chunksize`) of the grid portion from the first worker, which is essential for maintaining continuity between the grid portions handled by different processes.
  - The tag `tag_upper` is used to identify this incoming message.
- `update(chunksize, NYPROB, &u[iz][offset+1][0], &u[1-iz][offset+1][0]);`:
  - The `update` function is called to compute the new temperature values for the grid portion man `iz = 1 - iz;`:aged by the `Master`.
  - This update occurs on the grid rows from `offset + 1` to `offset + chunksize - 1`, leaving out the boundary rows that are managed through communication with the worker process.
  - The indices `iz` and `1-iz` alternate the usage of the current and previous states of the grid for updating the values.
- `iz = 1 - iz;`: The `iz` index is toggled between 0 and 1 after each time step to switch between the current and the new temperature grids, effectively alternating the use of the grid arrays.
- `MPI_Barrier(MPI_COMM_WORLD);`: This barrier ensures that all processes (both `Master` and workers) synchronize at the end of each time step. This synchronization is crucial for maintaining consistency across the different portions of the grid handled by each process.

So the `Master` process performs computations on its portion of the grid, covering the rows from 1 to `chunksize`. The for loop in the update function has been modified to start the temperature update from the first row of the local matrix u_local and continue until the last row. This allows for specific handling of the function when we are dealing with the update of the master's portion and the last task's portion. For the master, we need to start from offset+1, the second row of the global matrix.

2.4.2 Worker communication

```
printf("Iterating  over %d time steps... for task %d \n", parms.nts,taskid);

iz=0;
for (it = 1; it <= parms.nts; it++) {

 if (taskid > MASTER && taskid < numtasks - 1) {

   if (taskid % 2 ==0){

     MPI_Send(&u_local[iz][offset+recv_chunk - 1][0], NYPROB, MPI_FLOAT, taskid + 1, tag_lower, MPI_COMM_WORLD);
     MPI_Send(&u_local[iz][offset][0], NYPROB, MPI_FLOAT, taskid - 1, tag_upper, MPI_COMM_WORLD);


     MPI_Recv(&u_local[iz][offset-1][0], NYPROB, MPI_FLOAT, taskid - 1, tag_lower, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     MPI_Recv(&u_local[iz][offset+recv_chunk][0], NYPROB, MPI_FLOAT, taskid + 1, tag_upper, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     update(recv_chunk,NYPROB,&u_local[iz][offset][0], &u_local[1 - iz][offset][0]);

   } else if(taskid % 2 !=0 ){


     MPI_Recv(&u_local[iz][offset-1][0], NYPROB, MPI_FLOAT, taskid - 1, tag_lower, MPI_COMM_WORLD,MPI_STATUS_IGNORE );
     MPI_Recv(&u_local[iz][offset+recv_chunk][0], NYPROB, MPI_FLOAT, taskid + 1, tag_upper, MPI_COMM_WORLD,MPI_STATUS_IGNORE);


     MPI_Send(&u_local[iz][offset+recv_chunk - 1][0], NYPROB, MPI_FLOAT, taskid + 1, tag_lower, MPI_COMM_WORLD);
     MPI_Send(&u_local[iz][offset][0], NYPROB, MPI_FLOAT, taskid - 1, tag_upper, MPI_COMM_WORLD);
     update(recv_chunk,NYPROB,&u_local[iz][offset][0], &u_local[1 - iz][offset][0]);
   }



}else if (taskid == numtasks - 1) {
   if (taskid % 2 != 0){

     MPI_Recv(&u_local[iz][offset-1][0], NYPROB, MPI_FLOAT, taskid - 1, tag_lower, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     MPI_Send(&u_local[iz][offset][0], NYPROB, MPI_FLOAT, taskid - 1, tag_upper, MPI_COMM_WORLD);

     update(recv_chunk-1, NYPROB, &u_local[iz][offset][0], &u_local[1-iz][offset][0]);

   } else if(taskid % 2 ==0 ){
     MPI_Send(&u_local[iz][offset][0], NYPROB, MPI_FLOAT, taskid - 1, tag_upper, MPI_COMM_WORLD);
     MPI_Recv(&u_local[iz][offset-1][0], NYPROB, MPI_FLOAT, taskid - 1, tag_lower, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

     update(recv_chunk-1, NYPROB, &u_local[iz][offset][0], &u_local[1-iz][offset][0]);
   }
}

 iz=1-iz;
 MPI_Barrier(MPI_COMM_WORLD);

}//end for

////////////////////////////////////////////////////////////////////////////////END COMMUNICATION////////////////////////////////////////////////////////////////////////////////

printf("end communication for taskid %d \n",taskid);
```

Figure 8: Worker comunication

In this section, the worker processes handle the communication and computation for the 2D heat equation simulation, specifically focusing on the synchronization and exchange of boundary data between adjacent tasks.

