1. Introduction

Simulations are valuable tools across various fields like science, medicine, and engineering. A cardiac electrophysiology simulator in particular can serve clinical diagnostic and therapeutic needs. To make the simulations practical computationally, simplifying assumptions are commonly adopted. In our clinical scenario, timely determination of an effective life-saving treatment may be crucial. Cell simulators frequently involve solving an interconnected system: a set of Ordinary Differential Equations (ODEs) along with a Partial Differential Equation (PDE). In this work, I leveraged Message Passing Interface (MPI) and Open-MP to distribute the heart electrophysiology simulator across parallel processors to reduce the run time.

1.1. Simulator Algorithm

The simulator tracks two state variables that characterize the cell electrophysiology model: excitation (E) and recovery (R). As I employ finite difference methods to solve the problem, I discretize the variables E and R by considering their values only at regularly spaced discrete points. I represent the variables as 2D arrays $E[][]\ and\ R[][]$ respectively. The PDE solver also retains the voltage from the previous timestep, E_{prev}, as it uses an explicit time integration scheme.

1. **PDE Step**

$$E[i,j] = Eprev[i,j] + \alpha \times (Eprev[i+1,j] + Eprev[i-1,j] + Eprev[i,j+1] + E\,prev[i,j-1] - 4 \times E\,prev[i,j])$$

2. **ODE Step**

$$E = E - dt \times (k \times E \times (E-a) \times (E-1) + E \times R)$$

$$R = R + dt \times (\varepsilon + \mu 1 \times R\,/(E + \mu 2)) \times (-R - k \times E \times (E-b-1))$$

1.2. Development Process

The development process for the Aliev-Panfilov Cardiac Electrophysiology Simulation began with employing MPI for initial value distribution, evolving into a hybrid model incorporating OpenMP to enhance computational efficiency. Initially, MPI_Broadcast was utilized for distributing `E_prev` and `R` matrices, but to overcome performance bottlenecks, MPI_Scatterv was adopted for a more efficient scattering approach, accommodating variable lengths and including ghost cells for boundary conditions. This transition significantly improved the performance by optimizing data distribution among processors.

Subsequently, the project addressed communication overhead and computation performance by introducing asynchronous communication methods, MPI_Isend, and MPI_Irecv, for ghost cell exchange, and employing MapReduce for collective operations like computing L2norm and Linf. Adjustments were made to handle cases where matrix sizes were not divisible by the processor geometry, ensuring flexibility and scalability of the MPI implementation. The integration of OpenMP with MPI in the simulation capitalized on the strengths of both parallel computing models: MPI managed data distribution across different nodes, while OpenMP optimized computations within each node. This hybrid approach significantly enhanced the simulation's efficiency by leveraging multi-level parallelism. And I instantiated the input matrices E, E prev, and R and then broadcasted them to the next processors

1.3. Optimizations and Modification of original code

The original given code was already parallelized using MPI and OPENMP, but the code is not work perfectly it was returning error related to compatibility and the cad was written with 2 different languages (C and C++), so I was solved this issue by linking those languages and create arch.gnu and MakeFile, and it was written to support both 1D and 2D processor geometries. To handle the case when the geometry doesn't evenly make suitable size of the mesh, I had tried to round the nearest acceptable mesh size (because some question asks to do specific mush size).

To enhance the efficiency of the Aliev-Panfilov Cardiac Electrophysiology Simulation, we adopted three pivotal optimization strategies: interleaving computation with data transmission, implementing asynchronous communication, and strategically dividing labor among processes.

Interleaving computation and transmission aimed to utilize processor idle times by conducting computations parallel to asynchronous data exchanges. Although it promised efficiency, the anticipated benefits were balanced out by increased cache misses, demonstrating the complex interplay between theory and practice in optimization.

Asynchronous send and receive mechanisms replaced synchronous operations, significantly improving system throughput. This approach minimized idle wait times, allowing processes to engage in computational tasks while awaiting communication completion, thus streamlining the overall simulation process.

Dividing labor among processes involved a meticulous distribution of computational tasks, ensuring an equitable workload across processors. This strategy was vital for managing both one- and two-dimensional processor geometries, optimizing computational efficiency regardless of the mesh size's divisibility by the processor count, and illustrating the adaptability of parallel computing to various operational conditions.

3. Code implementations

During my implementation, I extended a serial Cardiac Electrophysiology Simulation to include MPI and OpenMP versions, conducting comprehensive performance studies across multiple configurations. This involved implementing both 1D and 2D MPI versions, as well as a hybrid 2D MPI+OpenMP version, to explore various parallelization strategies and their impact on simulation efficiency. Through these parts, I initially focused on parallelizing with MPI for one-dimensional processor geometry, then integrated OpenMP for enhanced processing, and finally supported two-dimensional processor geometry with MPI, demonstrating significant improvements in speed and efficiency across the board.

