COMP 429/529: Project 3

Muhammed Esad Simitcioglu

January 21, 2024

In Part 1, I parallelized the simulator using MPI by supporting one-dimensional processor geometry in this part under 1D geometry conditions (only -y is set).
In Part 2, I introduced OpenMP to enhance the processing, achieving a hybrid programming approach. As a result, each process now utilizes local threads, resulting in improved speed and efficiency.
In Part 3, I parallelized the simulator using MPI by supporting two-dimensional processor geometry in this part under 2D geometry conditions (-x and -y is set).

1 Part 1: MPI Parallelization with 1D Geometry

In the initial phase of this assignment, I created a parallelized simulator for 1D geometry. I implemented row-partitioning to ensure contiguous memory for ghost cells. The root process first collects data from the terminal, and then I broadcast all the parameters to the other processors.

```
 if (rank == 0)
{
  cmdLine(argc, argv, T, n, px, py, plot_freq, no_comm,
  num_threads);
}

MPI_Bcast(&T, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&px, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&py, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&plot_freq, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&no_comm, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&num_threads, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

I instantiated and initialized the input matrices E, E_prev, and R and then broadcasted them to the other processors.

```
if (rank == 0)
{

  cout << "RP = " << rp << endl;
  cout << "dte = " << dte << endl;
  cout << "dtr = " << dtr << endl;
  cout << "dt = " << dt << endl;
  cout << "alpha = " << alpha << endl;
  cout << "dx = " << dx << endl;


  int i, j;
  for (j = 1; j <= m; j++)
    for (i = 1; i <= n; i++)
      E_prev[j][i] = R[j][i] = 0;

  for (j = 1; j <= m; j++)
    for (i = n / 2 + 1; i <= n; i++)
      E_prev[j][i] = 1.0;

  for (j = m / 2 + 1; j <= m; j++)
    for (i = 1; i <= n; i++)
      R[j][i] = 1.0;

  cout << "Grid Size       : " << n << endl;
  cout << "Duration of Sim : " << T << endl;
  cout << "Time step dt    : " << dt << endl;
  cout << "Process geometry: " << px << " x " << py << endl;
  if (no_comm)
    cout << "Communication   : DISABLED" << endl;

```

```
    cout << endl;
  }

  for (int i = 0; i < m + 2; i++)
  {
    MPI_Bcast(E[i], n + 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(E_prev[i], n + 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(R[i], n + 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }
```

To address situations where the process geometry doesn't evenly divide the mesh size, I incorporate extra data into the last process and distribute it using Scatterv instead of Scatter. This ensures that the last process may receive more data compared to others when necessary, unlike Scatter, which evenly distributes data among all processors.

```
  double **my_E;
  double **my_E_prev;
  double **my_R;

  int y_size = 0;
  if (n % py == 0)
  {
    y_size = n / py;
  }
  else
  {
    if (rank == (P - 1))
    {
      y_size = n / py + (n % py);
    }
    else
    {
      y_size = n / py;
    }
  }

  int x_size = n / px;
  int x_pos = rank % px;
  int y_pos = rank / px;

  my_E = alloc2D(y_size + 2, x_size + 2);
  my_E_prev = alloc2D(y_size + 2, x_size + 2);
  my_R = alloc2D(y_size + 2, x_size + 2);

  int interval = (x_size + 2) * y_size;

  int sendcounts[P];
  int displs[P];
  for (int i = 0; i < P; i++)
  {
    int y_size_i = n / py + (i == P - 1 ? n % py : 0);
    int interval_i = (x_size + 2) * y_size_i;

    sendcounts[i] = interval_i;
    displs[i] = i * interval;
  }
```

```
42
43  MPI_Scatterv(&E[1][0], sendcounts, displs, MPI_DOUBLE, my_E[1],
      interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
44  MPI_Scatterv(&E_prev[1][0], sendcounts, displs, MPI_DOUBLE,
      my_E_prev[1], interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
45  MPI_Scatterv(&R[1][0], sendcounts, displs, MPI_DOUBLE, my_R[1],
      interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

