Q1. Development Flow

Q1.a. Working of Program

The `init()` function pads the `m x n` mesh with zeros, which is equivalent to the 4 sides of ghost cells for a single processor. However, when the submeshes are distributed over multiple processors, there are 9 possible scenarios in which a particular processor can be present. Since the outermost row and column of the `m x n` mesh is already padded with ghost cells, processors containing these outermost rows and/or columns don't require additional ghost cell padding in those directions. The 9 possible scenarios are described by the diagram below.

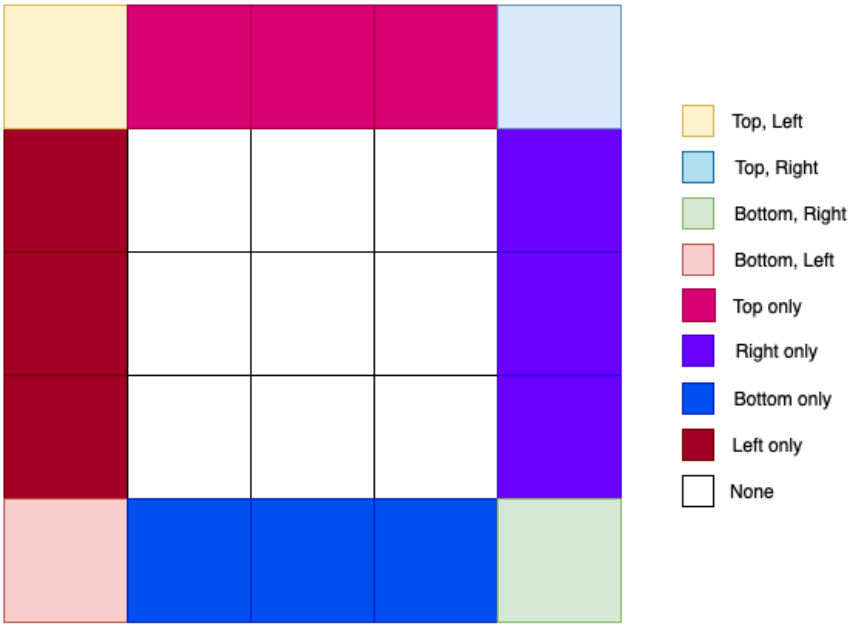


Figure 1: Possible processor scenarios

This means that the number of extra rows and columns required for ghost cell exchange will vary in each of these 9 different scenarios. For instance, the Cell already has ghost cells on its left side, and requires further ghost cell padding on its remaining three sides. Similarly, Cell already has the right and bottom ghost cells. So, it only requires further ghost cell padding on its top and left sides. Since MPI allocates processor rank increasing along the X-axis, the location of each processor is identified by `(rank / py, rank % py)`. Accordingly, we identify which of the above 9 categories the processor falls into, and add the additional number of rows and columns required for each processor. To ensure even distribution of rows and columns in submeshes, we first ensure that each submesh gets `m + 2 / px` rows and `n + 2 / py` columns. Then, `m + 2 % px` rows and `n + 2 % py` columns are again evenly distributed among submeshes. Processor 0 acts as a master node and sends the respective augmented submeshes (with ghost cells included for each submesh) to each of the other processors using `MPI_Isend()` and each of the other processors receives these submeshes using `MPI_Irecv()`. These two MPI calls are asynchronous and non-blocking. Since each of these operations is asynchronous, we invoke `MPI_Wait()` after completing all sends and completing all receives to ensure that the receives occur only after all sends have been completed, and all computations begin only after all receives have completed. This is somewhat similar to `_syncthreads()` used in PA2.

The code to identify location of the processor along with how many rows and columns are present in the submesh is replicated in the `solve.cpp` file. Accordingly, if the current processor contains the outermost rows and/or columns of the `m x n` mesh, the ghost cells are filled. In any other case the rows and columns are sent to neighboring processors whose locations are calculated since we know the position of the current processor in execution. These rows and columns are sent using a new data type created for each in MPI using `MPI_Type_contiguous` [2] for a row and `MPI_Type_vector` [3] for column since the latter is not contiguous. Following all sends, we begin receiving these ghost cells in each of the processor's outer rows and columns. As previously mentioned: since each of these operations is asynchronous, we invoke `MPI_Wait()` after completing all sends and completing all receives to ensure that the receives occur only after all sends have been completed, and all computations begin only after all receives have completed.