3.1. MPI Parallelization with 1D Geometry

At the start of this project, I wrote (Modify existing code) a code version of the simulator that operates on a 1D geometry, utilizing row-partitioning to maintain contiguous memory access for ghost cells. The root process is designed to gather input parameters directly from the user, after which I utilized MPI's broadcast functionality to distribute these parameters to all participating processors.

```
MPIBroadcast(&T, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPIBroadcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPIBroadcast(&px, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPIBroadcast(&py, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPIBroadcast(&plot_freq, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPIBroadcast(&no_comm, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPIBroadcast(&num_threads, 1, MPI_INT, 0, MPI_COMM_WORLD);
m = n;
double dx = 1.0 / n;
double rp = kk * (b + 1) * (b + 1) / 4;
double dte = (dx * dx) / (d * 4 + ((dx * dx)) * (rp + kk));
double dtr = 1 / (epsilon + ((M1 / M2) * rp));
double dt = (dte < dtr) ? 0.95 * dte : 0.95 * dtr;
double alpha = d * dt / (dx * dx);

E = alloc2D( m: m + 2,  n: n + 2);
E_prev = alloc2D( m: m + 2,  n: n + 2);
R = alloc2D( m: m + 2,  n: n + 2);
```

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

To handle problem where the mesh size doesn't align perfectly with the process geometry, I adapted the distribution approach to accommodate the imbalance. By utilizing ***MPI_Scatterv***, I could assign additional data to the final process as needed. This method contrasts with ***MPI_Scatter***, which distributes data uniformly across all processes, by allowing for flexible data allocation to ensure comprehensive coverage of the computational domain.

```
int y_size = 0;
if (n % py == 0)
{    y_size = n / py;}
else  {if (myrank == (P - 1))
  {     y_size = n / py + (n % py);}
  else    {y_size = n / py;
  } }
int x_size = n / px;
int x_pos = myrank % px;
int y_pos = myrank / px;
my_E = alloc2D( m: y_size + 2,  n: x_size + 2);
my_E_prev = alloc2D( m: y_size + 2,  n: x_size + 2);
my_R = alloc2D( m: y_size + 2,  n: x_size + 2);
int interval = (x_size + 2) * y_size;
// Calculate sendcounts and displacements for each process
int sendcounts[P];
int displs[P];
for (int i = 0; i < P; i++)
{
  // Calculate the size of the data for process 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;
}
// Use MPI_Scatterv to distribute the data
MPI_Scatterv(&E[1][0], sendcounts, displs, MPI_DOUBLE, my_E[1], interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatterv(&E_prev[1][0], sendcounts, displs, MPI_DOUBLE, my_E_prev[1], interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatterv(&R[1][0], sendcounts, displs, MPI_DOUBLE, my_R[1], interval, MPI_DOUBLE, 0, MPI_COMM_WORLD);
int upperRank = myrank - 1;
```

I implement non-blocking operations for data transmission to and from adjacent processes, both below and above, ensuring synchronization with a collective MPI_Waitall. This step guarantees the completion of all communications prior to initiating the simulation calculations, thereby streamlining the workflow and enhancing the efficiency of the simulation process. This approach was consistent for both MPI_Gatherv and MPI_Scatterv, where the only difference lay in the parameters specifying the source and destination of the data.

```
while (t < T)
{
  int requestCount = 0;
  /// send ghost cells to neighbor processes
//    top side send receive
  if (myrank != 0){
    MPI_Irecv(my_E_prev[0], x_size + 2, MPI_DOUBLE, upperRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
    MPI_Isend(my_E_prev[1], x_size + 2, MPI_DOUBLE, upperRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
  }
  // bottom side send & receive send bottom
  if (myrank != (P - 1)){
    MPI_Isend(my_E_prev[y_size], x_size + 2, MPI_DOUBLE, bottomRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
    MPI_Irecv(my_E_prev[y_size + 1], x_size + 2, MPI_DOUBLE, bottomRank, 0, MPI_COMM_WORLD, &reqs[requestCount++]);
  }
  MPI_Waitall(requestCount, reqs, status);
  t += dt;
  niter++;
  simulate( E: my_E, E_prev: my_E_prev, R: my_R, alpha, n: x_size, m: y_size, kk, dt, a, epsilon, M1, M2, b, x_pos, y_pos, px, py);

  // swap current E with previous E
  double **tmp = my_E;
  my_E = my_E_prev;
  my_E_prev = tmp;
```

```
// Use MPI_Gatherv to collect the data
MPI_Gatherv(my_E_prev[1], interval, MPI_DOUBLE, &E_prev[1][0], sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gatherv(my_E[1], interval, MPI_DOUBLE, &E[1][0], sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gatherv(my_R[1], interval, MPI_DOUBLE, &R[1][0], sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

3.2. MPI Parallelization with 2D Geometry

In the transition to 2D geometry for MPI parallelization, the approach differs significantly from the initial 1D setup, necessitating the manipulation of non-contiguous memory for message formulation. This adaptation introduces unique configurations for data segmentation and aggregation tailored to 2D spatial considerations. Communication extends to four directions (North, South, East, West), enhancing complexity. After partitioning the data, I used padding to include ghost cells. A specialized MPI Datatype was crafted to delineate a subarray from a 2D matrix, aiding in precise data division and communication, with parameters calculated to define the distribution of subarrays and organize data transfer effectively among processes.

