COMP 429/529: Project 2

Berkay Barlas

April 24, 2019

Date Performed:	March 17, 2019
Instructor:	Didem Unat

In this assignment I implemented MPI and OpenMP versions on top of given serial version for Cardiac Electrophysiology Simulation and conducted experiments.

To compile all the necessary files run m̈ake cardiacsim && make serial && make openmp¨
In this assignment I have completed

- 1D MPI version of Cardiac Electrophysiology Simulation
- 2D MPI version of Cardiac Electrophysiology Simulation
- 2D MPI+OpenMP version of Cardiac Electrophysiology Simulation
- Performance Studies for Part a, b, c, d, e

1 Implementation

1.1 MPI Implementation

Cardiac Electrophysiology simulator solves system of Ordinary Differential Equations (ODEs) together with Partial Differential Equations (PDEs).

- Initial data used in this equations are created by master process ,then, it is distributed to other processes using MPI_Scatterv.

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

- After the processes recieve the sub parts of data they only calculate their own part.

```
    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);
```

- These calculations requires some data which changes every iteration from neighbour processes, therefore, ghost cell implementation is needed. All processes use NonBlocking Messages to Communicate to their neighbours. They wait for their send and recieve requests with MPI_wait().

```
  /// send and recieve ghost cells to and from neighbor processes
    // top side send&recieve
    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++]);
    }
    // bottom side send&recieve
    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++]);
    }
    // left side send&recieve
    if(x_pos != 0 ) {
      int i;
      for(i = 0; i < y_size; i++) {
      leftSend[i] = my_E_prev[i+1][1];
      }
      MPI_Irecv(&left[0], y_size, MPI_DOUBLE, rightRank, LEFTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
      MPI_Isend(&leftSend[0], y_size, MPI_DOUBLE, rightRank, RIGHTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
    }
    // right side send&recieve
    if(x_pos != ((P-1) % px)) {
      int i;
      for(i = 0; i < y_size; i++) {
      rightSend[i] = my_E_prev[i+1][x_size];
      }
      MPI_Irecv(&right[0], y_size, MPI_DOUBLE, leftRank, RIGHTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
      MPI_Isend(&rightSend[0], y_size, MPI_DOUBLE, leftRank, LEFTTAG, MPI_COMM_WORLD, &reqs[requestCount++]);
    }
    //
    MPI_Waitall(requestCount, reqs, status);
    // left side
    int i;
    if(x_pos != 0 ) {
    for(i = 0; i < y_size; i++) {
      my_E_prev[i+1][0] = left[i];
    }
    }
    // right side
```

```
      if(x_pos != ((P-1) % px)) {
        for(i = 0; i < y_size; i++) {
          my_E_prev[i+1][x_size+1] = right[i];
        }
      }
```

- Since the data transferred from left and right sides are not sequential in memory, they are send to and recieved from array buffers.

```
double *leftSend, *rightSend, *left, *right;

right = (double*) malloc(y_size * sizeof(double));
left = (double*) malloc(y_size * sizeof(double));
rightSend = (double*) malloc(y_size * sizeof(double));
leftSend = (double*) malloc(y_size * sizeof(double));
```

- Then, the sub data in different processes are collected by master processes using MPI_Gatherv.
- To handle the case when the processor geometry doesn't evenly divide the size of the mesh. I increased the mesh size.

1.2 OpenMP Implementation

I Added OpenMP parallelism within an MPI process to support hybrid parallelism with MPI+OpenMP.

I defined parallel section where the 3 equations solved. I used #pragma omp for collapse(2) because the nested loops are data indepedent.

