High-Performance Computing

MPI for 2D Heat Equation

Anand Kamble
Department of Scientific Computing
Florida State University

1 Introduction

In this project we are using MPI to solve the 2 dimensional heat equation by parallelizing it over multiple processes.

2 Implementation

The source code for this program is written in the file `main.cpp`. To compile the program, you will have to use the mpiCC compiler (use capital 'C' for C++ compiling) as shown in in listing below, which is available from the module `openmpi-x86_64`.

Listing 1: bash

```
mpiCC -std=c++17  src/main.cpp -o bin/test.x
```

2.1 Initializing the MPI Environment

The MPI environment is initialized using the MPI_Init function from the mpi.h header file. The rank of the process and the total number of processes are obtained using MPI_Comm_rank and MPI_Comm_size, respectively.

Listing 2: main.cpp

```
int numProcesses, rank;
MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
```

2.2 Creating a 2D Cartesian Communicator

The program creates a 2D Cartesian communicator using MPI_Dims_create and MPI_Cart_create. This step divides the processes into a 2D grid, facilitating the distribution of work among processes.

Listing 3: main.cpp

```
MPI_Dims_create(numProcesses, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm2d);
```

2.3 Decomposing the Grid

The decompose1d function is used to decompose the grid into smaller grids, which are then assigned to individual processes. This function calculates the start and end indices of the grid for each process based on the grid size and the number of processes.

Listing 4: main.cpp

```
int x0, x1, y0, y1;
int NX = 100, NY = 134;
MPI_Cart_get(comm2d, 2, dims, periods, coords);
decompose1d(NX, dims[0], coords[0], &x0, &x1);
decompose1d(NY, dims[1], coords[1], &y0, &y1);
```

2.4 Initializing the Grid Values

The initial values of the grid are set according to the boundary conditions. The top, right, and bottom boundaries are set to 1.0, while the left boundary is set to 1.0 for all points except the bottom-left corner, which is set to 0.0. The interior points are initialized to 0.0.

Listing 5: main.cpp

```
// Initialize the values to zero so that we can set the boundary conditions
for (int i = 0; i < nx; i++){
    for (int j = 0; j < ny; j++){
        u[i][j] = 0.0;
    }
}
// For the left and right boundaries using the X coordinate
if (coords[0] == 0){
    for (int j = 0; j < ny; j++){
        u[0][j] = 1.0; // Left boundary
    }
}

if (coords[0] == dims[0] - 1){
    for (int j = 0; j < ny; j++){
        u[nx - 1][j] = 1.0; // Right boundary
    }
}

// For the top and bottom boundaries using the Y coordinate
if (coords[1] == 0){
    for (int i = 0; i < nx; i++){
        u[i][0] = 0.0; // Bottom boundary
    }
}

if (coords[1] == dims[1] - 1){
    for (int i = 0; i < nx; i++){
        u[i][ny - 1] = 1.0; // Top boundary
    }
}

// Copy the initial values to u_new
for (int i = 0; i < nx; ++i){
    for (int j = 0; j < ny; ++j){
        u_new[i][j] = u[i][j];
    }
}
```

2.5 Communicating Boundary Values

The program uses MPI_Sendrecv to communicate the boundary values between neighboring processes. This ensures that each process has the necessary information to update its portion of the grid correctly.

Listing 6: main.cpp

```
MPI_Sendrecv(&u[0][ny - 2], 1, xSlice, up, 123, &u[0][0], 1, xSlice,
             down, 123, comm2d, MPI_STATUS_IGNORE);
MPI_Sendrecv(&u[0][1], 1, xSlice, down, 123, &u[0][ny - 1], 1, xSlice,
             up, 123, comm2d, MPI_STATUS_IGNORE);
MPI_Sendrecv(&u[nx - 2][0], 1, ySlice, right, 123, &u[0][0], 1, ySlice,
             left, 123, comm2d, MPI_STATUS_IGNORE);
MPI_Sendrecv(&u[1][0], 1, ySlice, left, 123, &u[nx - 1][0], 1, ySlice,
             right, 123, comm2d, MPI_STATUS_IGNORE);
```

2.6 Writing Output to files

The program measures the execution time using MPI_Wtime. The master process prints the time taken to solve the equation. Additionally, each process writes its portion of the solution to a separate text file named solution_⟨rank⟩.txt.