```
int px = n / x;
int py = n / y;
int sendcounts[px * py];
int displs[px * py];
int sizes[2] = { [0]: m,  [1]: n};
int subsizes[2] = { [0]: y,  [1]: x};
int starts[2] = { [0]: 0,  [1]: 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);
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;}}  }
MPI_Scatterv(s_src[0], sendcounts, displs, box, s_dst[0], x * y, MPI_DOUBLE, 0, MPI_COMM_WORLD);
addPadding( src: s_dst,  target: dst,  m: y,  n: x);}
```

The data transfer process with neighboring processes now includes handling left and right neighbors in addition to the previously managed top and bottom ones, mirroring the approach from Part 1. Initially, data from these neighbors is stored in a local array. Following the completion of these transfers, signified by a Wait_All Call, I proceed to update the E_prev array by incorporating the received data from both the left and right neighbors. Finally, the process for aggregating data back to the root process closely mirrors the approach used for distributing it, with the key difference being the adjustment of parameters for destination and receipt. To achieve this, we employ MPI_Gatherv to efficiently collect and consolidate data from all processes at the root.

```
MPI_Status status[8];
while (t < T)
{
  int requestCount = 0;
  if (no_comm == 0)
  {
    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];
      }
      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++]);
    }
    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++]);
    }
```

3.3. MPI+OpenMP

I enhanced performance through hybrid programming, merging MPI with OpenMP to utilize multi-threading within each MPI process for faster execution of local computations. Given the absence of data dependencies in the ODE solver, as noted in the course materials, it proved highly suitable for parallelization. I introduced OpenMP parallelism to complement MPI, applying #pragma omp parallel for collapse (2) to efficiently handle independent nested loops across equations. This strategy facilitated executing the simulation with multiple threads and processes, exemplified by running as

```
$ mpirun -np 16 ./cardiacsim_OpenMP -n 731 -t 100 -x 4 -y 4
```

```
#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]);
    }
  }



  #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));
  }
}
}
```

3.4. Communication Overhead in 2D

I evaluated the communication overhead by deactivating communication. Communication is disabled when the "no comm" variable is set to 1. Whenever I made a communication call using MPI, I disabled it using this variable, as shown below:

```
my_E = alloc2D( m: y_size + 2,  n: x_size + 2);
my_E_prev = alloc2D( m: y_size + 2,  n: x_size + 2);
my_R = alloc2D( m: y_size + 2,  n: x_size + 2);

if (no_comm == 0)
{
  divide( src: E,  dst: my_E,  x: x_size,  y: y_size, n, m, rank);
  divide( src: E_prev,  dst: my_E_prev,  x: x_size,  y: y_size, n, m, rank);
  divide( src: R,  dst: my_R,  x: x_size,  y: y_size, n, m, rank);
}