Example execution:

mpirun -np 1 -bind-to socket -map-by socket ./cardiacsim-openmp -n 1024 -o 2 -t 100 -x 1 -y 1

```
#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 Studies

2.1 Part A: Strong Scaling

Optimal processor geometry: 1 x 32

Single Processor Gflops Rate: 6.04524

Original Serial Code Gflops Rate: 6.00341

Results

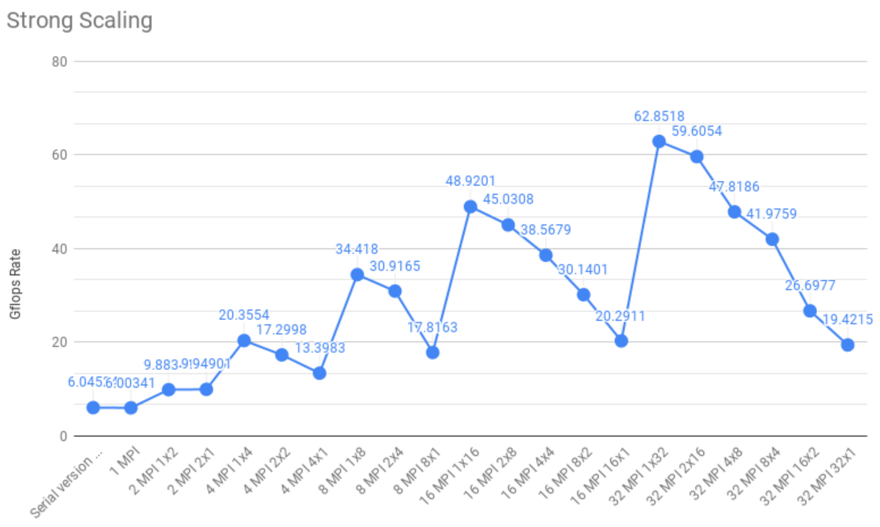

Figure 1: Strong Scaling results

Observation

There is a slight performance difference between my MPI code with single processor and original code. This difference might caused by MPI initialization. The difference is small because there is not communication between processors.

Increasing the number of processes increase the performance and 1D horizontal data partitioning performs better than other geometries.

2.2 PART B: Strong Scaling with ntasks-per-node=16

Results

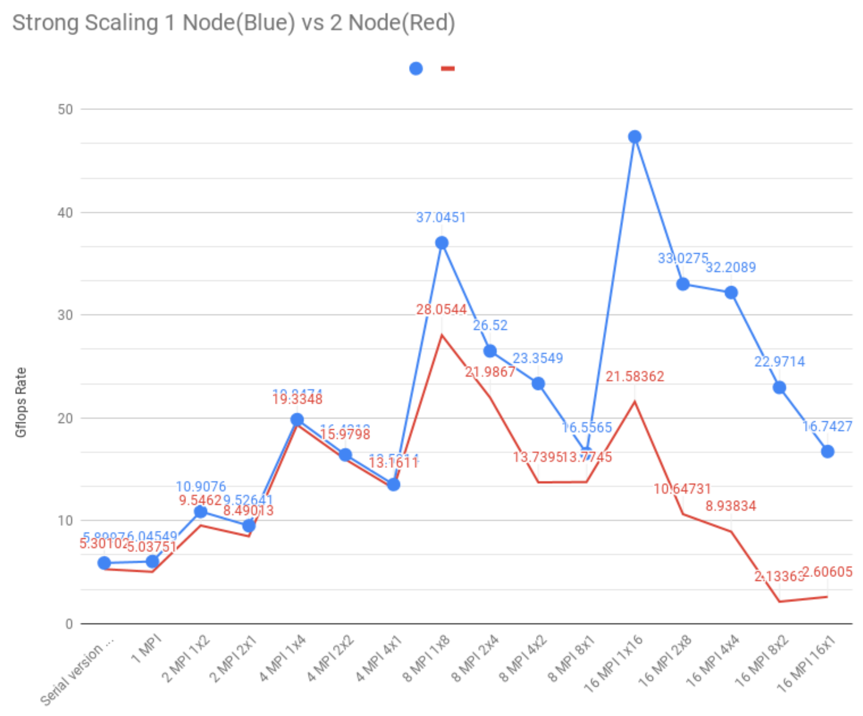

Figure 2: Strong Scaling results with Single and Dual nodes

Comparison the single node performance with dual node performance The single node performance is better than dual performance nearly for every geometry. Also, the performance difference is increases with increasing number of MPI processes. This might be caused by communication overhead between processes in different nodes.

2.3 Part C: Disabling Communication

In this part, I calculated Communication overhead by disabling Communication. When the no_comm variable is equal to 1 the communication is disabling.