Listing 7: main.cpp

```
160 std::ofstream outFile;
161 std::string filename = "solution_" + std::to_string(rank) + ".txt";
162 outFile.open(filename);
163 for (int i = 0; i < nx; i++)
164 {
165     for (int j = 0; j < ny; j++)
166     {
167         outFile << u[i][j] << "␣";
168     }
169     outFile << "\n";
170 }
171 outFile.close();
```

2.7 Finalizing the MPI Environment

Finally, the MPI environment is finalized using MPI_Finalize, which releases any allocated resources and gracefully exits the program.

Listing 8: main.cpp

```
177 MPI_Type_free(&xSlice);
178 MPI_Type_free(&ySlice);
179 MPI_Finalize();
```

3 Results

The program is scaling well on multiple number of processes. But after the number of processes increase above 4 we can clearly see that the speed up we are getting is not significant, and after 8 it almost stays the same.

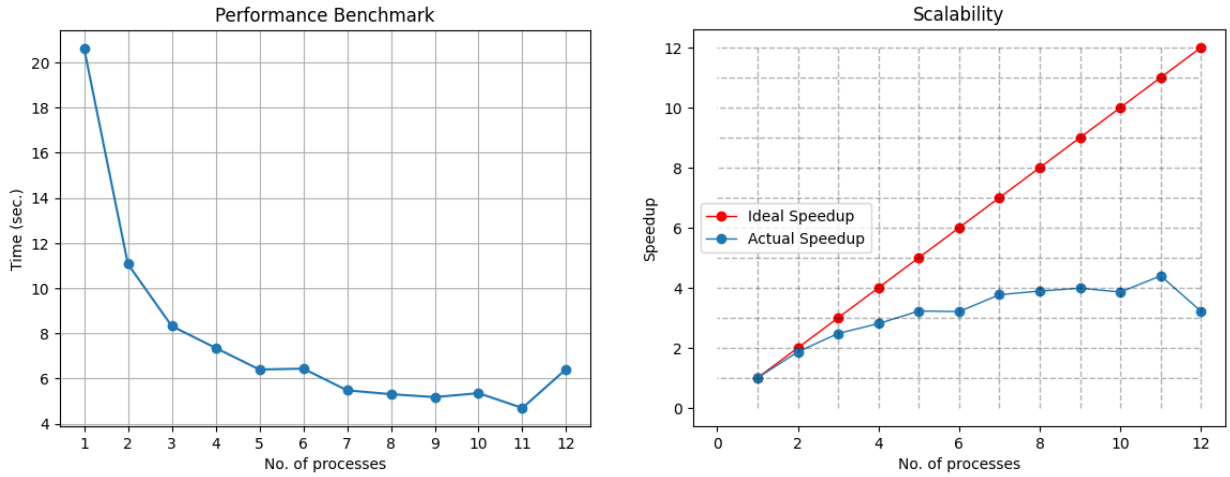
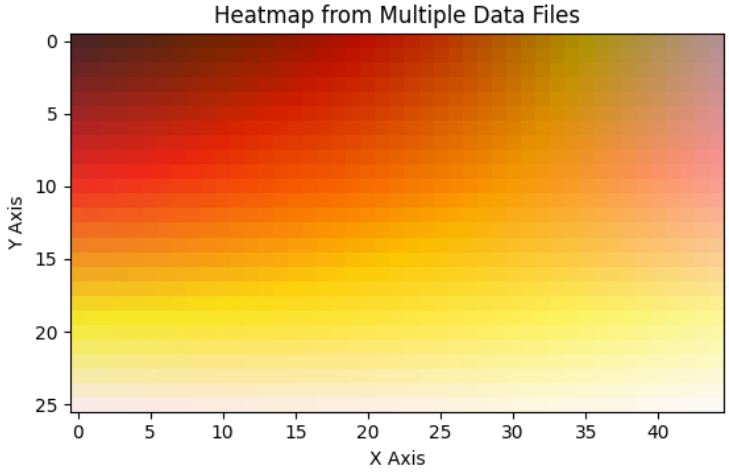


Figure 1: Timing and Scaling of the program

If we plot the output values at $T = 3$, we get the following result.



References

[1] Cecilia. *MPI_Dims_create throws error on remote machine* , https://stackoverflow.com/questions/46698735/mpi-dims-create-throws-error-on-remote-machine. Oct 11, 2017.

[2] DeinoMPI. *MPI_Dims_create* , https://mpi.deino.net/mpi_functions/MPI_Dims_create.html. 2009.

[3] G. Nervadof. *Solving 2D Heat Equation Numerically using Python* , https://levelup.gitconnected.com/solving-2d-heat-equation-numerically-using-python-3334004aa01a. Oct 13, 2020.