I utilize immediate operations for sending and receiving from both the bottom and top, followed by a waitAll to ensure that all communication is completed before executing the simulation function.

```
1     if (rank != 0){
2           MPI_Irecv(my_E_prev[0], x_size + 2, MPI_DOUBLE, upperRank
      , 0, MPI_COMM_WORLD, &reqs[requestCount++]);
3           MPI_Isend(my_E_prev[1], x_size + 2, MPI_DOUBLE, upperRank
      , 0, MPI_COMM_WORLD, &reqs[requestCount++]);
4         }
5         // send bottom
6         if (rank != (P - 1)){
7           MPI_Isend(my_E_prev[y_size], x_size + 2, MPI_DOUBLE,
      bottomRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
8           MPI_Irecv(my_E_prev[y_size + 1], x_size + 2, MPI_DOUBLE,
      bottomRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
9         }
10        MPI_Waitall(requestCount, reqs, status);
11
12        t += dt;
13        niter++;
14
15        simulate(my_E, my_E_prev, my_R, alpha, x_size, y_size, kk,
      dt, a, epsilon, M1, M2, b, x_pos, y_pos, px, py);
```

I employed identical parameters for both MPI_Gatherv and MPI_Scatterv, with the only variation being in the destination and source parameters.

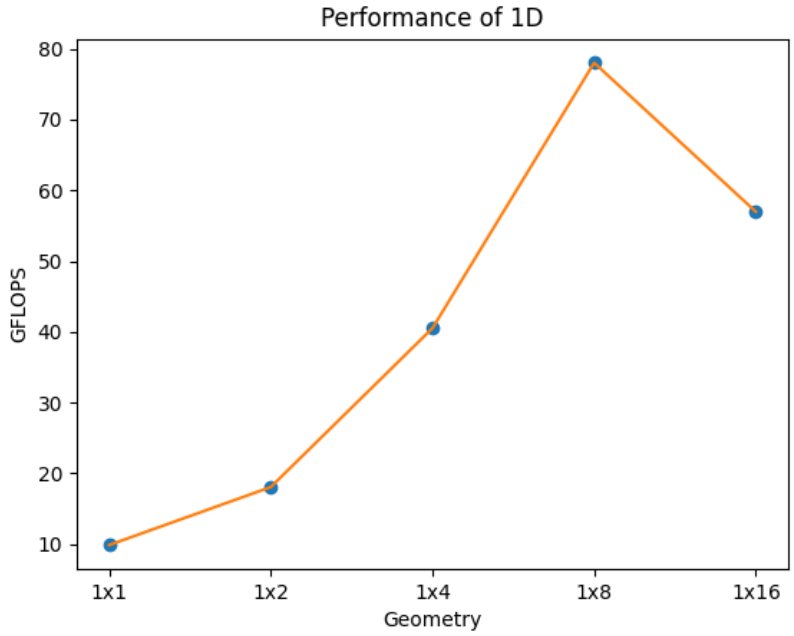
```
1   MPI_Gatherv(my_E_prev[1], interval, MPI_DOUBLE, &E_prev[1][0],
      sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
2   MPI_Gatherv(my_E[1], interval, MPI_DOUBLE, &E[1][0], sendcounts,
      displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
3   MPI_Gatherv(my_R[1], interval, MPI_DOUBLE, &R[1][0], sendcounts,
      displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

1.1 Experiment

Table 1: Study 1: Strong Scaling 1D

Configuration	Execution Time
Serial	8.10818
Geometry 1x1	9.86163
Geometry 1x2	18.0636
Geometry 1x4	40.5181
Geometry 1x8	78.0296
Geometry 1x16	56.9784

Results



(a) Study 1: Strong Scaling 1D Plot

Explanation of Speedup Curve The variance between the serial version and the 1x1 geometry primarily stems from MPI initialization and other overheads introduced by MPI gatherings. Although there is a slight difference between them, it is noticeable. Notably, the most optimal geometry is 1x8, achieving a GFLOPS rate of 78.0296. As evident, increasing the number of processes beyond 8 leads to a reduction in the GFLOP rate. Hence, the optimal choice does not necessarily involve the highest process count.

2 Part II: MPI + OpenMP Parallelism

In the second phase of this assignment, I incorporated hybrid programming by combining MPI and OpenMP. Each process now leverages multiple threads to accelerate the execution of its local tasks. The course materials indicate that the ODE solver exhibits 'No data dependency' and is inherently parallelizable, making it an ideal candidate for introducing parallelism.

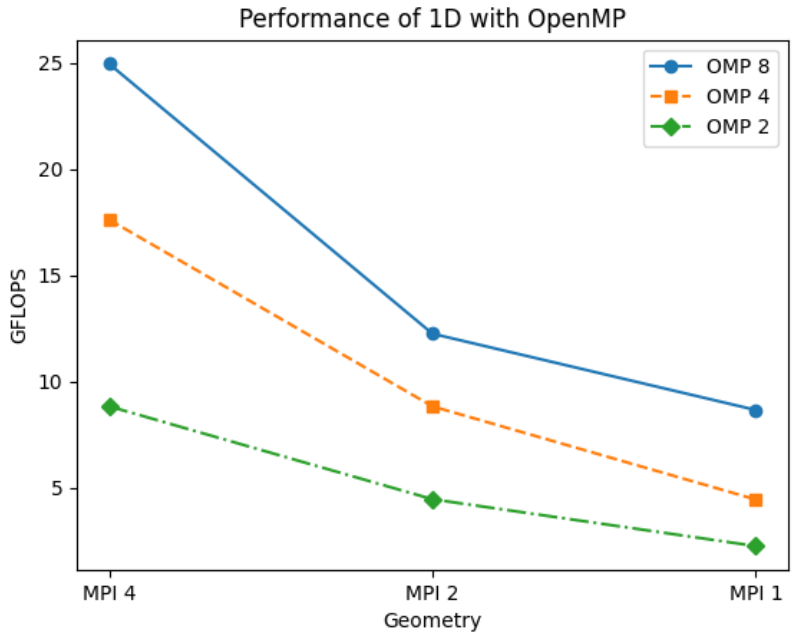
```
#pragma omp parallel
{
  #pragma omp for collapse(2)
  for (j=1; j<=m; j++){
    for (i=1; i<=n; i++) {
E[j][i] = E_prev[j][i]+alpha*(E_prev[j][i+1]+E_prev[j][i-1]-4*
  E_prev[j][i]+E_prev[j+1][i]+E_prev[j-1][i]);
    }
  }

