## Section (1) - Development Flow

### Q1.a)

Our program implements the Aliev-Panfilov cardiac simulation, a mathematical model with partial differential equations used to simulate the electrical activity of cardiac tissue. We also leverage parallel computing by utilizing the Message Passing Interface (MPI), enabling multiple processes to communicate and cooperate in a distributed computing environment. Our implementation has two main parts: initialization and computation with communication:

**Initialization:**

Given a matrix of dimension m x n and processor geometry py x px, we divide the matrix into blocks as evenly as possible by giving processors on the left/top one more column/row than the rest of the processors. Since we allow submatrices to exchange the data on their boundary cells to do the differentiation in the simulation, we used the extra cells to be the buffer for receiving data for computation on boundary cells. Here's an illustration when m=n=5 and py=px=3:

| | | |
|---|---|---|
| 2 x 2 | 2 x 2 | 2 x 1 |
| 2 x 2 | 2 x 2 | 2 x 1 |
| 1 x 2 | 1 x 2 | 1 x 1 |

We then assumed that only the processor with rank 0 has the information on the data matrices, and we distributed only the assigned blocks to other processors by calling Isend and Irecv.

**Computation with communication:**

From the rank of the current processor, we can calculate the rank of the four neighbors (or -1 if on boundary). Then in each iteration, we can send and receive the data on the ghost cells in the top/bottom/left/right direction to/from the neighbors to complete the information needed for calculation. We used MPI_Type_vector() for efficiently using message buffers for ghost cells. While waiting for the communication to complete, we have a flag to control whether we do part of the computation that does not need ghost cells while waiting or not. We also have a flag to control whether we fuse the loops to compute PDE and ODE to one or not. We have also implemented vectorization for the computation. The comparison will be discussed in more detail later.

After all iterations, we apply MPI_Reduce to aggregate the L2 and Linf norm from all processors, which will act as a metric for whether the resulting matrix is correct or not.

**Q1.b)**

1. We first implemented the init() method in helper.cpp. In order to use MPI, we need to specify the right amount of data each processor should handle. Given the size of the whole matrix, m*n, and the number of processors, py*px, we should allocate three submatrices E, E_prev, R, each of size (ceiling(m/py)+ 2) * (ceiling(n/px)+2), for each processor, except that for processor 0, we allocate matrices of size (m+2)*(n+2)) to store the initial data.

2. We then tried using uneven submatrix sizes with their difference in side length no more than 1. As demonstrated in the previous part, the submatrix side lengths should add up exactly to be the side length of the original matrix. In this way, we can avoid having redundant memory allocation and let our cache do more useful storage. Moreover, it helps us get rid of divergent index expressions and achieve accurate results for Linf and L2.

3. For rank0, we tried accessing the whole n*n matrix when doing computation (it only needs the top left corner ny*nx) while later realized that the memory offset of each column would be two large to be cache friendly. Hence, we aligned the memory so that we would only access the ny*nx as necessary, thereby treating it as no different from other ranks when doing computation.

3. Next, we used MPI_Isend() to send the initialization information about these matrices from processor 0 to other processors and MPI_Irecv() to receive the information by the other processors. The two methods are non-blocking so that when used in a loop, a previous iteration would not block a later iteration. Then, we used MPI_Waitall() to let all sending and receiving to complete before we proceeded to the computing phase.

4. We used to declare some local arrays as a buffer to store the data processor 0 needs to send to the other processors. However, we realized that these statically allocated memory are not 'safe' to use as they are part of the stack which has limited size. Then we learned that there is a custom MPI data type that can be used as a sparse block with a certain stride length. By defining a new MPI_Datatype that can capture a sub-block of the data matrix with custom block size and stride, we don't need to pack the data by ourselves. We just need to pass this data type to send or receive.

5. In the solve() function, we devised 4 blocks of code to handle for each iteration the communication among ranks to exchange data for each one of the four boundary cells of a submatrix. In each block of code, we had one send call and one receive call. Before, we had a wait call for each pair of send and receive, but later we used one wait call to synchronize all 4 pairs of send/receive calls to reduce waiting overhead.

6. We compared the fused version that computes E and R in a single nested loop and the non-fused version that computes those in two nested loops of the code and saw no significant difference. Therefore, we stick to the fused version.

