Programming Assignment 3: Aliev-Panfilov Cardiac Simulation

Colin Drewes (PID: A15616787)

Zihao Kong (PID: A15502295)

Abstract

In this assignment we have optimized the Aliev-Panfilov Cardiac Simulation [Rub99] with a message passing implementation. This allows the simulation to run on an arbitrary number of cores. In our tests we use up to 480 cores to get a performance of 1300 Terraflops on the San Diego Super Computer Cluster's Comet machine. Our message passing interfaces uses "ghost" tiles to communicate relevant information between different processors.

1 Development Flow

1.1 a

1.1.1 Processor Subdivision

The first step in our message passing solution to Aliev-Panfilov Cardiac Simulation is to subdivide the input grid based on the number of sub-processors. We let N refer to the size of the input grid we are simulating upon, px be the number of processors specified in the x direction of our grid, and py the number of processors in the y direction. Then, we form px*py tiles of size N/px x N/py. This creates a grid of tiles, where each tile represents a different processors. Processors are labeled by their rank, which is 0 through px*py - 1. This grid of px*py tiles is treated as a row major matrix for the processors. This means that the 0th element is processor 0. Then the next px - 1 elements will be processors 1,2,3. . . ,px-1, at which point we wrap around to the next layer of py. However, it is not always the case, or even often the case that N/px or N/py results in an integer. So, we need to have certain tiles larger than others to adjust for this. From the design specification document there should be no processor which has greater than 1 row or column than any of the other processors. We determine which processor tiles need to have added columns or rows based on the values of N % px and N % py. Then, the first (N % px) columns of *processors* will have an extra column added to them and the first (N % py) rows of *processors* will have an extra row added to the tile. [Chi20]

In contrast to the original naive implementation we no longer pad the tiles of the matrix with an extra layer on each side. We instead take an alternate strategy. For each processor tile we allocate 4 separate vectors which match the length of y direction of the tile, and 4 separate vectors which match the length of x direction of the tile. These will serve doubly: first they will be used to pass the outer layers of the processor tile matrix E to other processors, and second to simulate the boundary conditions on the outer edges of the computation which we still need to simulate. At this point we have a set of processor which refers to the subdivisions of E and R (which is a grid filled with the following initial conditions).

1.1.2 Initial Conditions

From the project specification we must have processor 0 be responsible for distributing the entirety of the *E_prev* grid and *R* grid initial conditions. Each processor has an associated sub-tile of *E_prev* and *R*. Processor 0 calculates the initial conditions for each of the other px * py - 1 processors. To accomplish this within the processor 0 thread we iterate through each of the other processor indices. For each of them we determine the absolute position of each of the indices within the *E_prev* and *R* sub-tiles. The absolute position is effectively just what the index would be if the sub-tiles were viewed as the original size N tile. This allows us to create a new sub-tile for *E_prev* and *R* from within processor 0 which can be sent to each of the other processors with the non-blocking *MPI_Isend*. By using the absolute position in the calculation of the send sub tiles we set the the right half of the *E_prev* to be all ones and the bottom half of rows of *R* to be all ones. Processor 0 manually copies the sub-tiles generated for *E_prev* and *R*, which does not require any MPI operations. Finally, all the processors besides 0 can received the *E_prev* and *R* sub-tiles with non-blocking *MPI_Irecv*. The data is just directly received into the processors local *E_prev* and *R* sub-tiles. The initial conditions have now been configured.

1.1.3 PDE and ODE Solving

The first step to solve the ODE and PDE is to perform the message passing between processors. If the processor has a tile which is falls on the perimeter, then this is the edge boundary condition and we must copy the inner layer to the boundary ghost layer. We simply use the same receive vectors which are used for message passing so that the ODE and PDE calculation is unchanged. We perform the same copying operation as in the initial start code, where the 2nd internal computation layer is copied to the outside for the next computation. Besides the boundary tiles we need to perform the message passing between processors. We copy the edges of the computational block to the send channels of the current processor. We use an *MPI_Isend* to non-blockingly send to the receive channel vectors of the other processors. These sent vectors are then received in the receive vectors in the other processor. What this accomplishes is meshing back together the tiles into the original NxN grid, yet each processor only takes a single tile of this grid. Once this is complete each processor can compute the ODE and PDE within the tile. The matrix R does not need any change from the initial implementation given in the starter code—it does not rely at all on the boundary outside computation box. However, the computation of the E matrix on the processors tile does rely on values outside of the computational box. This computation relies on the value of E_prev in all four cardinal directions. If such a value is not within the processor's computational box, then we look to the receive vector channels from the other processors. This allows us to complete the computation of E as originally performed in the stater code. This is performed for each one of the iterations specified.