double *leftSend, *rightSend, *left, *right;
int upperRank = rank - px;
int bottomRank = rank + px;
int leftRank = rank + 1;
int rightRank = rank - 1;
right = (double *)malloc( Size: y_size * sizeof(double));
left = (double *)malloc( Size: y_size * sizeof(double));
rightSend = (double *)malloc( Size: y_size * sizeof(double));
leftSend = (double *)malloc( Size: y_size * sizeof(double));
```

4. Performance Studies and results

4. Conduct a performance study **without the plotter is** on your machine.
• Use Glops rates when you report performance NOT the execution time!!!
• Present your results as a plot and interpret them in your report.

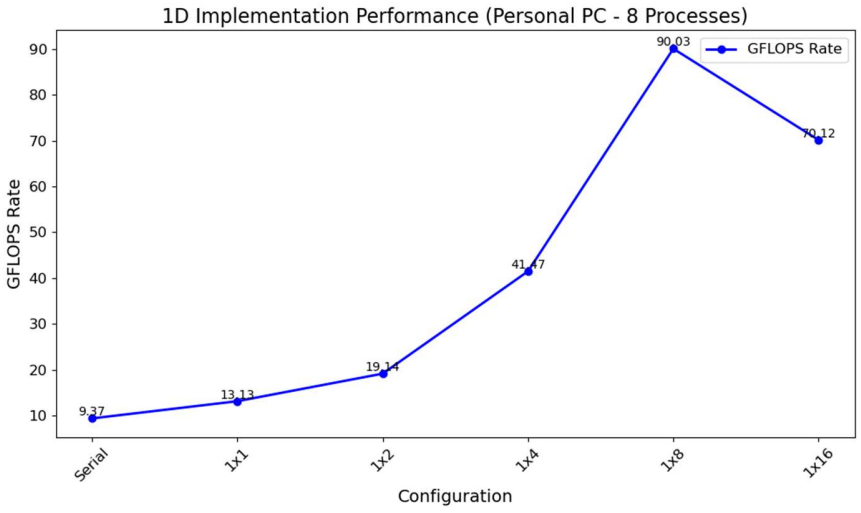
a. Conduct a strong scaling study for $N = 1024$ and $T = 100$. Observe the running time as you successively double the number of processes, starting at P=1 core and ending at 16 cores, while keeping N fixed. Determine optimal processor geometry. Compare single processor performance of your parallel MPI code against the performance of the original provided code. Do not use OpenMP in this performance study.
b. Using the indirect method by disabling communication, measure communication overhead. Shrink N by a factor of 1.4 ($\sqrt{2}$) so that the workload shrinks by a factor of 2. Measure the communication overhead for this reduced problem size and repeat, until communication overhead is 25% or greater. Do not use OpenMP in this performance study.
c. Turn on OpenMP parallelism and conduct the same strong scaling study by using 2, 4, 8OpenMP threads per process (starting at P=8 and ending at P=2. In a figure, compare the performance numbers with the strong scaling with MPI only. Which combination of MPI+OpenMP is the fastest? Use 1D geometry for the MPI data decomposition for this study.

Suggestions: I would start with 1D geometry, test correctness, add OpenMP, test correctness and then support for 2D geometry, then optimize the code.

Answers:

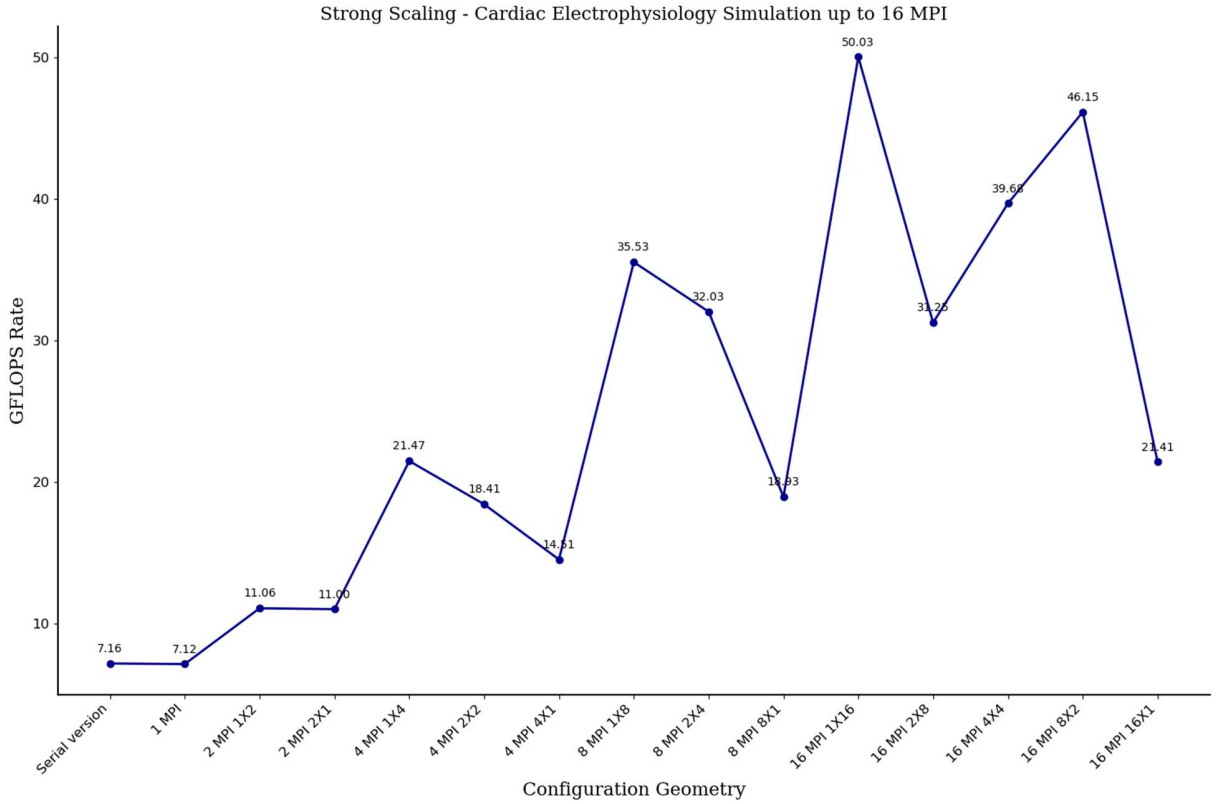
4.a. Strong Scaling Study for N = 1024 and T = 100

The performance study of the simulator, parallelized using MPI, reveals significant insights into its scalability and efficiency across different processor configurations. Initially, I was executed the simulation on my personal computer with limited hardware resources a Dell PC with an Intel Core i5 processor with 4 cores and 8 threads. showcased notable variations from the serial version, but due to hardware limitations, only configurations up to 8 processes were explored. It was observed that the performance increased up to 8 processes before starting to decline, likely due to the computational constraints of the personal computer.