The key points are as follows:

- `MPI_Send` and `MPI_Recv` operations are used to exchange boundary rows between adjacent tasks:
  - `taskid > MASTER && taskid < numtasks - 1`: Intermediate tasks (neither `MASTER` nor the last task) communicate with both the previous and next tasks.
  - `if (taskid % 2 == 0)`: Even-numbered tasks send their last row to the next task (`taskid + 1`) and their first row to the previous task (`taskid - 1`) before receiving the respective boundary rows from those tasks.
  - `else if (taskid % 2 != 0)`: Odd-numbered tasks first receive the boundary rows from the previous and next tasks, and then send their last and first rows respectively.
  - The `update` function is called after each communication round to update the temperature values in the local grid.
- `taskid == numtasks - 1`: The last task only communicates with the task before it (`taskid - 1`):
  - `if (taskid % 2 != 0)`: The last task receives the boundary row from the previous task, updates its grid, and then sends its first row back.
  - `else if(taskid % 2 == 0)`: It first sends its boundary row to the previous task and then receives the needed boundary data, followed by an update of its grid.
  - This behavior ensures that the top boundary of the last task aligns with the global grid boundary, requiring no communication for the bottom boundary.
- `MPI_Barrier(MPI_COMM_WORLD)`:
  - After each time step, all tasks synchronize using `MPI_Barrier`. This ensures that all tasks complete their communication and computation for the current time step before proceeding to the next one. This barrier is used in the same way as it is positioned in the communication between the Master and the first worker.

This section handles the necessary communication between adjacent tasks to ensure proper propagation of temperature values across the grid. The communication scheme alternates between even and odd task IDs to avoid deadlocks, for the last task's portion, the update will proceed up to `recv_chunk-1`, since the last row of the global matrix is initialized to zero.

The condition for the last task (whether it is even or odd) allows the code to run correctly with either an even or odd number of processors. This design ensures robust synchronization across all tasks, regardless of the number of processors involved.

2.5 Send and receive uptated portions

2.5.1 Sending to master new data

```
// Send results back to MASTER


MPI_Send(&offset, 1, MPI_INT, MASTER, tag1, MPI_COMM_WORLD);
int send_chunk = chunksize;
 if (taskid <= extra) {
    send_chunk += 1;
 }

MPI_Send(&u_local[1][offset][0],send_chunk*NYPROB, MPI_FLOAT, MASTER, tag2, MPI_COMM_WORLD);

printf("Send  updated %d elements to task %d with offset= %d\n",send_chunk,MASTER,offset);
} //end if principale
```

Figure 9: Workers send new data

After completing their assigned computation, each worker task sends its updated portion of the grid back to the MASTER process.

- **Offset Transmission:** The `MPI_Send` function is used to send the `offset` variable to the MASTER, indicating the starting position of the data within the global grid.
- **Determining `send_chunk`:** The variable `send_chunk` is initially set to `chunksize`. If the `taskid` is among those that received extra rows (i.e., `taskid <= extra`), `send_chunk` is incremented by one to account for this additional data.
- **Data Transmission:** The worker task then sends its portion of the updated grid data back to the MASTER using `MPI_Send`. The data is transmitted from the `u_local[1][offset][0]` array, covering `send_chunk * NYPROB` elements.

2.5.2 Receiveing from workers new data

```
//Master receive results from each worker//
j = 1;
for (source=1; source<numtasks; source++) {
   int recv_chunk = chunksize;
   if (j <= extra){
     recv_chunk =recv_chunk +1;
   }

  MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Recv(&u[1][offset][0],recv_chunk*NXPROB, MPI_FLOAT, source,tag2,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  printf("Received  %d elements from task %d with offset= %d\n",recv_chunk,source,offset);

}

prtdat(NXPROB,NYPROB,u[1],"final.dat");
```

Figure 10: Master receives new data

Master Receiving Results from Workers: After computation, the MASTER process gathers the updated grid data from each worker.