1.1.4 Linf and L2 Calculation

The *MPI_Reduce* function is used to collect the values of the Linf and L2 values. We use the *MPI_MAX* function to get the largest value of Linf. The sumSq values are summed across processors before calculating the L2Norm as usual. The stats function must be adjusted for our particular matrix sizes which no longer have the padding added around every side of the matrix. These final values are returned only by processor 0.

1.2 b

It was clear from the start that we needed to use a message passing to some degree in order to get the desired performance over multiple processors. This began with trying to stay with the same padded grid for E and R, and to have the processors work over the same grid matrix. This proved to be unruly to develop with and had challenging array indexing. We also predicted this would have worse cache behavior for a given processor, as the block a particular processors was acting on would not be in the cache at once—or really any block of that. This resulted in switching to each processor instantiating its own tile of E and R. Because each processor has its own tile of E and R, the values of any given processor is working on will be contiguous memory locations. This will result in significantly better cache behavior and it also allows us to manipulate E and R more easily. The initial implementation also padded out the edge of the size NxN grid input with an extra layer on every side. This prove for very cumbersome indexing schemes, and the initial implementation was extremely hard to read as a result. Because we did this extra padding through standalone matrices, our development was drastically faster.

1.3 c

Most all of this information is already covered in the last two question. But, the general approach is as follows:

- Message passing algorithm on complete NxN Grid
- Segment grid and have each processor store its own sub-tile of the original grid
- Remove the padding of matrices and use explicit vectors for message passing and boundary conditions

2 Results

2.1 a

The following tests were performed with 500 iterations and an input grid size of N = 500. This value was chosen to get a good idea of the long term behavior of the respective algorithms. Our implementation was run with the following command in the Bang cluster:

```
mpirun -np 1 ./apf -x 1 -y 1 -n 400 -i 500
```

This resulted in 0.9012 GFlops in performance. We ran the equivalent command on the naive original implementation provided to us. The command was as follows:

```
./apf -n 400 -i 500
```

This resulted in 0.4589 GFlops in performance. Our performance on one core is better than the starter code, this implies that we are doing something "more." To determine the overhead of message passing we will perform an experiment on 8 cores. It seems to be suggested that we should compare the performance of our algorithm on 8 cores to $8 * 0.4589$ (the projected optimal performance of the naive code on 8 cores). However, this analysis would ignore the fact that our single core performance which does not utilize MPI already performs better than the naive code. For this reason to get the actual overhead of MPI we will compare the performance of our algorithm on 8 cores to that of $8 * 0.9012 = 7.2096$. We run the following command:

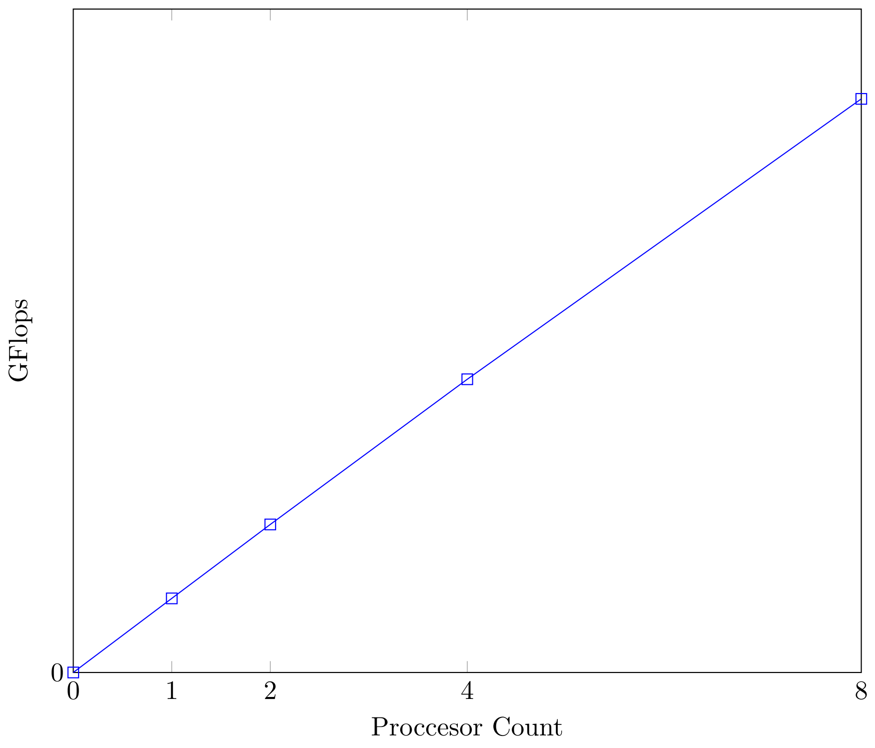
```
mpirun -np 8 ./apf -x 2 -y 4 -n 400 -i 500
```

