Parallelization of the 2D Laplace Equation Solver Using MPI

1. Introduction

This project focuses on solving the two-dimensional Laplace equation using iterative methods and parallel computing with MPI. The Laplace equation models many physical phenomena, such as heat distribution or electrostatics, making it a common benchmark for parallel performance studies.

The main goal is to start from a working sequential version of the algorithm and progressively parallelize it using MPI with different communication methods. The project evaluates the correctness and performance of each version by comparing execution times and results.

2. Original Code Explanation

The original code is a sequential implementation of a Jacobi solver for the Laplace equation on a 2D grid.

- The grid size is n x m (default: 4096×4096).
- Two matrices are used: A holds the current values, and Anew stores the new values for each iteration.
- The left and right boundaries are initialized with specific sine and exponential functions, while the top and bottom are set to zero.
- In each iteration:
    1. Anew values are calculated as the average of their four neighbors (left, right, top, bottom).
    2. The error is computed as square root of the maximum difference between A and Anew.
    3. A is updated with Anew.
- This loop continues until either the error is small enough (convergence) or a maximum number of iterations is reached (default: 100, changed to 1000 for final results).
- The program prints the error every 10 iterations and the total execution time at the end (We added this last part using MPI_Wtime()).

3. Parallelization Process

To begin parallelization, I added calls to **MPI_Init, MPI_Comm_rank,** and **MPI_Comm_size** to initialize the MPI environment and determine the number of processes and the rank of each process.

```
// Initialize MPI
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

I then continued by splitting the original matrix horizontally into N separate processes. I also made sure to not forget any rows just in case the result was affected (ensure propper load balancing, see results/fixing_load_balancing).

```
// Calculate the number of rows each process will handle, this way the result is exactly the same
// as if the matrix was divided evenly among the processes.
// The first process will handle the extra rows if n is not divisible by size
int base_rows = n / size;
int extra_rows = n % size;

int local_n;
int first_row;

if (rank < extra_rows) {
  local_n = base_rows + 1;
  first_row = rank * local_n;
} else {
  local_n = base_rows;
  first_row = rank * base_rows + extra_rows;
}

// The data is dynamically allocated
// We add rows above and below the local_n rows to handle boundaries
float **A = (float **)malloc((local_n + 2) * sizeof(float *));
float **Anew = (float **)malloc((local_n + 2) * sizeof(float *));
for (i = 0; i < local_n + 2; i++) {
  A[i] = (float *)malloc(m * sizeof(float));
  Anew[i] = (float *)malloc(m * sizeof(float));
}
```

Another change I had to implement was to properly determine the boundaries with the new divisions (so they remained as before).

```
// set boundary conditions (left and right)
for (i = 0; i < local_n + 2; i++) {
  int global_i = first_row - 1 + i;
  if (global_i >= 0 && global_i < n) { // Ensure top and bottom boundaries are 0
    A[i][0] = sinf(PI * global_i / (n - 1)); // Left boundary
    A[i][m - 1] = A[i][0] * expf(-PI); // Right boundary
  }
}
```

Afterwards, we had to actually implement the parallelism with everything set up already. To avoid blocking, I made use of non-blocking functions (**Isend** and **Irecv**). I also made a copy of the code using standard blocking functions, to compare the results.

```
// Main loop: iterate until error <= tol a maximum of iter_max iterations
while (error > tol && iter < iter_max) {
  MPI_Request requests[4];
  int req_count = 0;
  // Synchronize the boundary rows with neighboring processes
  if (rank > 0) {
    MPI_Isend(A[1], m, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(A[0], m, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD, &requests[req_count++]);
  }
  if (rank < size - 1) {
    MPI_Isend(A[local_n], m, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(A[local_n + 1], m, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
  }

  MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
```

Then, we had to adapt the rest of the code, so it still performed the Anew values as well as the local error and updating the A matrix. For the local error calculation, I also had to perform a reduction across all processes (max function).

```
// Calculate the new value for each element based on the current
// values of its neighbors.
for (i = 1; i <= local_n; i++)
  for (j = 1; j < m - 1; j++)
    Anew[i][j] = (A[i][j + 1] + A[i][j - 1] + A[i - 1][j] + A[i + 1][j]) / 4;

// Compute local_error = maximum of the square root of the absolute differences
// between the new value (Anew) and old one (A)
float local_error = 0.0f;
for (i = 1; i <= local_n; i++)
  for (j = 1; j < m - 1; j++)
    local_error = fmaxf(local_error, sqrtf(fabsf(Anew[i][j] - A[i][j])));

// Update the value of A with the values calculated into Anew
for (i = 1; i <= local_n; i++)
  for (j = 1; j < m - 1; j++)
    A[i][j] = Anew[i][j];

MPI_Allreduce(&local_error, &error, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD); // Reduce the error across all processes MPI Max
```

Lastly, we just printed the time it took for the execution with **MPI_Wtime**.

4. Results

I did all my project using the **Wilma** cluster.

If we take a good look in my results folder, I have several underlying repositories.

One of them is the standard output to make sure I was parallelizing without altering the results.

Then we have a folder I previously mentioned, fixing_load_balancing, which I used to distribute all rows equally to maintain constant results. I used a 12x12 matrix to properly see how forgetting rows affected the results.

Lastly, we have the folders of the results using 100 and 1000 iterations, and both versions, blocking and non_blocking, with 1,2,4,6,8 and 10 processes (all with a 4096x4096 matrix).

The results are equal in terms of error, which indicates a proper parallelization since time was reduced. However, the difference between the blocking and non-blocking functions are barely noticeable.

That can be explained quite easily: in the non-blocking version, there are no operations that can be done without completing the message passing, which essentially means that my versions are both equal, one being blocked by Send() and Recv() and the other with the Wait().

The results I will showcase right now are the non-blocking results for 1000 iterations, with 1, 2, 4 and 8 processes. To see the rest, you can just see the results folder in my GitHub:

https:/github.com/PolPuigPuy/MPI_Project_PolPuig.git

Results (Wilma Cluster, 1000 iterations, 4096x4096)				
Nº of processors	K = 1	K = 2	K = 4	K = 8
Time	159.5584	89.7028	46.2645	30.5769
SpeedUp	1	1.7787	3.4488	5.2183
Efficiency	1	0.8894	0.8622	0.6523

The results show clear performance gains when using multiple processes, with substantial reductions in execution time. However, speedup isn't doubling and efficiency drops as the number of processes increases, which reflects the growing impact of communication overhead and coordination between processes.