  /*
   * Solve the ODE, advancing excitation and recovery to the
   *     next timtestep
   */
  #pragma omp for collapse(2)
  for (j=1; j<=m; j++){
    for (i=1; i<=n; i++)
E[j][i] = E[j][i] -dt*(kk* E[j][i]*(E[j][i] - a)*(E[j][i]-1)+ E[j
  ][i] *R[j][i]);
  }

  #pragma omp for collapse(2)
  for (j=1; j<=m; j++){
    for (i=1; i<=n; i++)
R[j][i] = R[j][i] + dt*(epsilon+M1* R[j][i]/( E[j][i]+M2))*(-R[j
  ][i]-kk* E[j][i]*(E[j][i]-b-1));
  }
}
```

2.1 Experiment

Table 2: Study 4 - Hybrid Programming

Geometry	GFLOPS
MPI4 + OMP8	24.9667
MPI2 + OMP8	12.2496
MPI1 + OMP8	8.65379
MPI4 + OMP4	17.6235
MPI2 + OMP4	8.82627
MPI1 + OMP4	4.43746
MPI4 + OMP2	8.82806
MPI2 + OMP2	4.44943
MPI1 + OMP2	2.24286

Results



(a) Study 4 - Hybrid Programming Plot

Observation

In my analysis, I investigated the performance of various configurations combining MPI and OpenMP (OMP) parallelization. The GFLOPS achieved for different combinations of MPI processes and OpenMP threads are presented in Table 2. Notably, increasing the number of MPI processes while maintaining 8 OpenMP threads resulted in a substantial GFLOPS improvement, signifying enhanced performance. Conversely, reducing the number of MPI processes while keeping 8 OpenMP threads led to diminished GFLOPS, underlining the inefficiency of fewer MPI processes. Furthermore, decreasing the number of OpenMP threads with the same MPI process count exhibited a noticeable GFLOPS decrease, emphasizing the significance of striking the right balance between these parameters for optimal performance in this specific workload. The best GFLOPS result was achieved with the MPI4 + OMP8 Geometry configuration. I couldn't do any more experiment because of the KUACC limitations. Maybe I would find a more optimal geometry configuration with different OMP and MPI counts.

3 Part III: MPI Parallelization with 2D Geometry

In this part, I add 2D geometry instead of 1D geometry part 1. In this part, the implementation must pack and unpack discontiguous memory locations to create messages. There are some different configurations in the code from the part 1. For example, dividing and collecting the data is different. Because we need to divide the data for 2D geometry. Also, for receiving and sending, I send and receive from 4 different sides (N,S,E,W). Let's begin by dividing the data

I've created an MPI Datatype in this code to facilitate data division and transfer. Initially, I defined a 2D matrix and specified a subarray within it. This subarray's dimensions and starting point were crucial for the subsequent Scatterv operation. The MPI Datatype was constructed to represent this subarray, taking into account its memory layout. After committing the datatype, I calculated the necessary parameters, such as the number of subarrays to send in the x and y dimensions (px and py). Lastly, I determined the send counts and displacements for Scatterv, which is essential for distributing data among MPI processes.