This gives us a result of 6.93 GFlops performance. This allows us to calculate the overhead created by MPI in the case of 8 cores as 7.1912 - 6.93 = 0.2612 GFlops. This indicates that our MPI code does not add significant overhead in our implementation. In fact, for each core it appears to only be adding .2612/8 = 0.03265 GFlops which is an extremely minimal performance loss.

2.2 b

For these experiments we use 2000 iterations and an input grid of size of 400. We get the following scaling graph:

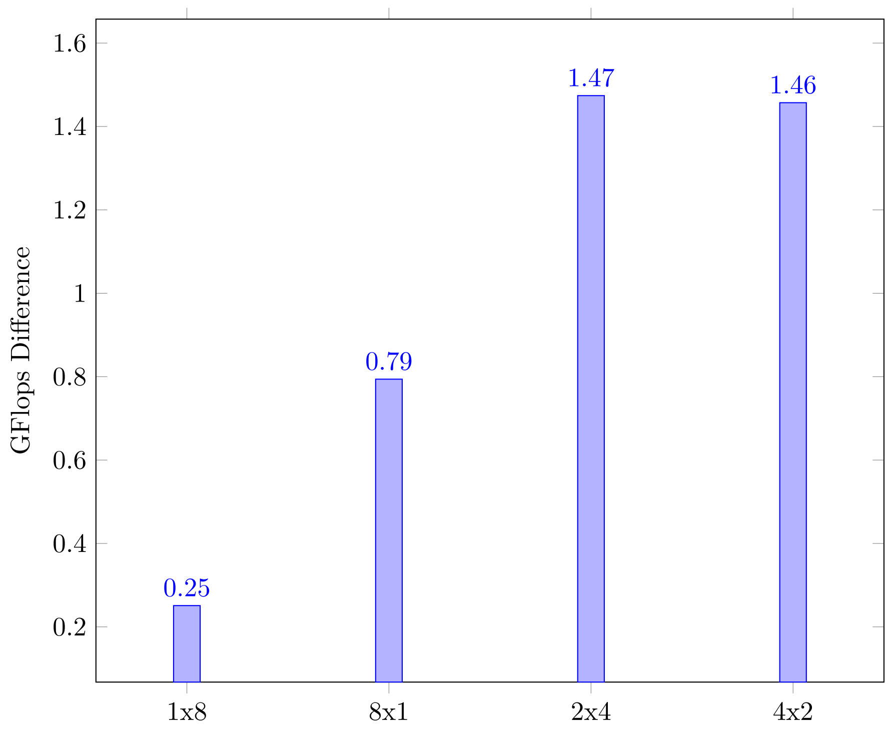
i = 200, N = 400, Strong Scaling Study



The linearity of this is very convincing that Our algorithm scales strongly. The set of points is given by $(0,0)(1,0.8934)(2,1.785)(4,3.536)(8,6.917)$. While not perfect, notably from $3.536 * 2 = 7.072$ which is a little greater than 6.917, but this is certainly with the acceptable error bounds.

2.3 c

To measure communication overhead we will vary the -k flag. This flag controls whether any message passing is used. While we no longer get the correct output when running the algorithm with -k, as the processors are not getting the relevant information to correctly perform the ODE and PDE solves. We will explore different processor geometries and the effect that those have on the communication overhead. The following graph shows the difference in performance (code run without MPI - code run with MPI), where the x axis is the varying processor geometry. For these experiments we use 1000 iterations and an input grid size of n=400.