```
    if(no_comm == 0) {

    // Communication codes

    }
```

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

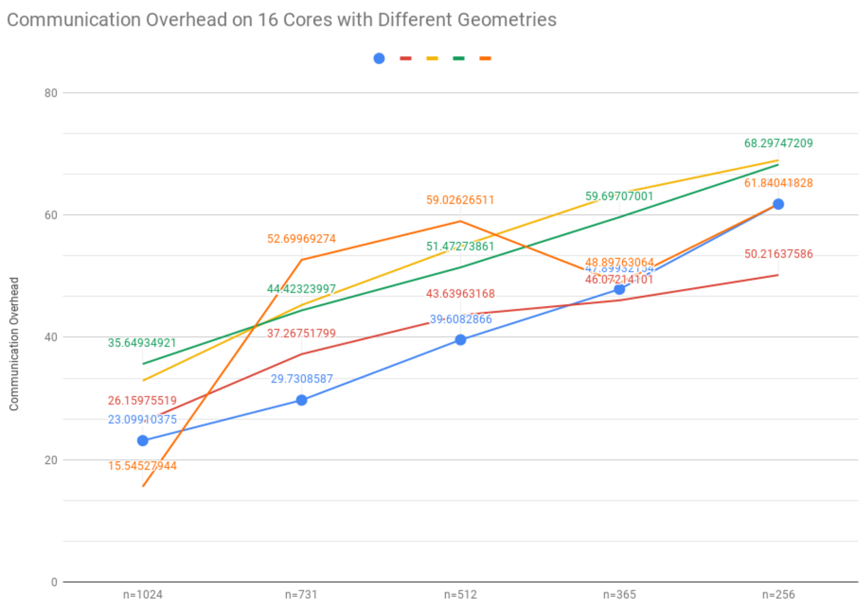

Figure 3: Communication Overhead results with different geometries on 16 cores. **Blue=1x16 Red=2x8 Yellow=4x4 Green=8x2 Orange=16x1**

Observation

The communication overhead increases as the total amount of data decrease because the calculation / communication ratio decrease for every core.

Also, the communication overhead results are higher than what I expected this might happened because of the change in calculations. A better approach would be creating the no_comm system such that the processes would make same calculations with communication enabled system.

2.4 Part D: Single and Dual Node Performance with OpenMP

In this study, 1D geometry used for MPI+OpenMP hybrid parallelism.

Which combination of MPI+OpenMP is the fastest?:
8 MPI Processes + 4 OpenMP Thread is the fastest

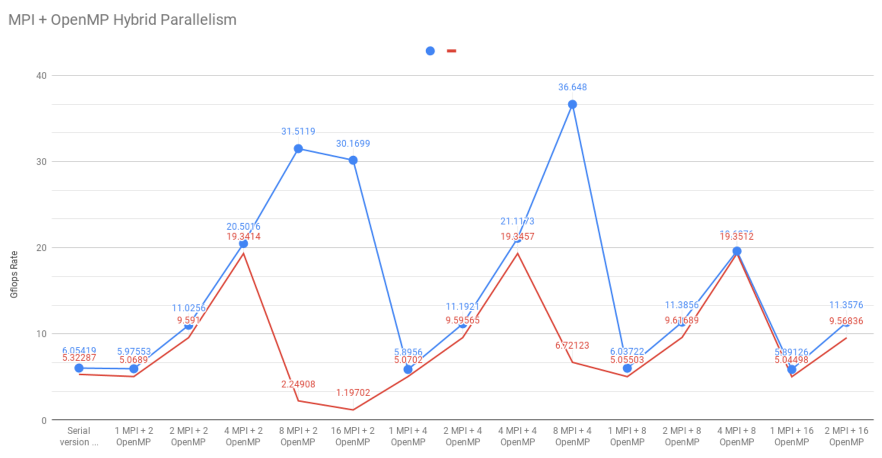

Figure 4: Single(Blue) and Dual(Red) node performance study

Observations

The single node performance is also better than dual performance nearly for every geometry in this study with OpenMP. This might be caused by communication overhead between processes in different nodes. The 2 nodes might be run on different machines, this might create an unequal systems which gives unstable results.