```
int sizes[2]    = {m, n};
int subsizes[2] = {y, x};
int starts[2]   = {0,0};
MPI_Datatype type, box;
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
    MPI_DOUBLE, &type);
MPI_Type_create_resized(type, 0, x*sizeof(double), &box);
MPI_Type_commit(&box);

  int px = n / x;
  int py = n / y;
  int sendcounts[px*py];
  int displs[px*py];

  if (rank == 0)
  {
      for (int i=0; i<px*py; i++)
        sendcounts[i] = 1;
      int disp = 0;
      for (int i=0; i<py; i++) {
          for (int j=0; j<px; j++) {
            disp = j + y * i * px;
              displs[i*px+j] = disp;
          }
      }
  }
```

After partitioning the data, I apply padding to accommodate ghost cells.

```
    MPI_Scatterv(s_src[0], sendcounts, displs, box,
                 s_dst[0], x*y, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    addPadding(s_dst, dst, y, x);
```

When transferring data from neighboring processes, the process is similar to Part 1, with the additional consideration of left and right neighbors. Initially, I copy the cells from the left and right neighbors into an local array. After completing each transfer using a WaitAll Call, I update my_E_prev by copying the left and right cells.

```
      if(y_pos != 0){
        MPI_Irecv(my_E_prev[0], x_size+2, MPI_DOUBLE, upperRank,
    BOTTOMTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
        MPI_Isend(my_E_prev[1], x_size+2, MPI_DOUBLE, upperRank,
    UPPERTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
      }

      if(y_pos != ((P - 1) / px) ) {
        MPI_Irecv(my_E_prev[y_size+1], x_size+2, MPI_DOUBLE,
    bottomRank, UPPERTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
        MPI_Isend(my_E_prev[y_size], x_size+2, MPI_DOUBLE,
    bottomRank, BOTTOMTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
      }

      if(x_pos != 0 ) {
        int i;
        for(i = 0; i < y_size; i++) {
            leftSend[i] = my_E_prev[i+1][1];
```

