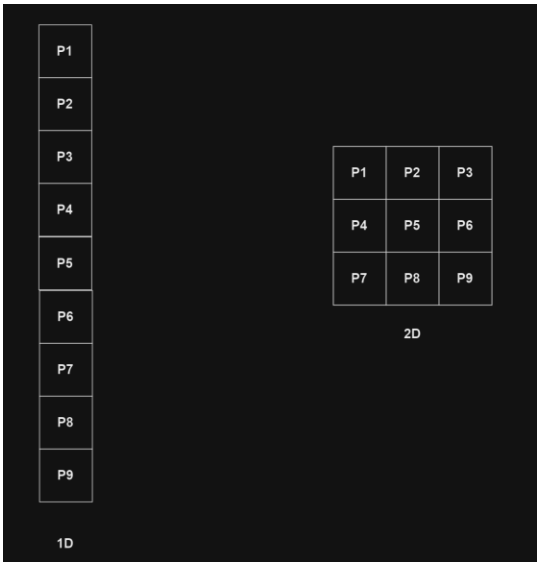
MPI CARDIACSIM – BEYZA ÇAVUŞOĞLU

Code Explanations:

As we know, the structured grid is partitioned such that each partition goes to a unique process. I chose the one below.



1D Implementation:

First, the E, E_prev and R are distributed to processes local memory as local arrays. This is done by MPI_Scatter as below. Also, after all the processes are done, they will be gathered by MPI_Gather in the master process rank 0.

```
void distribute_to_processes(double **source, double **dest, int size_x, int size_y, int rank, int m, int r
{

    int area_size = size_x * size_y;
    MPI_Scatter(source[0], area_size, MPI_DOUBLE, dest[1], area_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    //if(rank == 0){
        // printf("process : %d", rank);
//    printf("sizex: %d, sizey: %d ",size_x, size_y);
    //print2D(dest, m+2, n+2);
    //}
}

// reverse of the distribution function above
void gather_from_processes(double**dest, double **source, int size_x, int size_y, int m, int n)
{
    int area_size = size_x * size_y;
    MPI_Gather(dest[1], area_size, MPI_DOUBLE, source[0], area_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

```
// Allocation for the local data of processes
local_E = alloc2D(n + 2, m + 2);
local_E_prev = alloc2D(n + 2, m + 2);
local_R = alloc2D(n + 2, m + 2);
```

```
// After the allocation for processes, data should be sent to processes memory using Scatter
distribute_to_processes(E, local_E, size_X+1, size_Y+2, rank, m, n);
distribute_to_processes(E_prev, local_E_prev, size_X+1, size_Y+2, rank, m, n);
distribute_to_processes(R, local_R, size_X+1, size_Y+2, rank, m, n);
```

```
int rank_north = rank - 1;
int rank_south = rank + 1;
int rank_west  = rank;      // itself because i choosed to use 1d vertical process geometry, not gonna be used tho
int rank_east  = rank;
```

```
355   // Necessary for MPI_Waitall
356   MPI_Status arr_status[4];
357
358
359   while (t<T) { //while time set by user is not exceeded
360   // Ghost cell communications between neighbor processes should happen here after allocation.
361
362   int request = 0;
363
364   // Receiving from the north neighbor
365   if(rank != 0) // check if its not the top most one
366     MPI_Irecv(local_E_prev[0], size_X+2, MPI_DOUBLE, rank_north, 0, MPI_COMM_WORLD, &arr_request[request++]);
367
368   // Receiving from the south neighbor
369   if(rank != (num_processes - 1)) // check if its not exceeding the south most one
370     MPI_Irecv(local_E_prev[n-1], size_X+2, MPI_DOUBLE, rank_south, 0, MPI_COMM_WORLD, &arr_request[request++]);
371
372   // Sending to north neighbor
373   if(rank != 0)
374     MPI_Isend(local_E_prev[1], size_X+2, MPI_DOUBLE, rank_north, 0, MPI_COMM_WORLD, &arr_request[request++]);
375
376   // Sending to south neighbor
377   if(rank != (num_processes - 1))
378     MPI_Isend(local_E_prev[n-2], size_X+2, MPI_DOUBLE, rank_south, 0, MPI_COMM_WORLD, &arr_request[request++]);
379
380   MPI_Waitall(request, arr_request, arr_status);
```

```
404     MPI_Barrier(MPI_COMM_WORLD);
405     gather_from_processes(E_prev,local_E_prev ,size_X+1, size_Y+2, m, n);
```

1D + OPENMP:

Inside the simulate() function, the nested loops are parallelized by collapse 2.

```
// OPENMP IMPLEMENTATION
#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));
  }
 }
}
```

```
int i,j;
// Initialization
#pragma omp parallel
{
  #pragma omp  for collapse(2)
for (j=1; j<=m; j++)
    for (i=1; i<=n; i++)
      E_prev[j][i] = R[j][i] = 0;

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

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

}
```

Also, in the main() function, initialization of E,E_prev and R can be parallelized by OPENMP.

2D:

The rank for the 4 sides is different from 1d. In 1D, normally north rank was (rank - 1) because of vertical 1d process geometry, but here px should be subtracted to jump to the north neighbor process by passing the next ones in the same line of process. There was also not west and east because in 1D vertical, we don't need the ghost cells and communication from these sides, only north and south is enough.