The PDE and ODE are solved for each processor's computational mesh (excluding ghost cells). We observed a higher performance when the two loops for computing PDE and ODE are fused. After solving the two differential equations, the sum of squares (`sumSq`) and Linf are computed using the `stats()` function, and these two values are aggregated to the "master" processor (rank 0) using `MPI_Reduce()`.

Note: In our implementation, x determines the number of processors in the vertical direction and y determines the number of processors in the horizontal direction.

Q1.b. Development Process

We started the groundwork for parallelization by distributing `E_prev` and `R` (and `E`, in case of plotting) submeshes to different processors. This is required since each processor performs computations on a different submesh of `E_prev` and `R`. After verifying synchronous blocking calls like `MPI_Send()` and `MPI_Recv()` were working without error, we proceeded to implement asynchronous MPI.
Next, we implemented the logic to send rows and columns to neighboring processors and receive rows and columns to fill up ghost cells. Next, we altered the indexing of PDE and ODE loops to access elements at the new locations since each processor now had submeshes with different values and locations in memory. Lastly, we used `MPI_Reduce()` to collect required stats from all the processors and compute L2 and Linf.

Q1.c. Performance Optimization Steps

1. Divide the computation task among multiple processors using MPI and ghost cell exchanges described in Q1a. This optimization was enough to yield satisfactory performance results.
2. We used a fused version of PDE and ODE solver because it yields more cache locality (reduces cache misses) and thereby helps improve performance.
3. We tried to overlap the communication phase with computation. So, we first send ghost cells asynchronously and start the computation for all cells except boundary cells in the inner block. Then, we receive ghost cells and we do computation for the inner block boundary cells. This didn't give us a performance boost and we had to remove this optimization. It seemed like having multiple conditions and for loops for such split-phase computation incurred more overhead.

Q2. Result

Q2.a. Strong Scaling Study: MPI Overhead (N0)

All the experiments in this subsection have been performed using the provided N0 configuration, i.e., `m = n = 800` and `i = 40000`, while using a shared queue on expanse.

Original Code Performance (GFlops) for a single core	**MPI Code Performance (GFlops) for a single core**
11.6	12.7

Number of Processors (MPI)	**Processor Geometry (MPI)**		**MPI Parallelized Performance (GFlops)**	**MPI Overhead (seconds)**	**MPI Overhead (%)**
	x	**y**			
1	1	1	12.7	0.0563	0.12
2	2	1	26	0.0829	0.27
4	4	1	50.9	0.0703	0.51
8	8	1	103	0.0697	0.93
16	16	1	176	0.0691	1.78

Our MPI-based code works slightly better for 1 core than the original provided code because we have a more optimal read-write pattern (i.e. takes advantage of cache locality) to fill the initial boundary cells.

We notice that the percentage of MPI overhead when compared to the overall running time increases almost linearly with the number of cores. This is because with the increase in number of cores, the number of ghost cell exchanges and wait time for all exchanges to complete increases.

Q2.b. Strong Scaling Study: Running Time (N0)

All the experiments in this subsection have been performed using the provided N0 configuration, i.e., `m = n = 800` and `i = 40000`, while using a shared queue on expanse.

Number of Processors (MPI)	**Processor Geometry (MPI)**		**Running Time (seconds)**
	x	**y**	
1	1	1	56.3496
2	2	1	27.6188
4	2	2	14.079
8	8	1	6.9749
16	16	1	4.0616

We notice that the overall running time decreases almost linearly with respect to the number of cores. This is because with the increase in number of cores, we have more throughput (i.e. more instructions executed each cycle in the overall system). Moreover, cache locality at individual processors gives further advantage of having lesser cache misses (i.e. less miss penalty). All these performance benefits altogether outweigh the communication overhead and thereby result in nearly linear decrease in running time.

