Section 1 - Development Flow

How the program works

This program implements signal propagation in cardiac tissue using the Aliev-Panfilov method. This cardiac simulation has two state variables E (trans-membrane potential) and R (recovery of the tissue) which are represented as two matrices in the program. This method has partial and ordinary differential equations that use a 5-point stencil method to compute new values for each point in matrices E and R. The goal of this program is to efficiently run the simulation on multiple processors using Message Passing Interface (MPI).

Basically, the program divides the matrices E and R into p sub-matrices where p is the number of processors. Each processor operates on a different sub-matrix and solves the differential equations. The processors communicate with each other whenever one processor requires values that the other one has. This is repeated for a number of iterations, which is specified as an input to the program. First, process 0 initializes E and R matrices, divides them into p sub-matrices, and distributes them to all the processes running the program. Every processor receives sub-matrices and stores them as inner computation blocks. To solve the 5-point stencil-based equations for a point on the boundary of the inner block, a neighbor value might be required but these values reside in other processors. So, they are obtained using message passing between processors. To facilitate this, every processor has some space around the inner block, called ghost cells, which store the neighboring values of boundary points, and these values are obtained from other processors. This communication between two neighboring processors is called ghost cell exchange.

In a ghost cell exchange, each processor sends the rightmost column in its inner block to the processor operating on the right sub-matrix, the leftmost column to the processor operating on the left sub-matrix, the topmost row to the processor above, and the bottom-most row to the processor below. Similarly, it receives rows and columns from surrounding processors and stores them in its ghost cells. This exchange of rows and columns ensures that every processor has neighbor values for all its boundary points and is termed as a ghost cell exchange which is done using MPI.

But, there might be a case where a processor does not have any other processor to its right or left or top or bottom. Such processors hold a sub-matrix that lies on the boundary of the big original matrix created by process 0. In this condition, the ghost cells copy values from the inner block such that the points on the boundary of the inner block have the same values around them.

Sometimes, the size of the initial matrices may not be divisible by the processor geometry. In this case, all processors will not have submatrices of the same size. To get optimal performance, matrices have to be divided among processors in such a way that no processor has to work on more than 1 row or 1 column of data as compared to any other processor. The program ensures this by increasing the size of the submatrices of a few initial processors by 1 row or 1 column. The number of such processors is determined by the remainder when the original matrix size N is divided by each dimension of the processor geometry cb.px and cb.py.

Once a processor has all the values it needs, it solves the differential equations to compute new values for the next iteration in the simulation and the whole process repeats for a given number of iterations.

Development Process

The first step was to distribute the initial values to all the processes. We first used MPI_Broadcast to send the whole E_prev and R matrices but this caused a performance bottleneck. So, we then used MPI_Scatterv to scatter initial values to all the processors to improve performance. In this method, the program first packs the E_prev and R matrices so that the sub-matrices appear in order in a 1-D array. Then this packed array is

scattered to all processes using MPI_Scatterv (it supports variable length scattering also) and every process uses the data received from the scatter to prepare their local E_prev and R matrices (these local matrices include the boundary ring of ghost cells).

For processors that handle submatrices that lie on the boundary of the big original matrix, some of the ghost cells simply copy values from the inner block so that values around the boundary of the inner block are the same. The boundary conditions are copied to avoid computing single-sided differences at the boundaries. This is called padding of the submatrix.

The next part was the ghost cell exchange. Synchronous send and receive were used at first for the exchange to weed out any deadlock issues. Once the exchange worked properly, send and receive were made asynchronous by using MPI_Isend and MPI_Irecv to improve performance. Also, the program uses MPI_Vector_Type to move column data en masse.

Then come the differential equations. There was not much work in this part as the program had to just use the local matrices and correct indices to access values properly in the MPI version of the code. After doing this, L2norm and Linf had to be calculated. This was done using MPI_Reduce, which takes the sum of squares and max absolute values from all processes, reduces them (MAX for Linf and SUM for sum of squares) and process 0 gets the final reduced results which are then used to compute Linf and L2norm.