7. We also tried to do some part of the computation while waiting for synchronization. Computation of inner cells in each block doesn't depend on the communication with other blocks. So we could do such computation when sending and receiving ghost cells. We thought this may reduce communication overhead and hence improve performance. The single wait call now can go after the computation of the inner cells of each submatrix block but before that of its cells. This gives us a decent boost in performance.

8. To further improve the performance, we implemented vectorization by hand rather than relying on the vectorization by the C++ compiler. We used Intel SSE2 intrinsics with the __m128d data type. The operations we used included: _mm_loadu_pd(), _mm_loadl_pd(), _mm_loadh_pd(), _mm_storeu_pd(), _mm_storel_pd(), _mm_storeh_pd(), _mm_add_pd(), _mm_sub_pd(), _mm_mul_pd(), _mm_div_pd(), _mm_set1_pd(), and _mm_set0_pd().

Q1.c) If you improved the performance of your code in a sequence of steps, document the process. It may help clarify your explanation if you summarize the information in a list or a table. Discuss how the development evolved into your final design.

We measured our code’s performance on 16 cores on a shared node before we are determined with the most optimized configurations. Although almost half of our time was devoted to debugging the computation steps to get the correct Linf and L2 results, we did have three major improvements as shown by the table below:

| Step | Time | GFLOPS |
|---|---|---|
| Baseline | 0.3166s | 113.2 |
| Reducing #wait call | 0.2874s | 124.7 |
| Overlapping commutation with computation | 0.2309s | 155.2 |
| Vectorization in computation by hand | 0.1936s | 185.2 |

Our final implementation involves keeping all the optimized configurations as listed. Reducing wait calls allows the messages from boundary ranks to be received by the center rank in arbitrary order, thereby having lower latency than maintaining some order in separate wait calls. Overlapping communication hugely reduces the MPI overhead as we can do the computation while waiting for all ghost cell data to be sent or received. Vectorization gives as a final bost on the computation itself, which is probably independent of the MPI calls. We may later analyze how vectorization may be benefitted by MPI, during message buffering and transmitting for example. We think a compiler-level vectorization may optimize the computation similarly.

## Section (2) - Result

### Q2 (a):

Single processor:

| Type | Time | GFLOPS |
|---|---|---|
| Original code | 3.64s | 9.84 |
| Vectorized without MPI | 2.701 | 13.27 |
| Vectorized with MPI | 2.716 | 13.19 |

As we can see, the vectorization gives us a quite favorable boost in serial performance. The MPI overhead is negligible as expected. The time is not counting the MPI initialization part.

1 - 16 processors:

| Cores | Geometry | GFLOPS | Time (comm) | Time (no comm) | MPI overhead |
|---|---|---|---|---|---|
| 1 | 1*1 | 13.19 | 2.716s | 2.701s | 0.55% |
| 2 | 1*2 | 25.84 | 1.387s | 1.334s | 3.82% |
| 3 | 1*3 | 35.8 | 1.006s | 0.998s | 0.79% |
| 4 | 1*4 | 50.88 | 0.7044s | 0.6706s | 4.79% |
| 5 | 1*5 | 62.3 | 0.5757s | 0.5243s | 8.92% |
| 6 | 1*6 | 75.5 | 0.4747s | 0.4373s | 7.87% |
| 7 | 1*7 | 86.8 | 0.4128s | 0.3771s | 8.64% |
| 8 | 1*8 | 99.52 | 0.3601s | 0.3357s | 6.77% |
| 9 | 1*9 | 109 | 0.3297s | 0.3058s | 7.24% |
| 10 | 1*10 | 119 | 0.3003s | 0.2787s | 7.19% |
| 11 | 1*11 | 132 | 0.2716s | 0.2532s | 6.77% |
| 12 | 1*12 | 146 | 0.2460s | 0.2327s | 5.40% |
| 13 | 1*13 | 151 | 0.2367s | 0.2170s | 8.32% |
| 14 | 1*14 | 160 | 0.2238s | 0.2033s | 9.15% |
| 15 | 1*15 | 164 | 0.2181s | 0.1906s | 12.61% |
| 16 | 1*16 | 185.2 | 0.1936s | 0.1659s | 14.30% |

The above table is the result of running on a compute node on Expanse, with side length n = N0 = 800, number of iterations i =2000. The calculated values for Linf and L2 are 0.992479 and 0.563440, respectively. The MPI overhead is calculated by (t_comm - t_no_comm) / t_comm.