```
15          }
16          MPI_Irecv(&left[0], y_size, MPI_DOUBLE, rightRank, LEFTTAG,
        MPI_COMM_WORLD, &reqs[requestCount++]);
17          MPI_Isend(&leftSend[0], y_size, MPI_DOUBLE, rightRank,
    RIGHTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
18
19        }
20        if(x_pos != ((P-1) % px)) {
21          int i;
22          for(i = 0; i < y_size; i++) {
23              rightSend[i] = my_E_prev[i+1][x_size];
24          }
25          MPI_Irecv(&right[0], y_size, MPI_DOUBLE, leftRank, RIGHTTAG
        , MPI_COMM_WORLD, &reqs[requestCount++]);
26          MPI_Isend(&rightSend[0], y_size, MPI_DOUBLE, leftRank,
    LEFTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
27
28        }
29        //
30        MPI_Waitall(requestCount, reqs, status);
31
32        int i;
33        if(x_pos != 0 ) {
34            for(i = 0; i < y_size; i++) {
35              my_E_prev[i+1][0] = left[i];
36            }
37        }
38        if(x_pos != ((P-1) % px)) {
39          for(i = 0; i < y_size; i++) {
40            my_E_prev[i+1][x_size+1] = right[i];
41          }
42        }
```

Gathering data from other processes to the root process shares a common approach with data division. The primary distinction lies in modifying the destination and receive parameters. We utilize gatherV to assemble the data.

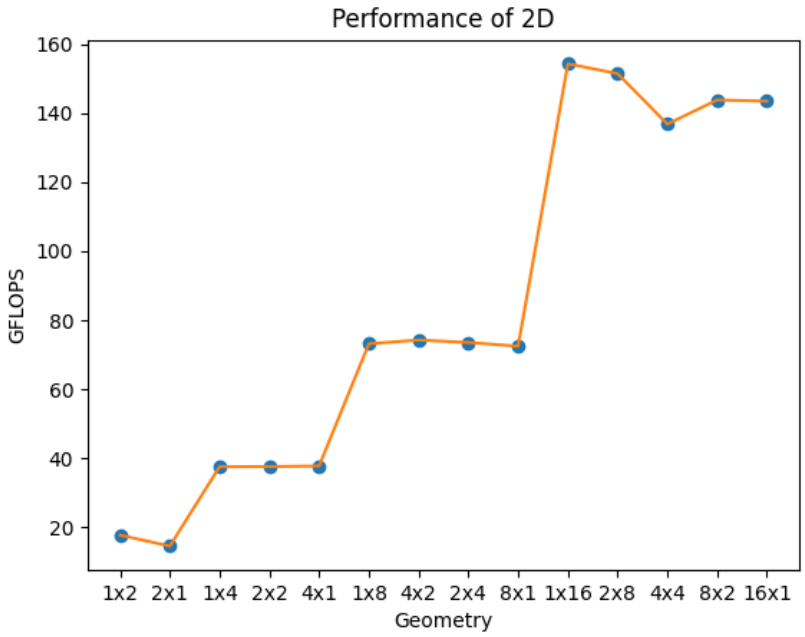
3.1 Experiment

3.1.1 Study 2 - Strong Scaling in 2D

Geometry	GFLOPS
1x2	17.6851
2x1	14.4999
1x4	37.5146
2x2	37.5534
4x1	37.728
1x8	73.1636
4x2	74.2463
2x4	73.5394
8x1	72.4491
1x16	154.342
2x8	151.507
4x4	136.844
8x2	143.86
16x1	143.515

Table 3: Table of Geometry and GFLOPS

Results
Observations
Notably, configurations with more processors, such as 1x16, 8x2, 2x8, and 16x1, exhibit superior computational throughput, indicating efficient parallel processing. Exceptionally high GFLOPS values are observed in the 1x16 and 2x8 geometries, suggesting their potential optimization for parallel workloads. Conversely, the 4x4 geometry yields slightly lower GFLOPS values, indicating potentially less efficient processing for this specific workload. It is crucial to consider that GFLOPS is influenced by various factors, including workload specifics and parallelization efficiency, emphasizing the importance of selecting an appropriate geometry to achieve optimal performance.



(a) Study 2 - Strong Scaling in 2D Plot

3.1.2 Study 3 - Measure Communication Overhead in 2D

In order to computer the communication overhead, I have used the 'no_comm' variable. Whenever I use communication call with MPI, I disabled it with this variable like in the below:

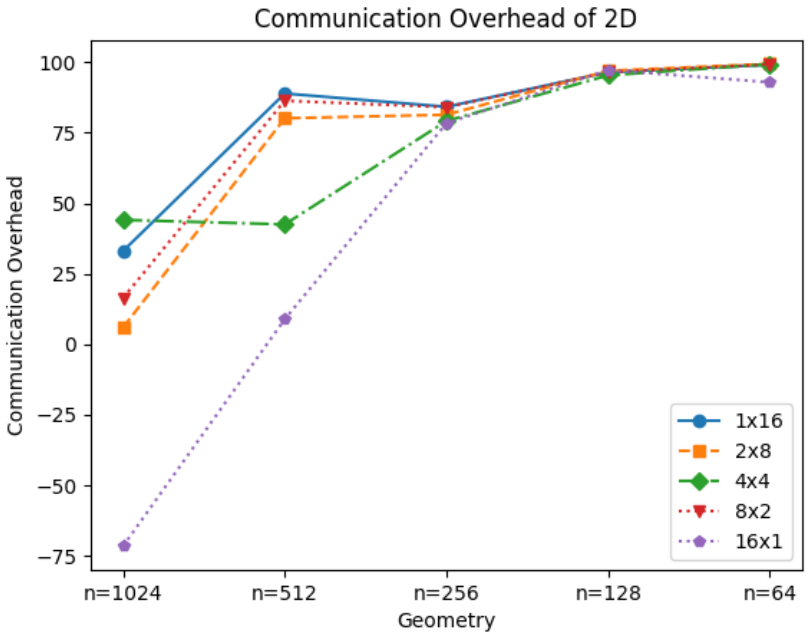
```
if (no_comm == 0)
  {
    divideData(E, my_E, x_size, y_size, n, m, rank);
    divideData(E_prev, my_E_prev, x_size, y_size, n, m, rank);
    divideData(R, my_R, x_size, y_size, n, m, rank);
  }
```

Results

Observations

As the problem size decreases, the communication overhead tends to increase. This phenomenon occurs because the calculation-to-communication ratio decreases for each core as the problem size diminishes. With less computational work per unit of communication, the overhead becomes more prominent, leading to increased inefficiencies in data exchange.

It's important to consider that when conducting experiments to calculate the communication within the algorithm, the communication tends to decrease as we reduce the problem count. This reduction is primarily because less data needs to be transferred in this configuration.



(a) Study 2 - Measure Communication Overhead in 2D Plot

An anomaly occurred with the cluster in the 16x1 configuration for n=1024. The specific cause of this anomaly remains unclear, and despite attempts to re-experiment with that data point, unfortunately, the issue could not be resolved.

4 Formulas Used

a. *Communication Overhead*

$$CommunicationOverhead = (1 - \frac{ExecutionTimewithNoCommunication}{ExecutionTimewithCommunication} * 100)$$