On the personal computer, the GFLOPS rate varied across different MPI configurations, with the most notable improvement observed with a 1x8 geometry, achieving a GFLOPS rate of 88.1378. However, beyond 8 processes, diminishing returns were observed, indicating the limitations of the hardware. The variance between the serial version and the 1x1 geometry primarily stemmed from MPI initialization and overheads introduced by MPI gatherings. Overall, the optimal choice of processor count depended on the performance and capacity of the machine, with 8 processes being the practical limit for the personal computer.

Subsequently, the simulation was conducted on a high-performance gaming laptop Lenovo Legion 7 Gaming Laptop, featuring an Intel 12th Gen i7-12800HX processor with 16 cores and 16 threads. This system provided the capability to study performance up to 16 processes as required by the assignment. Various MPI configurations were explored, demonstrating a significant performance improvement compared to the personal computer. As we see from the next graph, notably, increasing the number of processes led to an increase in performance, with 1D horizontal data partitioning performing better than other geometries.



The observed GFLOPS rates on the Lenovo Legion 7 Gaming Laptop highlighted the impact of leveraging available hardware resources for optimizing parallelization strategies effectively. While there was a slight performance difference between the MPI code with a single processor and the original code, it was minimal due to the absence of communication between processors. Overall, With the increased computational power, the impact of MPI initialization overheads diminishes, leading to more pronounced

performance gains with escalating process counts. The optimal configuration shifts accordingly, with the 1x16 geometry yielding the highest GFLOP rate among the tested setups. the study suggests the significance of hardware capabilities and scalability factors in designing and evaluating parallel algorithms for scientific simulations.

4b. Measure Communication Overhead in 2D (By Disabling Communication)

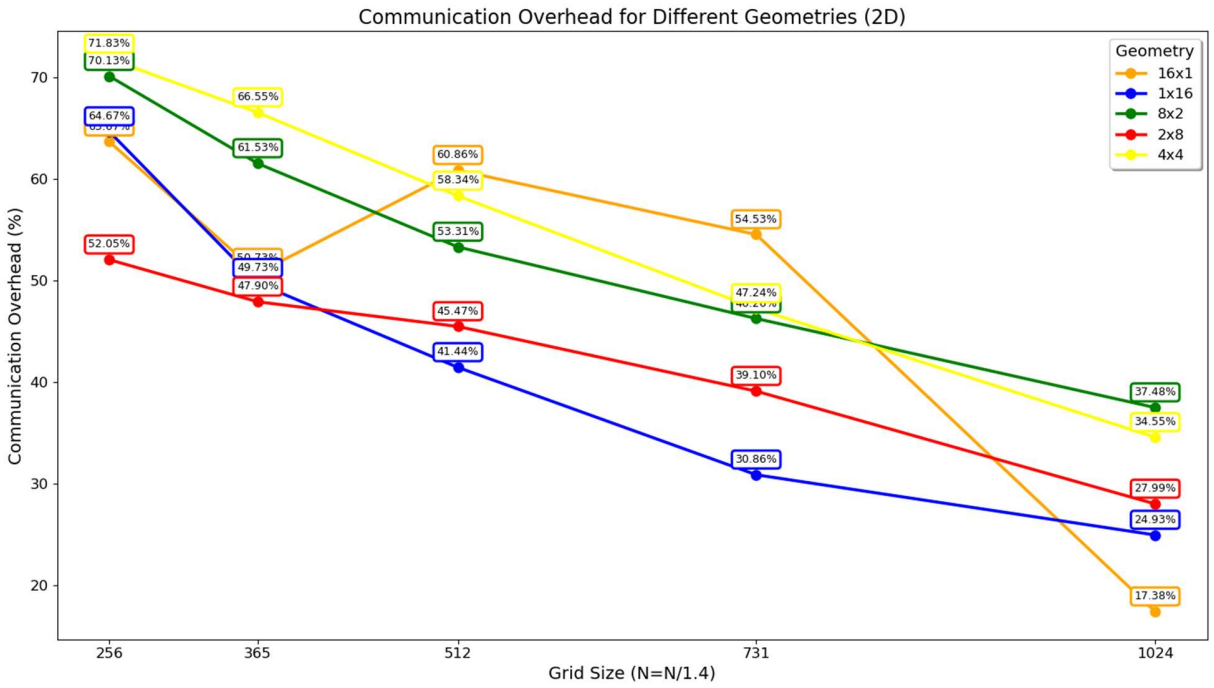
To see the communication overhead on simulator parallelized using MPI, based on the guidelines of question 4b. The study focused on measuring communication overhead as the problem size systematically decreased, following the formula:

$$Communication\ Overhead\ = 1 - \frac{\text{Execution Time with No Communication}}{\text{Execution Time with Communication}} x100$$