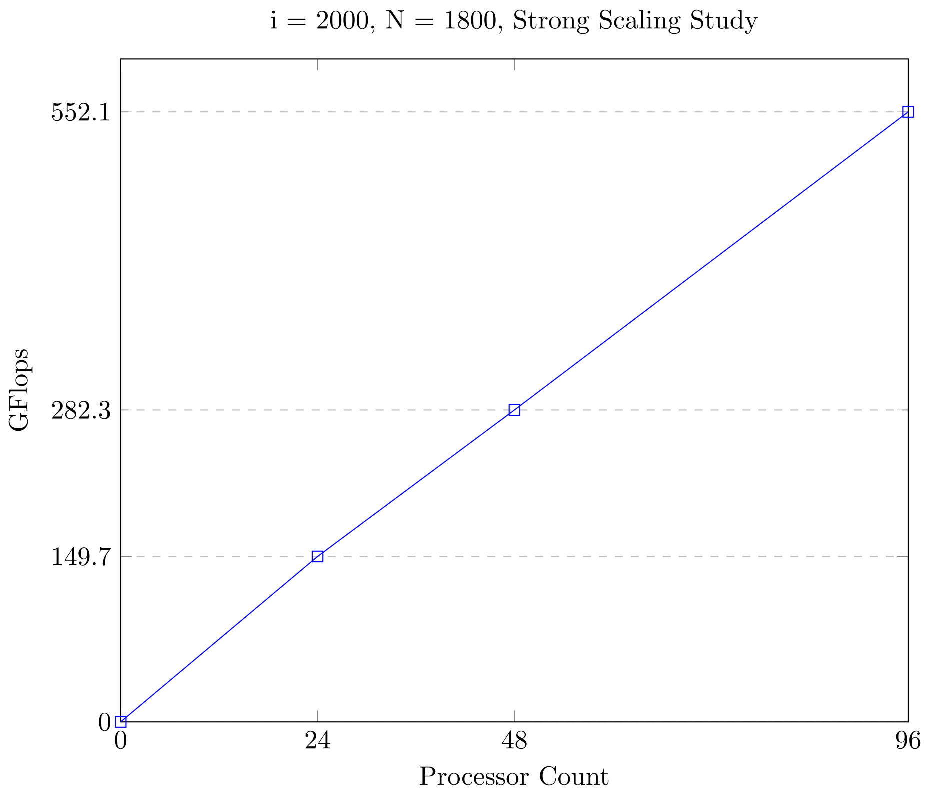


The values are calculated as follows:

1x8 = 7.189 − 6.938 = .251
8x1 = 7.754 − 6.96 = .794
2x4 = 8.408 − 6.934 = 1.474
4x2 = 8.429 − 6.972 = 1.457