Once the program correctness was ensured, the next part was to handle the cases where matrix size N is not divisible by processor geometry. Assuming that the number of processors in x dimension is cb.px, N%cb.px is non-zero in this case. To handle this, the program increases the size of local matrices in x dimension by 1 column for the first N%cb.px processors in x direction. The same is done for the y dimension also. This way the program can use MPI even when matrix size N is not divisible by processor geometry.

To improve performance, there was an attempt to interleave computation and transmission of ghost cells so that the processors can do some computation while waiting for send and receive requests to complete. During this time, it can compute values that don't need ghost cells and once the exchange is complete, the remaining values that need ghost values can be computed. But no performance improvement was observed from this idea and so this was not a part of the final implementation.

Ideas for improvement

These are the ideas we tried, to improve the performance of the program :-

1. **Initial condition** :- To distribute the initial conditions to all the processes, we first used broadcast to send the whole E_prev and R matrices to all the processes. This resulted in a subpar performance because sending the whole matrices to each and every process uses a lot of bandwidth in processor communication. A better method is to divide the big matrix into smaller sub matrices and send only those sub matrices to processors which they need. The program does this by packing the big matrices and scattering them to all the processes using MPI_Scatterv. This method resulted in an improved performance.
2. **Asynchronous send and receive** :- First, synchronous send and receive were used to detect any deadlock issues. But if there is no deadlock problem, synchronous send and receive add a lot of unnecessary wait. Allowing Isend and Irecv for ghost cell exchange lets the processors put independent send and receive requests all at once and each process can just wait only for its own requests to complete before proceeding with the differential equation computation. This improves the performance by a lot, especially when the number of iterations is large.

3. **Interleaving computation and transmission** :- Another optimization we tried was to interleave the computation of differential equations with ghost cell exchange. The idea was that, while a processor waits for its asynchronous send and receive requests to complete, it can perform computations by solving equations for the values that do not need ghost cells, thereby utilizing the wait time. But, we did not observe any performance improvement. This is probably because of the higher cache misses that the program will face while computing values that need the ghost cells because in this computation, the process is working on a ring of memory where no two values are contiguous. So, the improvement from interleaving is counterbalanced by degradation from cache misses.
4. **Other minor optimizations** :- Use of pointers directly in MPI calls instead of dereferencing an array and referencing it again is a minor optimization in the program. Apart from this, scattering only E_prev and R matrices and not the E matrix during initial value distribution also improves the performance. Distributing only E_prev and R and not E is correct because E is something that the process computes in the current iteration and is not used, whereas E_prev and R are something that are used in the current iteration. So, E_prev and R are needed initially but not E.

In the final design, process 0 sets up initial values, packs them in a 1D array and distributes them among all the processes using MPI_Scatterv. All the processes receive data from scattering and construct their local E_prev and R matrices. Then the ghost cell exchange takes place using Isend and Irecv asynchronous MPI calls. Once the exchange is complete, each process performs differential equation computation. After completing all iterations, process 0 gathers the sum of squares and max absolute value data from all processes using MPI_Reduce and this data is reported in the final result to check correctness.

Section 2 - Results

Single processor performance of parallel MPI code vs original provided code.

	GFlops/sec	Running Time (sec)
Original code	11.2	191.415
Single-processor MPI	13.14	163.7

NOTE: The results are for N0 = 800 & run for i = 120,000 iterations

Performance of original code and single processor MPI are similar because of no communication in MPI.