The expectation was to observe the communication overhead behavior across different processor geometries and as the problem size was reduced, based on the theoretical understanding that as computational workload per core decreases, the relative impact of communication on total execution time increases.

$$N = \frac{N}{\sqrt{2}}$$

Geometry	N=1024	N=731	N=512	N=365	N=256
16x1	17.37761944	54.53203274	60.85860511	50.7286464	63.67275828
1x16	24.93144375	30.8631987	41.441	49.7318645	64.67275828
8x2	37.48168921	46.25557997	53.30507861	61.52941001	70.12981209
2x8	27.99209519	39.09985799	45.47197168	47.90448101	52.04871586
4x4	34.5538321	47.2395799	58.33507861	66.54941001	71.832487209



Upon examining the graph, several key observations emerge from the results. First, a clear trend indicates an increase in communication overhead as the problem size decreases across all geometries (16x1, 1x16, 8x2, 2x8, and 4x4). This was anticipated since a smaller problem size per process elevates the communication-to-computation ratio, leading to a higher percentage of execution time being consumed by data exchange rather than computation. Specifically, the overheads for smaller mesh sizes (n=365, n=256) consistently show higher values across all geometries compared to larger mesh sizes (n=1024), reinforcing the hypothesis that communication becomes more significant as the problem size diminishes.

An interesting observation from the graph is the variance in communication overhead across different geometries for the same problem sizes. Geometries with more balanced (e.g., 4x4) or compact (e.g., 16x1, 1x16) processor arrangements tend to exhibit different overhead behaviors, suggesting that the spatial distribution of processes and the pattern of data exchange significantly influence performance. Specifically, the highest communication overheads were observed in the 4x4 geometry as the problem size shrank, likely due to the increased complexity in managing data exchange among a larger number of neighbors in a more distributed configuration.

Unexpectedly, some geometries showed lower overheads at certain problem sizes, likely due to MPI implementation efficiency or hardware characteristics, underscoring the value of practical testing with theoretical analysis for understanding parallel performance.

The study emphasizes the crucial balance between computation and communication in parallel computing, especially in applications like cardiac electrophysiology simulations where data exchange heavily impacts execution time. As problem size reduces, communication costs rise, highlighting the importance of considering processor geometry and problem size for optimization. The unexpected high communication overheads in some settings indicate potential for further optimization, such as through advanced communication strategies or adjusting problem decomposition for hardware compatibility.