The performance difference is also effected by the thread number. When the number of MPI cores increases the performance decrease due to OpenMP parallelization overhead.

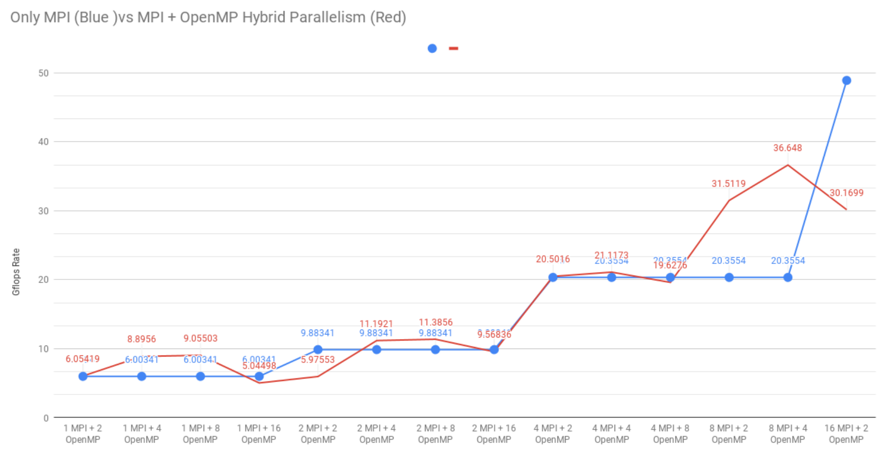

Figure 5: MPI Only(Blue) vs MPI + OpenMP performance comparison on single node

Observations

With the same number of MPI processes, Number of OpenMP thread increases performance until some point ,then, decrease the performance.

Number of MPI cores affect performance more than Number of OpenMP Threads. Overall performance increases as the MPI core number increase.

2.5 Part E: Weak Scaling Study

In this study, I conduct a weak scaling to observe the running time while successively doubling the number of cores, starting at P=1 core with N = 256 ending at 32 cores on two nodes. I calculated t such that iteration number is equals 10000.

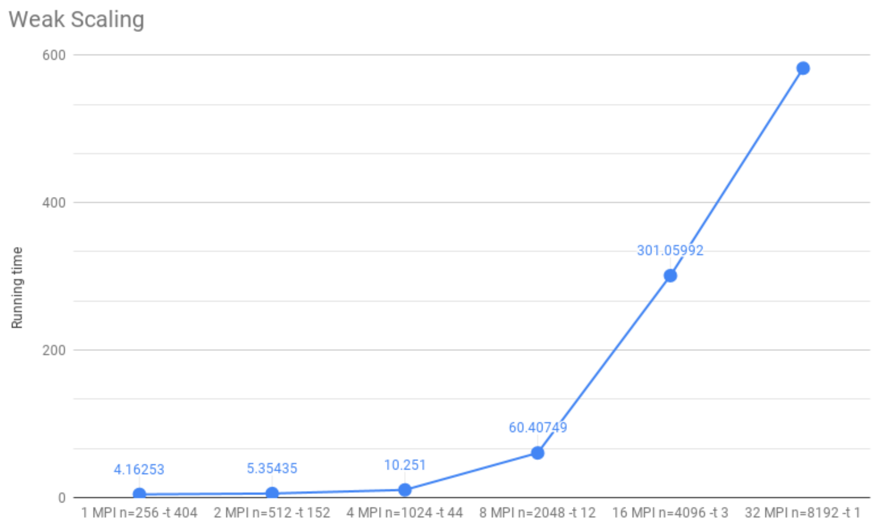

Figure 6: Results for the weak scaling

Eventhough, number of cores and N values are growing linearly, the running time increases polynomially because the complexity of system is N2̂.

3 Formulas Used

a. *Communication Overhead*

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