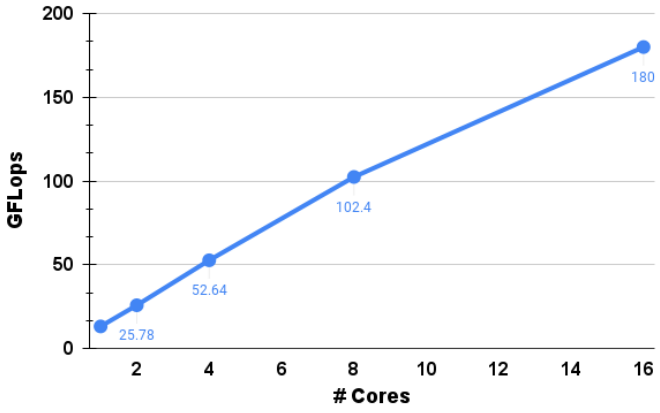
Cores	Geometry	GFlops/s	Time (s)	GFlops/s (No Comm)	Time (s) (No Comm)	MPI Overhead (s)	MPI Overhead (% of running time)
1	1x1	13.14	163.7	13.19	163.1	0.6	0.37
2	1x2	25.78	83.42	26.08	82.47	0.95	1.14
4	1x4	52.64	40.85	53.12	40.48	0.37	0.9
8	1x8	102.4	21	106.1	20.28	0.72	3.43
16	1x16	180	11.96	213.2	10.09	1.87	15.64

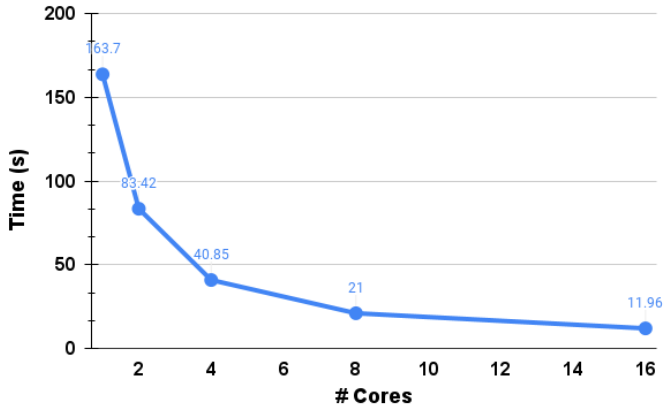
NOTE: The results are for N0 = 800 & run for i = 120,000 iterations

We observe that as we scale MPI from 1 to 16 cores, the MPI overhead also increases at a superlinear rate. This is expected as having more processors also entails a larger communication requirement amongst each other.

Strong scaling study on 1 to 16 cores on Expanse with size N0

We note a strong scaling while scaling from 1->2->4->8 cores, but get sub-linear when scaling from 8->16 cores. We hypothesize that this artifact happens because of the increase in the communication time for 16 cores (as noted from the MPI Overhead % running time column). Such scaling is also dictated by Amdahl's Law.