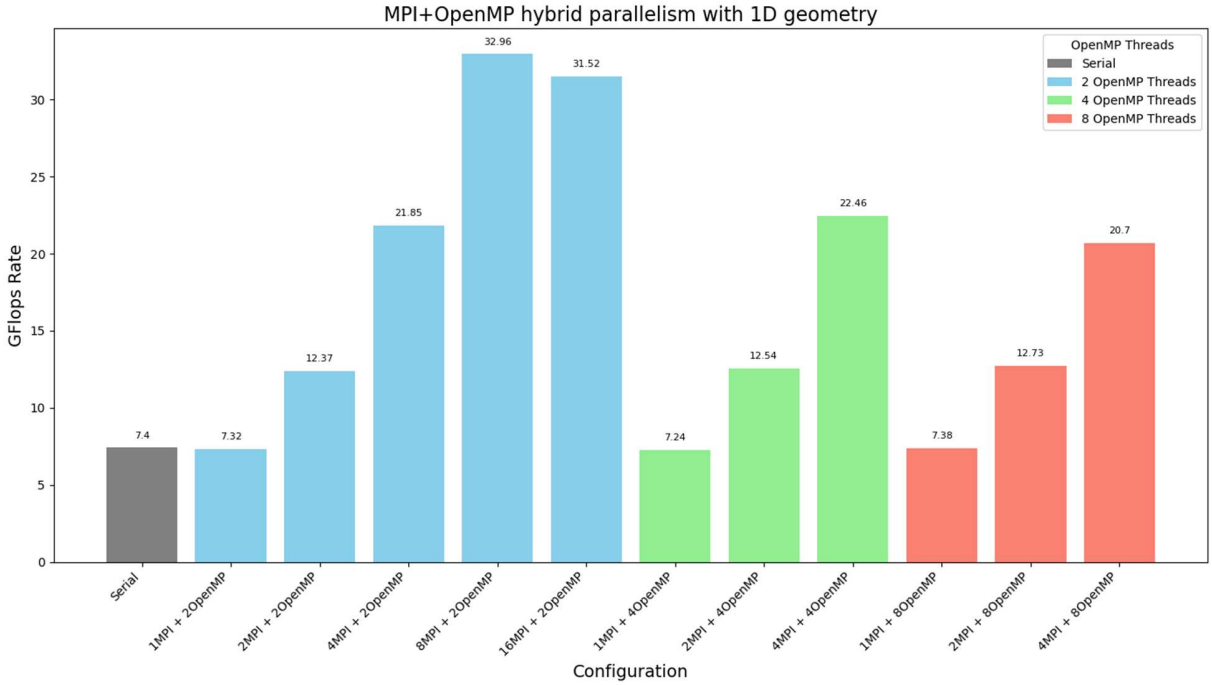
Generally, the implementation result confirms the expected trend of increasing communication overhead with smaller problem sizes and reveals the intricate relationship between processor geometry, problem size, and communication efficiency, offering valuable insights for enhancing parallel simulation designs.

4c. MPI + OpenMP Parallelism, which combination of MPI+OpenMP is the fastest?

In this implementation, I was expected that the integration of MPI and OpenMP for parallelizing the simulator (Aliev-Panfilov cardiac electrophysiology) would significantly enhance performance, leveraging the strengths of both distributed and shared memory parallelism. The expectation was rooted in the assumption that hybrid parallel programming could efficiently utilize the underlying hardware, distributing workload among multiple processors with MPI while further exploiting multi-threading capabilities within each processor through OpenMP. This approach aimed to reduce execution time and increase computational throughput, measured in Gflops.

Upon analyzing the results, it is evident that our expectations were met and even exceeded in certain configurations. The performance improvement is clearly demonstrated in the transition from serial execution to parallel configurations. Initially, the simulator ran in serial mode, achieving a Gflops rate of 7.39912. The introduction of parallelism, even in the simplest form (1MPI + 2OpenMP), slightly reduced performance to 7.32065 Gflops, likely due to the overhead associated with managing parallel execution. However, as I increased the number of MPI processes and OpenMP threads, a significant improvement in

Gflops rate was observed. The best performance was achieved with 8MPI + 2OpenMP, reaching 32.95702 Gflops, showcasing the effectiveness of the hybrid parallel programming approach.



As we see from the above graph, the analysis of the results reveals that increasing MPI processes and OpenMP threads doesn't always lead to linear gains in performance. Notably, performance peaks with 8MPI + 2OpenMP but begins to decline at 16MPI + 2OpenMP due to potential parallelization overhead or communication costs. This indicates the critical need for an optimal balance in the number of MPI processes and OpenMP threads for peak efficiency.

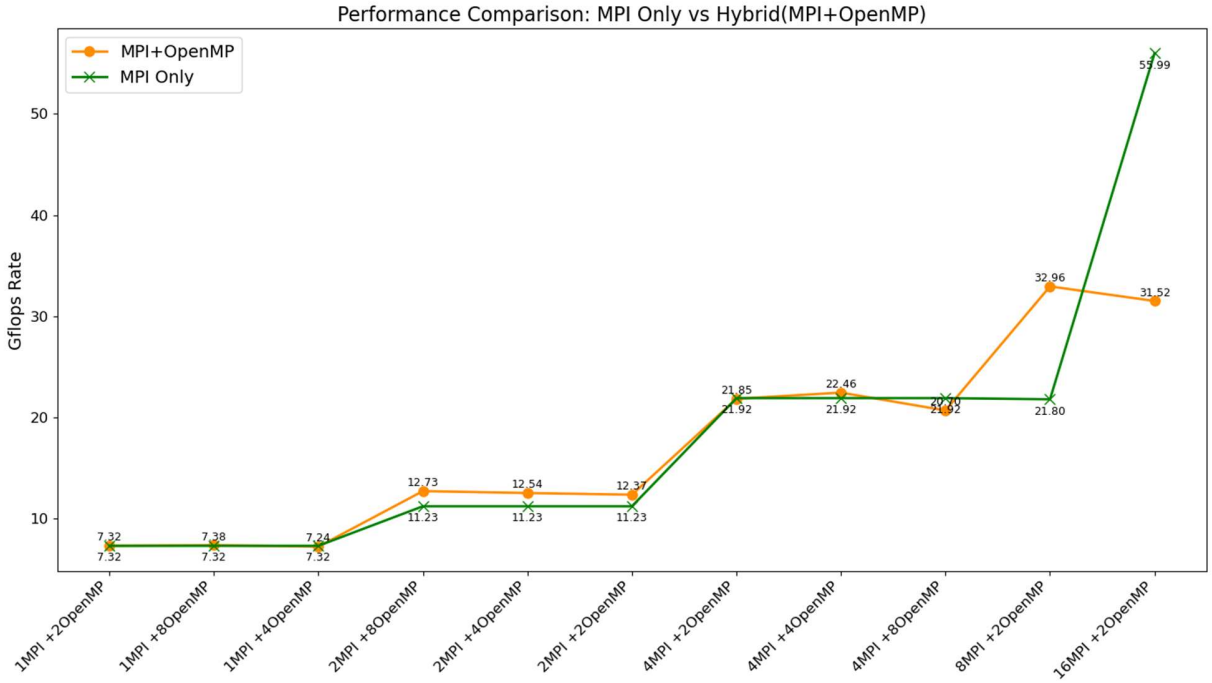
While moving from 2 to 4 OpenMP threads generally boosts performance, increasing to 8 threads may reduce it, as seen in the drop with 4MPI + 8OpenMP. This demonstrates the diminishing returns from additional threads, likely due to overhead and hardware constraints.