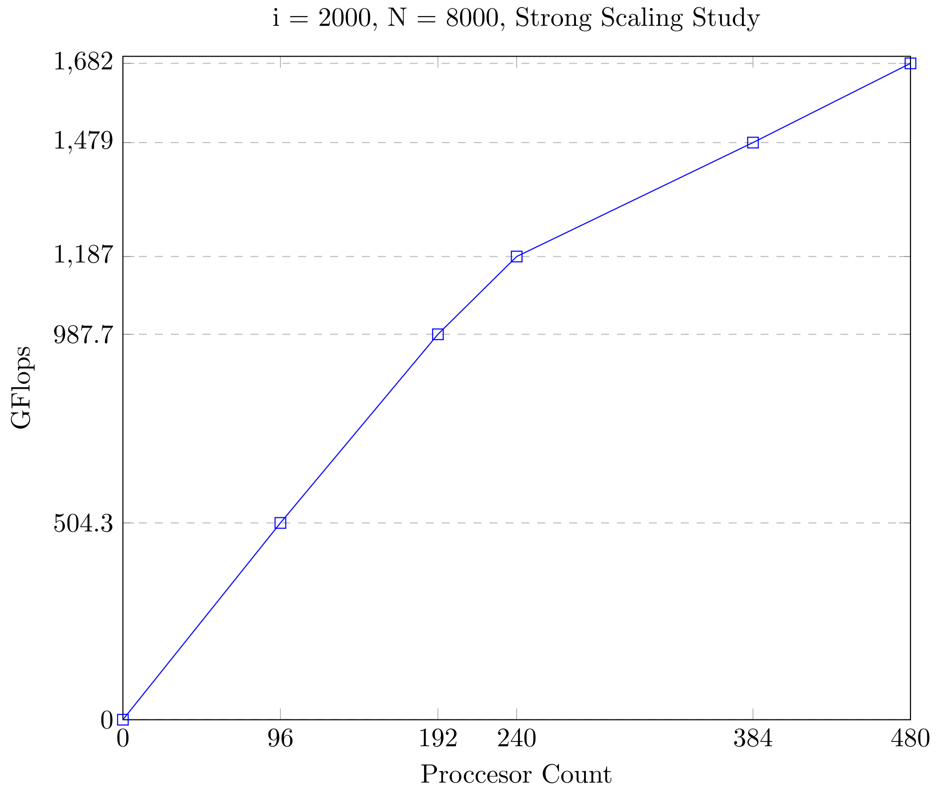
These results are fairly expected. Firstly, we know that MPI will incur some overhead, due to the nature of the copy operations it is performing between ghost tiles between different cores. The 1x8 result is consistent with what we found in part 8 in terms of MPI overhead. We expect that that 1x8 solution would have the best performance compared to the no communication version because the only message passing being performed is between the top an bottom of cells. The top and bottom cells can be updated as a continuous block of memory of E. This will created much better cache behavior, and we use this fact as a motivation for much of our future processor grid sizes. The next largest MPI overhead cost is the 8x1 implementation. This one will only pass messages from left to right. We expect this to be worse than the 1x8 version because the send and receive challenges can not be updated as one contiguous block of E in memory. This brings us to the 2x4 and 4x2 version which exhibit the most overhead when compared to their non-message passing version. In reality the message passing enabled version performs equivalently to all other processor grid sizes, however the no communication version is significantly faster than the others. This is because significantly more MPI operations are being avoided when communication is disabled in the 2x4 and 4x2 block size. There are 14 MPI exchanges for a 2x4 grid as well as a 4x2 grid. This in comparison to the 8x1 and 1x8 version which each have 8 MPI exchanges is a difference of 6, which is non-negligible. However, overall the MPI operations do not introduce great overhead. And, it is important to note that because the initial conditions rely on MPI, the no communications version effectively skip the initial configuration. While we use a fairly large value of i = 1000 iterations which minimizes this, it is still a factor. Nevertheless, on these small scales the MPI does not incur significant overhead.

2.4 d



Note that these are not necessary the best performance for each processor size. However, these are geometry sizes which first meet the performance requirements and second meet the scaling requirements.

2.5 e



Again please note that Note that these are not necessary the best performance for each processor size. However, these are geometry sizes which first meet the performance requirements and second meet the scaling requirements. Also there is a large difference in running the p=96 version with 20 nodes on comet. We do not do this explicitly, but in our tests we "batched" in the 96,192,240,384,480 tests alongside to save on compute time. This radically changes the performance of p=96 compared to running with just 4 nodes of 24 cores as we ordinarily would have needed.

2.6 f

In order to make each run consistent and collect precise data, when we are running experiments with more than four nodes, We measured the computation cost using a bigger problem size with $N = 8000$ and $i = 2000$.