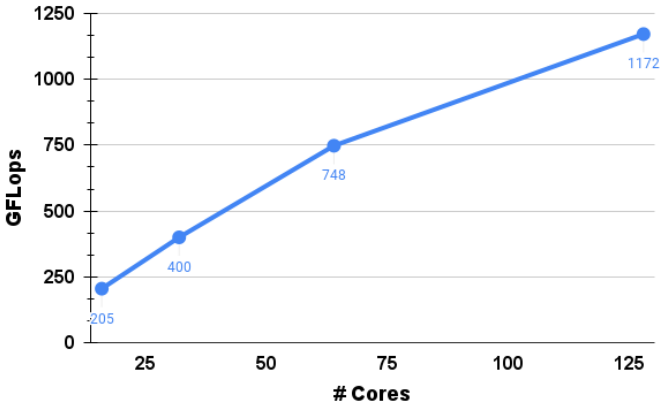


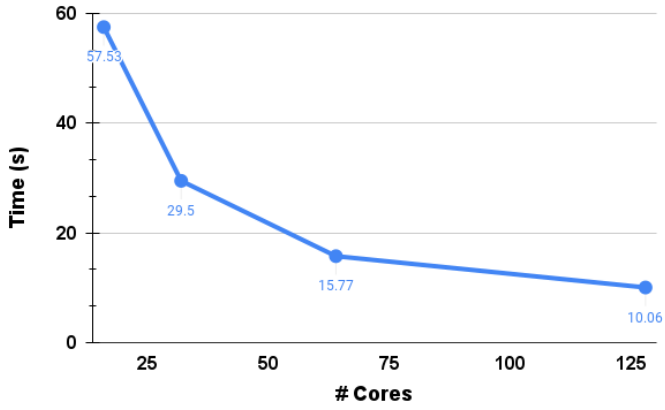


Cores	Geometry	GFlops/s	Scaling Factor	Time (s)	MPI Overhead (s)	MPI Overhead (% running time)
1	1x1	13.14	–	163.7	0.6	0.37
2	1x2	25.78	0.96	83.42	0.95	1.14
4	1x4	52.64	1.04	40.85	0.37	0.9
8	1x8	102.4	0.95	21	0.72	3.43
16	1x16	180	0.76	11.96	1.87	15.64

NOTE: The results are for N0 = 800 & run for i = 120,000 iterations

Strong scaling study on 16 to 128 cores on Expanse with size N1.





Cores	Geometry	GFlops/s	Scaling Factor	Time (s)	MPI Overhead (s)	MPI Overhead (% running time)
16	2x8	205	–	57.53	1.83	3.18
32	2x16	400	0.95	29.5	1.6	5.42
64	8x8	748	0.87	15.77	1.39	8.81
128	8x16	1172	0.56	10.06	1.374	13.66

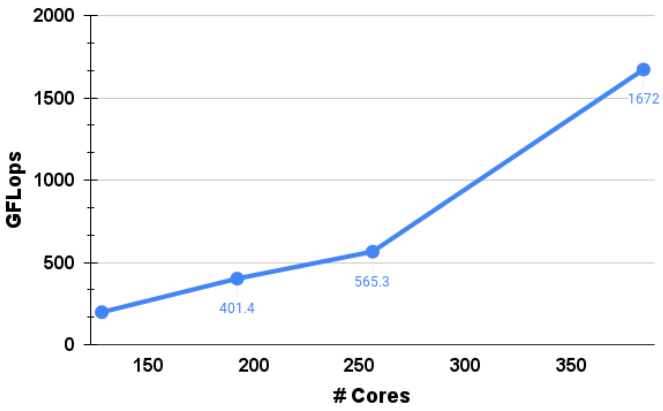
NOTE: The results are for N1 = 1800 & run for i = 130,000 iterations

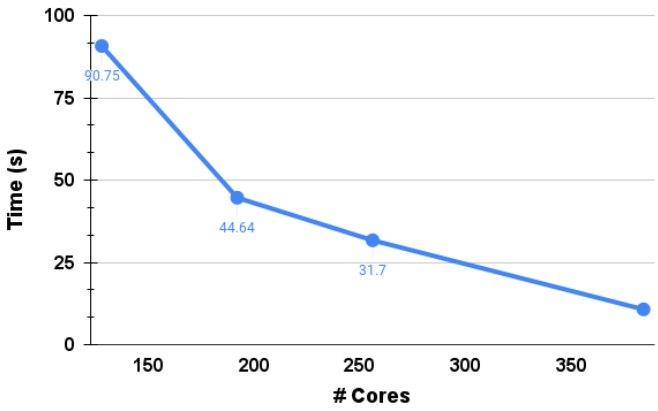
From the table and the graphs, strong scaling can be observed as the number of processors (p) to perform the same task increases. Also, absolute overhead time decreases with more p. This is because, as p increases, the sub-matrix of each processor becomes smaller. So, the amount of data for ghost cell exchange is less when p is more. Less data means less communication and so absolute overhead is less.