- **Determining `recv_chunk`:** For each worker, `recv_chunk` is set to `chunksize`, and is incremented by one if the worker received an extra row (i.e., `j <= extra`).
- **Receiving Data:** The MASTER uses `MPI_Recv` to first receive the `offset` and then the corresponding portion of the grid (`recv_chunk * NXPROB` elements) from each worker.
- **Final Output:** After all data has been collected, the grid is saved to `final.dat` using `prtdat`.

This step consolidates the processed grid segments from all workers into the final global grid on the MASTER. After receiving and storing the data in `final.dat`, the program calls `MPI_Finalize`. This function is used to properly shut down the MPI environment, ensuring that all processes finish their tasks and release any resources used during the communication. Once this is done, the program prints a final message indicating that it has completed, and then it exits.

3 Final results

The choice to use blocking communication between the workers was made for several practical reasons, primarily related to simplicity, implicit synchronization, and resource management. Indeed non-blocking communication(`MPI_Isend` or `MPI_Irecv`) is powerful and can enhance performance in some scenarios but it introduces additional complexity. It requires explicit

management of synchronization, using calls like `MPI_Wait` or `MPI_Test`, and there is a risk of encounter conditions or deadlocks if they are not handled correctly. Below is the grid displaying the updated data:

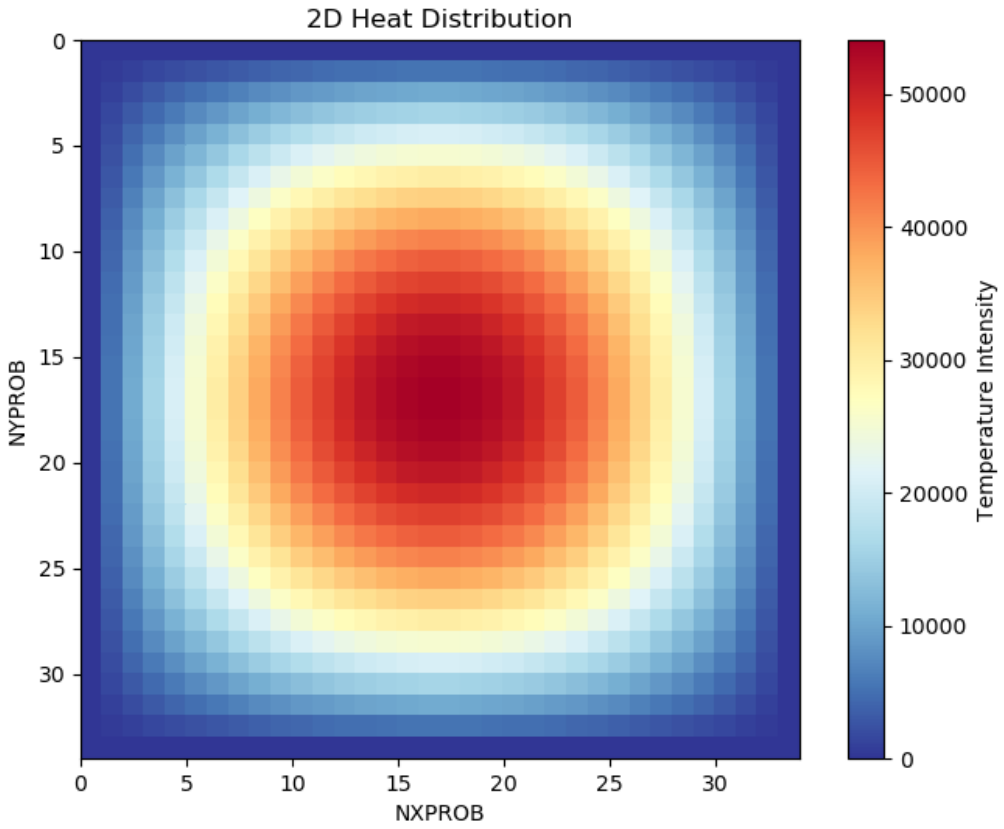

Figure 11: Updated grid

Contents