```
320     int rank_north = rank - px;
321     int rank_south = rank + px;
322     int rank_west = rank + 1;
323     int rank_east = rank - 1;
```

In 2d implementation, now the difference is the communication with 4 sides, not just north and south. MPI_Isend and MPI_Irecv are changed accordingly as below:

```
348     // North neighbors - receive and send
349     if (pos_Y != 0)
350     { // check if its the top-most one
351     MPI_Irecv(local_E_prev[0], size_X+2, MPI_DOUBLE, rank_north, 0, MPI_COMM_WORLD, &arr_request[request++]);
352     MPI_Isend(local_E_prev[1], size_X+2, MPI_DOUBLE, rank_north, 0, MPI_COMM_WORLD, &arr_request[request++]);
353     }
354
355     // South neighbors - receive and send
356     if (pos_Y != ((num_processes - 1) / px))
357     { // check if its not exceeding the south-most one
358
359     MPI_Irecv(local_E_prev[n-1], size_X+2, MPI_DOUBLE, rank_south, 0, MPI_COMM_WORLD, &arr_request[request++]);
360     MPI_Isend(local_E_prev[n-2], size_X+2, MPI_DOUBLE, rank_south, 0, MPI_COMM_WORLD, &arr_request[request++]);
361     }
362
363     // West neighbors - receive and send
364     if (pos_X != 0)
365     { // check if not the west most one
366
367         for (int i=0; i < size_Y; i++){
368             west_2[i] = local_E_prev[i+1][1];
369         }
370
371         MPI_Irecv(&west[0], size_Y, MPI_DOUBLE, rank_east, 0, MPI_COMM_WORLD, &arr_request[request++]);
372         MPI_Isend(&west_2[0], size_Y, MPI_DOUBLE, rank_east, 0, MPI_COMM_WORLD, &arr_request[request++]);
373     }
374     // east neighbors - receive and send
375     if (pos_X != ((num_processes - 1) % px))
376     {
377     for (int i=0; i < size_Y; i++){
378         east_2[i] = local_E_prev[i+1][size_X];
379         }
380         MPI_Irecv(&east[0], size_Y, MPI_DOUBLE, rank_west, 0, MPI_COMM_WORLD, &arr_request[request++]);
381         MPI_Isend(&east_2[0], size_Y, MPI_DOUBLE, rank_west, 0, MPI_COMM_WORLD, &arr_request[request++]);
382     }
```

```
384
385     // For the west part
386     if (pos_X != 0)
387     {
388         for (int i = 0; i < size_Y; i++)
389         {
390             local_E_prev[i + 1][0] = west[i];
391         }
392     }
393     // For the east part
394     if (pos_X != ((num_processes - 1) % px))
395     {
396         for (int i = 0; i < size_Y; i++)
397         {
398             local_E_prev[i + 1][size_X + 1] = east[i];
399         }
400     }
401
402     MPI_Waitall(request, arr_request, arr_status);
```

EXPERIMENTAL RESULTS:

STUDY 1 & STUDY 2) Strong Scaling with 1D and 2D Process Geometry

Process Count	Serial	1D Gflops Rates	2D Gflops Rates
1	4.98908	3.6012	3.47139
2		3.10336	2.64493
4		0.791518	1.55737
8		0.766741	1.02074
16		0.439739	0.521337

The performance of a computing device is often measured in terms of GFLOPS (GigaFLOPS or billions of floating-point operations per second).

MPI is most beneficial for large-scale parallel computing. If the problem size is too small, the overhead of setting up and managing MPI processes might overshadow the benefits of parallelization. This might be the reason why MPI 1D and MPI 2D are not performing better than the serial code. Because MPI involves communication between different processes, this can introduce overhead. The frequent communication or the communication patterns may not be optimized which can lead to performance degradation.

The optimal processor geometry is 2D except for 2 processes.

STUDY 3)

In this part, the communication is disabled and with different process geometries the test Is conducted in 1D MPI without OPENMP. The test is done only on 16 cores with process geometries (x,y) = (1,16) | (2,8) | (4,4) | (8,2) | (16,1). The results as shown as below:

	MPI 16 PROCESSES NO COMMUNICATION				
size:	X=16 Y=1	X=1 Y=16	X=4 Y=4	X=8 Y=2	X=2 Y=8
n = 64	0.442974	0.418056	0.420916	0.430897	0.421685
n = 32	0.0980442	0.098764	0.10422	0.102897	0.10239

When the problem size is shrinked, the GFlops also is reduced. The best performance is with X = 8 and Y = 2 or the viceversa configuration is also doing good results.

In MPI, communication between processes can cause overhead. If the processes are arranged such that communication is minimized, it can lead to better performance. For the geometries (2,8) and (8,2), there might be a balance achieved between computation and communication, leading to optimal results. Also, there might be good load balancing.

STUDY 4) MPI + OPENMP:

In this experiment, different configurations are tested as below:

Gflops				
MPI 1 + OPENMP 16	MPI 2 + OPENMP 8	MPI 4 + OPENMP 4	MPI 8 + OPENMP 2	MPI 16 + OPENMP 1
0.491228	0.828704	0.660277	0.622573	0.524183

The best configuration is MPI 2 processes and 8 OPENMP threads with higher Gflops. There can be several reasons behind this results. In MPI, processes communicate with each other, and there is overhead associated with inter-process communication. If the problem size is not large enough to justify the use of multiple MPI processes, the communication overhead may dominate the computation time, leading to inefficient performance. It was already shown in Study 1 that MPI overhead can even make the case worse than CPU implementation.