On the other hand, if overhead is represented as a percentage of running time, it can be observed that the overhead percentage increases with p. This can be explained by saying that sub-matrices shrink when p increases but with this shrinking, the amount of computation (area of sub-matrix) reduces faster than the amount of communication (perimeter of sub-matrix). So, even though absolute communication overhead goes down, its percentage with respect to overall running time goes up.

The increase in overhead percentage explains the sub-linear scaling in this case. The percentage of running time spent in communication increases as p increases and so, time spent in performing computation decreases. Due to this, more time is required to complete the same number of operations. This results in sub-linear scaling.

Performance study from 128 to 384 cores with size N2.





Cores	Geometry	GFlops/s	Scaling Factor	Time (s)	Time (s) (No Comm)	MPI Overhead (s)	Overhead as % of running time
128	8x16	197.5	–	90.75	87.99	2.76	3.04
256	4x64	565.3	1.86	31.7	13.45	18.25	57.57
192	8x24	401.4	–	44.64	44.59	0.05	0.11
384	4x96	1672	3.16	10.72	8.07	2.65	24.72

NOTE: The results are for N2 = 8000 & run for i = 10,000 iterations

Super-linear scaling is observed for both 128->256 and 192->384 scalings because in both cases we add an extra compute node on expanse. In addition to more cores, we also add an extra memory system. Hence, as we add more processors, both the memory and cache systems will be less bottlenecked, leading to a super-linear scaling. If in a scenario we were not adding more memory but just more processors, we hypothesize a sub-linear scaling.

Communication overhead differences between <=128 cores and >128 cores.

When p <= 128 i.e. for N1, overhead varies linearly. Absolute overhead increases with p and the overhead percentage decreases with p in a linear fashion. This can be explained by the area-perimeter analogy: the size

of the sub-matrix decreases when p increases and computation (which is similar to the area of the sub-matrix) goes down faster than communication (which is similar to the perimeter of the sub-matrix).

But when p>128, i.e. for N2, the change in overhead is erratic and not linear. From p=128 to p=192, it goes down to almost zero and for p=256 it jumps up suddenly to 18.25 sec. We hypothesize this is because when p>128, the number of nodes in the slurm scripts has to be more than 1. With processes distributed in different nodes, communication time depends on the time for a message to travel from one node to another. Even with the erratic behavior for p>128, the absolute overhead for p=384 is less than that for p=128 and the overhead percentage for p=384 is greater than that for p=128. This is similar to what was observed for p < 128 in N1.

On comparing directly, absolute overhead increases on going from N1 (p<=128) to N2 (p>=128) for a fixed p. This is because when matrix size increases, the boundary of the sub-matrix of every processor increases. This in turn increases the communication time because now, more data has to be exchanged with other processors.

Cost of computation

Cores	Computation cost (core-seconds) = #cores x computation time (s)
128	11,616 = 128 x 90.75
192	8570.88 = 192 x 44.64
256	8,115.2 = 256 x 31.7
384	4,116.48 = 384 x 10.72

NOTE: The results are for N2 = 8000 & run for i = 10,000 iterations

Using p=384 cores is the most optimal setting because computation cost is the least for p=384. Even though more cores are used as compared to other settings, both (a) the running time, and (b) the product of the number of cores and running time, are the least. Hence, this is the most optimal.

Section 3 - Determining Geometry

Top-performing geometries at N1 for p = 128.

Cores	Geometry	GFlops/s
128	1x128	875.9
128	2x64	1063
128	4x32	1065
128	8x16	1172
128	16x8	1107
128	32x4	1037
128	64x2	938.3
128	128x1	549.2

NOTE: The results are for N1 = 1800 & run for i = 130,000 iterations

Top performing geometries are 8x16, 16x8, 4x32, 2x64 and 32x4 as highlighted in the table above.

Patterns in geometry and performance.