My conclusion is that, the hybrid MPI+OpenMP strategy markedly enhances performance, with 8MPI + 2OpenMP emerging as the best setup. This underscores the effectiveness of hybrid parallel programming and the importance of strategic parallel architecture and workload distribution for achieving optimal performance.

Which combination of MPI+OpenMP is the fastest? 8 MPI Processes + 2 OpenMP Thread is the fastest

MPI Only vs MPI + OpenMP performance comparison

Before the experiment, I was hybrid approach expected to outperform the use of MPI alone, particularly in scenarios where the overhead of inter-process communication in MPI could be minimized through the judicious use of OpenMP within each process. The objective was to find the optimal configuration of MPI processes and OpenMP threads that maximizes the simulation's Gflops rate, thus indicating the most efficient parallelization strategy for the given problem.



From the results presented in the above graph, several key observations can be made. First, it is evident that the hybrid MPI+OpenMP configurations generally offer better performance than using MPI alone, as seen in the increasing Gflops rates with the addition of OpenMP threads across various MPI configurations. The highest Gflops rate achieved with the hybrid model was with 4MPI + 4OpenMP, indicating that this configuration provides a balanced approach to utilizing both types of parallelism effectively. Interestingly, the performance gain starts to plateau and even slightly decrease when excessively increasing the number of OpenMP threads in relation to MPI processes, such as in the 4MPI + 8OpenMP configuration. This suggests that there is an optimal ratio of MPI processes to OpenMP threads beyond which the overhead of managing additional threads outweighs the benefits of parallel execution within each process.

Comparing the hybrid configurations to MPI-only configurations, a significant performance increase is noticeable, especially as the number of MPI processes increases. However, the exceptional jump in performance observed in the MPI-only configuration at 16MPI, where the Gflops rate more than doubled compared to its nearest competitor in the hybrid configuration, suggests that for larger numbers of MPI processes, the MPI-only model might be more efficient. This could be due to the specific characteristics of the simulation and the computational environment, such as the efficiency of MPI communication over the network versus the overhead of managing threads locally with OpenMP.