There seems to be a trivial MPI overhead when measured using 1 processor, which can be due to the variation across measurements. We kept the 1D geometry of the processors the same, i.e. px=1 and py=numProcessors, and found out that when their number increases, the MPI overhead also scales. For example, going from 8 processors to 16 processors increases our MPI overhead by a factor of ~2. This is expected as when the processing units grow, the message latency and network bandwidth may increase proportionally. Nevertheless, we got linearly growing computation performance, which may hint us that these geometries are all cache friendly.

**Q2.b)**

The following table is the result of running on a compute node on Expanse, with n = N0, i =2000:

| Cores | Geometry | Time (comm) | Time (no comm) | GFLOPS ( comm) | GFLOPS (no comm) |
|---|---|---|---|---|---|
| 1 | 1*1 | 2.716s | 2.701s | 13.19 | 13.27 |
| 2 | 1*2 | 1.387s | 1.334s | 25.84 | 26.87 |
| 4 | 1*4 | 0.7044s | 0.6706s | 50.88 | 53.45 |
| 8 | 1*8 | 0.3601s | 0.3357s | 99.52 | 106.8 |
| 16 | 1*16 | 0.1936s | 0.1659s | 185.2 | 216 |

The scaling from 1 core to 2 cores is (25.84/13.19)/2 =98%, almost 100%.

The scaling from 2 cores to 4 cores is (50.88/25.84)/2 =98.5%, almost 100%.

The scaling from 4 cores to 8 cores is (99.52/50.88)/2 =97.8%, almost 100%.

The scaling from 8 cores to 16 cores is (185.2/99.52)/2 =93%.

Hence, our code scales almost linearly with the number of cores for 1D geometry up to 16 cores. This shows a strong scaling as we maintain our n constant. To calculate speedup and efficiency of the parallel program with regard to the serial one, we have:

2 core: speedup = 25.84/13.19 = 1.959, efficiency = 1.959/2 = 0.9795

4 cores: speedup = 50.88/13.19 = 3.857 , efficiency = 3.857/4 = 0.964

8 cores: speedup = 99.52/13.19 = 7.545 , efficiency = 7.545/8 = 0.943

16 cores: speedup = 185.2/13.19 = 14.041 , efficiency = 14.041/16 = 0.878

The efficiency drops as we have more network time with more cores, which increases the overall overhead.

**Q2.c)**

| Cores | Geometry | Time (comm) | Time (no comm) | GFLOPS ( comm) | GFLOPS (no comm) | MPI overhead |
|---|---|---|---|---|---|---|
| 16 | 1*16 | 44.03s | 41.58s | 206.1 | 218.2 | 5.6% |
| | 2*8 | 44.01s | 41.71s | **206.1** | 217.5 | 5.2% |
| 32 | 1*32 | 26.27s | 20.98s | 345.4 | 432.4 | 20.1% |
| | 2*16 | 25.59s | 21.24 | 354.5 | 427 | 17.0% |
| | 4*8 | 24.36s | 20.56 | **372.4** | 441.3 | 15.6% |
| 64 | 1*64 | 16.28s | 10.85s | 557.2 | 836.3 | 33.4% |
| | 2*32 | 12.3s | 10.5 | 737.4 | 863.6 | 14.6% |
| | 4*16 | 12.1s | 10.33s | **749.6** | 878 | 14.6% |
| | 8*8 | 12.73s | 10.53s | 712.7 | 861.4 | 17.3% |
| | 16*4 | 13.71s | 10.91s | 661.9 | 831.3 | 20.4% |
| 128 | 2*64 | 8.17s | 6.35s | 1110 | 1426 | 22.2% |
| | 4*32 | 7.46s | 6.26s | 1216 | 1448 | 16.1% |
| | 8*16 | 7.42s | 6.32s | **1222** | 1435 | 14.8% |
| | 16*8 | 7.74s | 6.38s | 1171 | 1422 | 17.5% |
| | 32*4 | 8.04s | 6.52s | 1127 | 1391 | 18.9% |

Linf and L2 for N1=1800 at 100000 iterations: 9.83616e-01 7.55135e-01

We now calculate the scaling:

From 16 cores to 32 cores: (44.01/24.36)/2=90.4%

From 32 cores to 64 cores: (24.36/12.1)/2=100.7%

From 64 cores to 128 cores: (12.1/7.42)/2=81.5%

Given the variation of the measured time across experiments, we still consider our program to be strongly scaled from 16 to 128, with the expectation that the scaling will be decreased at the end.