Processor geometry is of the form **X** x **Y** where **X** is the number of processors in the x direction and **Y** is the number of processors in the y direction. Geometries with **X** < **Y** have an advantage as they maximize cache utilization, leading to lower memory latency. More specifically, **X** < **Y** means that each processor's sub-matrix

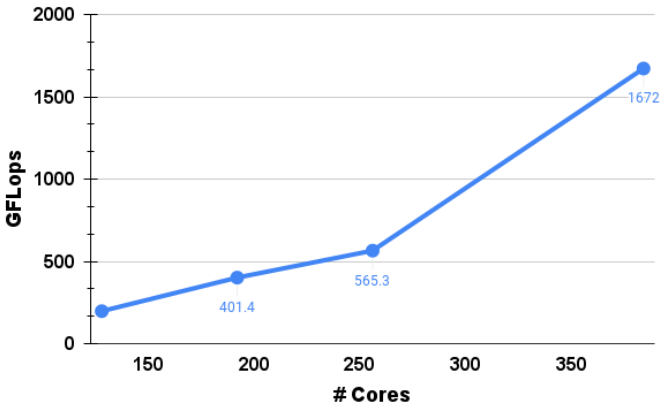
will have more elements in a row than in a column. Since elements along a row are contiguous in memory, an improved cache locality will lead to more time spent on computation and hence better performance.

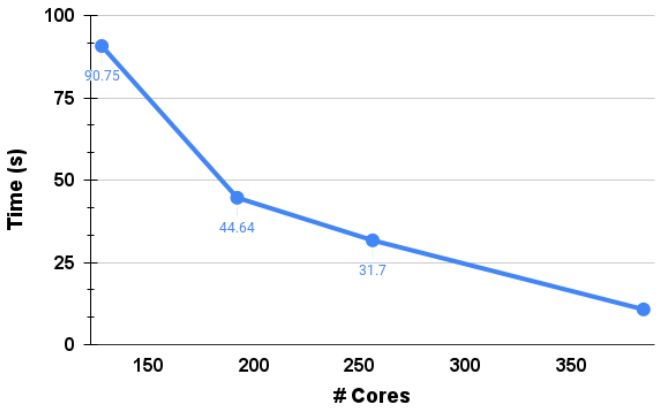
But, if the number of elements along the row is not a multiple of cache line size, then it may reduce the performance. Moreover, if there are too many elements in a row then more data has to be sent to processors above and below during the ghost cell exchange as compared to processors to the left and right. Processors in vertical order do not have consecutive ranks and might be present in different nodes making vertical communication more expensive than horizontal communication. Hence, having too many elements in a row can have greater communication costs, degrading performance.

Therefore, geometries that maximize the time spent on computation (by reducing memory latency for example) and minimize communication costs between processors give optimal performance. In general, we note geometries with **X** < **Y** perform better than **X** > **Y**. For instance, 8x16 geometry performs better than 16x8 geometry, 4x32 performs better than 32x4, and 2x64 performs better than 64x2. The aforementioned geometries were chosen as the highest-performing geometries because their performance lies within 10% of the best-performing geometry. 3 of them have **X** < **Y** but 2 of them have **X** > **Y**.