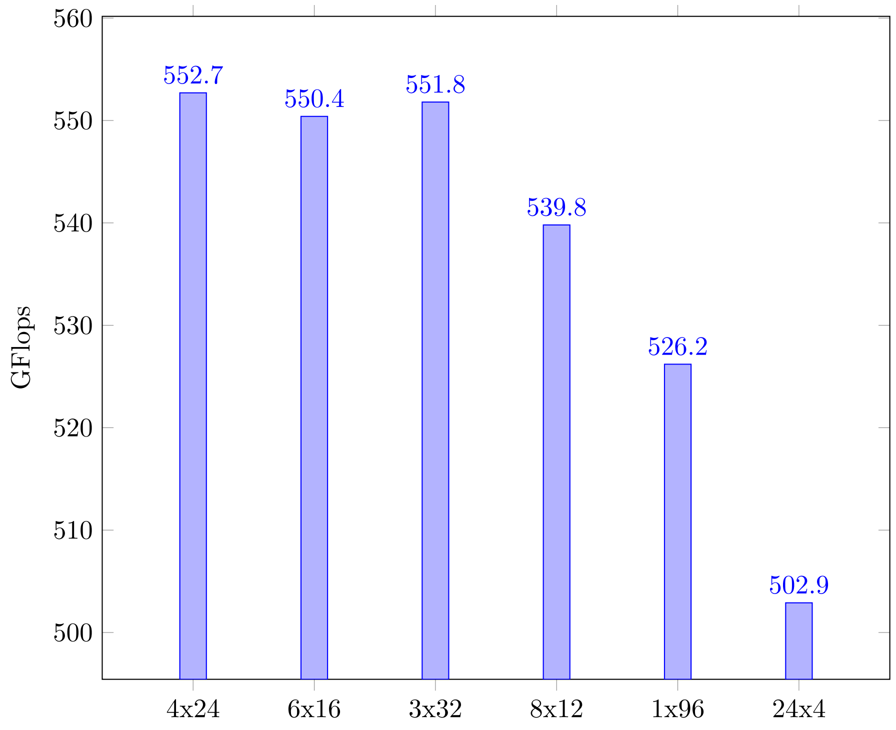
Table 1. *Table of Computation Cost for Various Number of Cores*

Cores	Computation Cost	Optimized Geometry
96	674.4	4*24
192	696.76	4*48
240	715.68	6*40
384	935.42	12*32
480	1060.8	16*30

3 Determining Geometry

3.1 a

All of the experiments were performed with an n of 1800 and 2000 iterations as suggested. The top performing geometry is 4x24 followed closely by 6x16. 4x24 has 552.7 GFlops of performance which implies 10 percent of that is 55.27. All of the values within 10 percent of 4x24 are reported in the graph bellow:



3.2 b

The most general trend in optimal geometries in one we have touched on early: a bias toward larger values of y in the processor grid results in the best performance. Our message passing implementation copy's vertical and horizontal vectors of E to pass to the other processors. Because our matrices are stored as row major, there will be significantly better performance when copying across a row of E rather than a column of E. This is because the entire row can be brought into the cache because it is continuous memory locations, however, columns of E are not continuous in memory. Besides the message passing we also need to copy interior rows and columns of E to the boundary for the next computation as the algorithm description specifies. This explains why we see the 24x4 geometry appearing in the top 10 percent. When we chose processor geometries with larger values of y it improves our message passing ability (due to contiguous memory copies of E), but harms the performance of this edge boundary copying. For example, at 4x24 we have $8 * 1 + 4 * 22 * 2 = 182$ "efficient" copies (or equivalently copies which use contiguous memory locations). However, this also incurs 44 inefficient boundary condition copies. In contrast, a processor geometry of 24x4 will have only $24 * 2 + 2 * 22 * 2 = 136$ "efficient" MPI transactions with only 8 inefficient boundary condition copies. It is this trade off we see balanced in our analysis of the top performing geometries. However, we note that on larger processor sizes (p = 96), for each one value we increase y we get 2 more inefficient boundary copies while we get x more efficient MPI operations. This makes it generally better to favor large values of y for the processor geometry as we have observed in the graph. We have not discussed the cost on computation but only the cost on communication. This is because our computation is relatively unchanged between different processor block geometries. While this may be slightly more inefficient, in our computation we are checking to see if we need some value which is outside of the computational window of E, we then go to the receive channel in that direction for that particular processor. Because we have eliminated the 1 layer padding around the entirety of E, we always have to check if a requested value of E is outside the box. While this is slightly slower, it does imply that after the communication is completed their is no difference in how the computation is performed.

4 Future Work

If we had more stime, we can implement a multi-threaded plotting routine, because the current plotting algorithm only works for single thread, which will slow the entire program down because one thread will be doing more work than the rest. We can also vectorize the inner loop, which will vectorize the inner loop in every MPI process, making the for loop more efficient.

References

[Rub99] Alexander V.Panfilov Rubin R.Aliev. “A simple two-variable model of cardiac excitation”. In: (1999).

[Chi20] Bryan Chin. “Stencil Methods, Aliev Panfilov Method”. In: (2020).