Cost of communication:

**Q2.d)** https://forms.gle/fbUd5FJeHUBHBNf89

Best geometry and performance:

| Cores | Geometry | GFlops |
|---|---|---|
| 128 | 4*32 | 200 |
| 256 | 128*2 | 564 |

|  |  |  |
|---|---|---|
| 192 | 6*32 | 402 |
| 384 | 4*96 | 2009 |

The scaling from 128 to 192 is 134%, the scaling from 192 to 256 is 105%, and the scaling from 256 to 384 is 237%.

Search for the best geometry when p=128:

| Geometry | GFlops | MPI overhead |
|---|---|---|
| 1*128 | 198 | 2.6% |
| 2*64 | 199 | 2.5% |
| 4*32 | 200 | 2.5% |
| 8*16 | 199 | 3.4% |

Search for the best geometry when p=256:

| Geometry | GFlops | MPI overhead |
|---|---|---|
| 16*16 | 510 | 82.9% |
| 32*8 | 541 | 82.7% |
| 64*4 | 519 | 83.1% |
| 128*2 | 564 | 65.9% |

Search for the best geometry when p=192:

| Geometry | GFlops | MPI overhead |
|---|---|---|
| 6*32 | 402 | 84.2% |
| 12*16 | 400 | 84.3% |
| 16*12 | 397 | 54.0% |
| 24*8 | 399 | 84.1% |

Search for the best geometry when p=384:

| Geometry | GFLOPS | MPI overhead |
|---|---|---|
| 1*384 | 1653 | 49.5% |
| 2*192 | 1749 | 42.5% |

| | | |
|---|---|---|
| 3*128 | 1616 | 57.0% |
| 4*96 | 2009 | 47.2% |
| 6*48 | 1771 | 52.7% |
| 8*48 | 1847 | 53.7 |
| 16*24 | 1250 | 27.2% |
| 24*16 | 1337 | 65.6% |
| 32*12 | 1104 | 69.3% |
| 48*8 | 1080 | 72.7% |
| 64*6 | 1063 | 71.7% |
| 96*4 | 1080 | 72.5% |
| 128*3 | 1118 | 69.0% |

## Q2.e)

We noticed a significant increase in overhead when using p <= 128 compared with using p > 128, by comparing the average overhead across geometries. We also saw a huge variation in the overhead for p > 128. For example, when p = 384, the minimum overhead is 27.2 when the geometry of the processors is 16*24, while the maximum overhead is 72.7 when the geometry is 48*8. The lowest overhead hints us that a row size of 384/24=16 is more cache friendly to store the message buffers, though it is not very efficient in computation. The highest overhead is at a row size of 384/8=48 which is two much for the cache to handle. However, the MPI overhead is not proportional to the overall computation time, as we interleaved computation with communication.

## Q2.f)

Cost of computation:

| Cores | Computation Cost = #cores x computation time |
|---|---|
| 128 | 128*71.81s = 9191.68s |
| 256 | 256*25.41s = 6504.96s |
| 192 | 192*35.87s = 6887.04s |
| 384 | 384*7.731s = 2968.7s |

The optimal cost is achieved by 384 cores. Even though it uses more processors, the computation time is much smaller due to the strong scaling, which results in the cheapest total cost.

## Section (3) - Determining Geometry

### Q3.a)

Here are the top-performing (top 10%) geometries at N1 for p=128:

| Geometry | Time (with communication) | Time (without communication) | GFLOPS | MPI overhead |
|---|---|---|---|---|
| 2*64 | 8.17s | 6.35s | 1110 | 1426 |
| 4*32 | 7.46s | 6.26s | 1216 | 1448 |
| 8*16 | 7.42s | 6.32s | **1222** | 1435 |
| 16*8 | 7.74s | 6.38s | 1171 | 1422 |
| 32*4 | 8.04s | 6.52s | 1127 | 1391 |

### Q3.b)

Using the model described in class, we can model the communication time for each processor at each iteration of the simulation by $t = \alpha + \beta_{\infty}^{-1}(8(\frac{n}{px} + \frac{m}{py}))$, where $\alpha$ is the message latency and $\beta$ is the peak bandwidth, and the message length is $8(\frac{n}{px} + \frac{m}{py})$ because we are both sending and receiving 2 messages from each of the four directions. The computation time is proportional to $(\frac{n}{px} * \frac{m}{py})$.