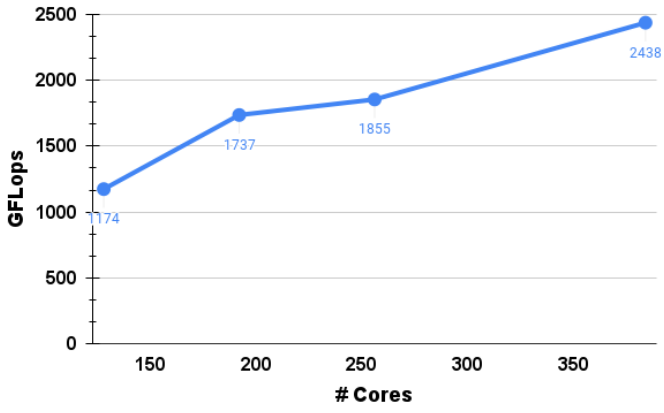
Section 4 - Strong and Weak Scaling

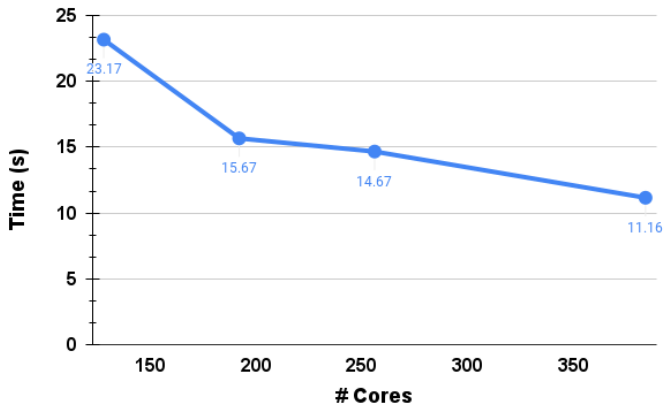
Performance with best N1 geometry at 128, 192, 256, and 384 cores.





STRONG SCALING
N2 = 8000 & run for i = 10,000 iterations





WEAK SCALING
N1 = 1800 & run for i = 300,000 iterations

Cores	N2 Geometry (strong scaling)	N2 GFLops/s	N2 overhead percentage	N1 Geometry (weak scaling)	N1 GFLops/s	N1 overhead percentage
128	8x16	197.5	3.04	8x16	1174	0.13
192	8x24	401.4	0.11	8x24	1737	0.24
256	4x64	565.3	57.57	4x64	1855	0.32
384	4x96	1672	24.72	4x96	2438	0.40

Differences in the behavior of strong scaling experiments for N1 and N2

For both N1 and N2, strong scaling can be observed as performance increases on increasing the number of cores. But for a fixed p and fixed geometry, performance goes down as matrix size goes from N1 to N2. Since the number of floating point operations increases on going from N1 to N2, less performance can be attributed to increased total run time. Total run time includes communication time and grind time. Communication time increases with N but not as fast as the number of operations (area vs perimeter analogy). So, it must be the grind time that has increased. As matrix size increases from N1 to N2, the program encounters more cache misses because the top and bottom points in the 5-point stencil are now farther apart as compared to the previous case. Due to this, the time to compute 1 point in the matrix goes up. This explains why the GFlops/sec is less in N2 than in N1 for a fixed p and a fixed geometry.

Section 5 - Potential Future Work

1. Implement the same program for non-square matrices.
2. Improve performance by vectorizing the code using AVX2 or hand vectorization.
3. Find out why interleaving computation with transmission does not improve performance.
4. Implement a method to plot the results of the simulation on a multi-core MPI application.
5. Implement the same program for a 7-point 3D stencil.

Section 6 - References (cite all references used)

1. Microsoft MPI documentation:- MPI Functions - Message Passing Interface | Microsoft Learn

2. Pacheco, “An Introduction to Parallel Programming” http://www.cs.usfca.edu/~peter/ipp/ipp source.tgz, Morgan Kaufman Publishers

3. “Some ways to think about parallel programming”, http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture25.pdf

4. M. Banikazemi, R. K. Govihdaraju, R. Blackmore and D. K. Panda, "MPI-LAPI: an efficient implementation of MPI for IBM RS/6000 SP systems," in IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 10, pp. 1081-1093, Oct. 2001, doi: 10.1109/71.963419.

5. A. Y. Grama, A. Gupta and V. Kumar, "Isoefficiency: measuring the scalability of parallel algorithms and architectures," in IEEE Parallel & Distributed Technology: Systems & Applications, vol. 1, no. 3, pp. 12-21, Aug. 1993, doi: 10.1109/88.242438.

6. Amdahl’s Law:- https://en.wikipedia.org/wiki/Amdahl%27s_law