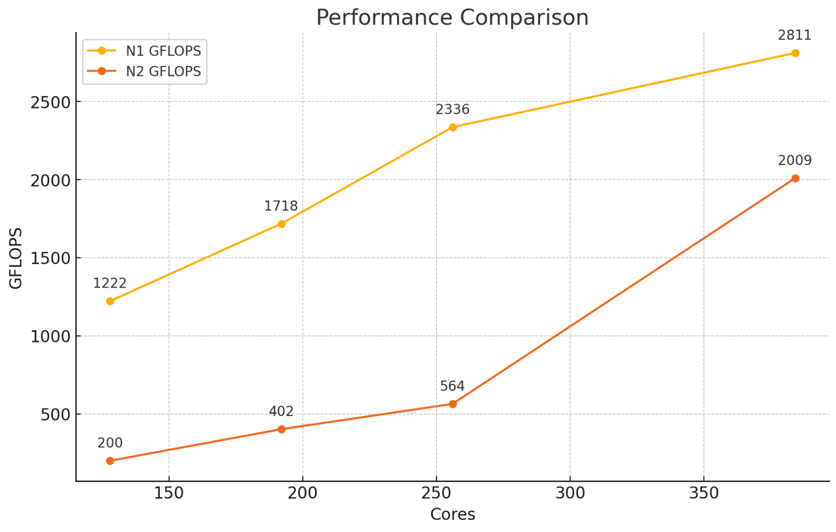
Hence, when px and py are closer to each other, neither of $\frac{n}{px}, \frac{m}{py}$ are huge, which will reduce the communication overhead and speed up the process, which explains why none of the numbers in the table is too large or too small. Also, when px is lower, each sub-block will have longer rows and shorter columns, which will make use of cache locality to enhance performance, which explains why 8*16 geometry is slightly better than other choices. To convince ourselves, there seems to be a balance between making nx large to approach ny for efficient network bandwidth usage, and making ny larger to take advantage of the cache locality when buffering messages and doing computation row-wise.

## Strong and Weak Scaling (4)

### Q4.a)

| Cores | N1 GFLOPS | N2 GFLOPS |
|---|---|---|
| 128 | 1222 | 200 |
| 256 | 2336 | 564 |
| 192 | 1718 | 402 |

|  |  |  |
|---|---|---|
| 384 | 2811 | 2009 |



## Q4.b)

We notice that for p=256 our code performs much better when n=N1 than when n=N2. This may due to a large n increase the size for each submatrix, where originally a row or a message buffer able to be stored in cache now no longer fits in cache (higher cache miss rate). Also, during computation, the memory address offset of accessing two cells adjacent in a column is larger, making the accessing pattern less efficient.

## Section (5) Extra Credit

Early submission - 5pts

## Section(6) - Potential Future work

We didn't have a chance to implement the plotting on a multi-core (MPI) application by collecting all computed cell data in rank 0. If we had more time, it would be very interesting to see how the cells transmit information in the mesh in different iterations. It may also be a choice to run with p > 384 if possible to study how the MPI overhead behaves in that case.

**Section (7) - References (as needed)**

[1] MPI tutorial:
https://mpitutorial.com/tutorials/

[2] MPI guide:
http://cseweb.ucsd.edu/~baden/Doc/mpi.html

[3] MPI Documentation:
https://www.mpich.org/static/docs/v4.1/www3/

[4] A Paper about MPI and ghost cells:
https://fredrikbk.com/publications/ghost_cell_pattern.pdf

[3] Expanse Webinar:
https://www.sdsc.edu/event_items/202202_ExpanseWebinar-M.Thomas.html

[4] Intel Intrinsic Reference
https://www.intel.com/content/dam/develop/external/us/en/documents/18072-347603.pdf