Q2.c. Strong Scaling Study: Communication Overhead (N1)

All the experiments in this subsection have been performed using the provided N1 size configuration, i.e., `m = n = 1800` and `i = 150000` while using a compute queue on expanse.

Number of Cores	Geometry		GFlops	Communication Overhead (seconds)	Communication Overhead (%)	Running Time (seconds)
	x	y				
16	8	2	203	1.4069	2.09	67.028
32	16	2	384.6	2.6075	4.26	35.3809
64	8	8	729.3	1.6008	8.58	18.6582
128	16	8	1131	1.9601	16.29	12.0329

We notice that the communication overhead with respect to the total running time (and thereby communication cost) increases almost linearly with the number of processors. This is because as the number of processors increase, there is an increase in the number of ghost cell exchanges and the wait time to complete all the ghost cell exchanges.

Although the communication cost increases, there is a linear decrease in the computation cost with an increase in the number of processors. The near linear decrease in running time shown above demonstrates that the compute cost advantage we get with more cores outweighs the communication overhead incurred and results in nearly linear decrease in overall running time.

Q2.d. Performance Study (N2)

All the experiments in this subsection have been performed using the provided N2 size configuration, i.e., `m = n = 8000` and `i = 10000`, while using a compute queue on expanse.

Number of Cores	Geometry		GFlops	Communication Overhead (seconds)	Communication Overhead (%)
	x	y			
128	16	8	196.5	2.6077	2.86
192	32	6	391.3	0.6975	1.52
256	64	4	543.9	18.5119	55.82
384	96	4	1515	3.8172	42.81

There is a nearly linear increase in performance with an increase in the number of cores. Also there is significant communication overhead (42% - 44%) for >= 256 cores.

Q2.e. Communication Overhead Differences

For <= 128 cores, we noticed < 16% overhead and this increase in overhead is nearly linear with an increase in the number of cores.

For 192 cores, we didn't see a significant increase in communication overhead. We suspect that this is because the geometry perfectly fits the node such that there is minimal inter-socket communication and high LLC-cache locality.

However, for large cores sizes like 256 and 384 cores, we notice a significant communication overhead (around 42-55%) and huge variance in communication overhead (not a clear linear increasing trend). This may be because of the following factors:

1. Time needed for ghost cell exchange varies significantly based on whether the receiving core is on the same socket or different one. Communication between processors on the same socket has low latency as well as LLC cache locality advantage.

2. Depending on the processor geometry and mesh size, there will be significant difference in cache misses for packing ghost cells.Thus, the overall miss penalty varies widely based on the processor geometry.

3. There is a significantly higher number of ghost cell exchanges and the wait time for all ghost cell exchanges to complete will also be higher.

Q2.f. Cost of Computation (N2)

Number of Cores	Computation Cost = #cores x computation time
128	128 x 88.6058 = 11341.5424
192	192 x 45.1025 = 8659.6800
256	256 x 14.4341 = 3695.1296
384	384 x 8.0101 = 3075.8784

The most optimal geometry based on computation time and computation cost is achieved at 384 cores (`x = 96, y = 4`). This geometry results in significant reduction in computation time per core to the extent that its overall computation cost is also the lowest. This is because this particular geometry has the highest throughput (i.e. instructions executed per cycle in the overall system) and cache locality achieved at each processor reduces the overall miss penalty.

Q3. Determining Geometry (N1)

All the experiments in this section have been performed using the provided N1 size configuration, i.e., `m = n = 1800` and `i = 150000` while using a compute queue on expanse.

Q3.a. Top-performing Geometries for `p = 128`

Geometry		GFlops
x	y	
16	8	1118
8	16	1081

Q3.b. Patterns and Hypotheses

In our code, x refers to the number of processors in vertical direction while y refers to the number of processors in horizontal direction.

Increasing processors along x direction leads to more communication along the top-bottom direction and gets the advantage of cache locality while sending and receiving ghost cell rows. This leads to a decrease in communication cost. But, in this case, if each processor has large size rows, there will be cache misses when the PDE - ODE solver tries to access the top and bottom row cells. This leads to an increase in computation cost.

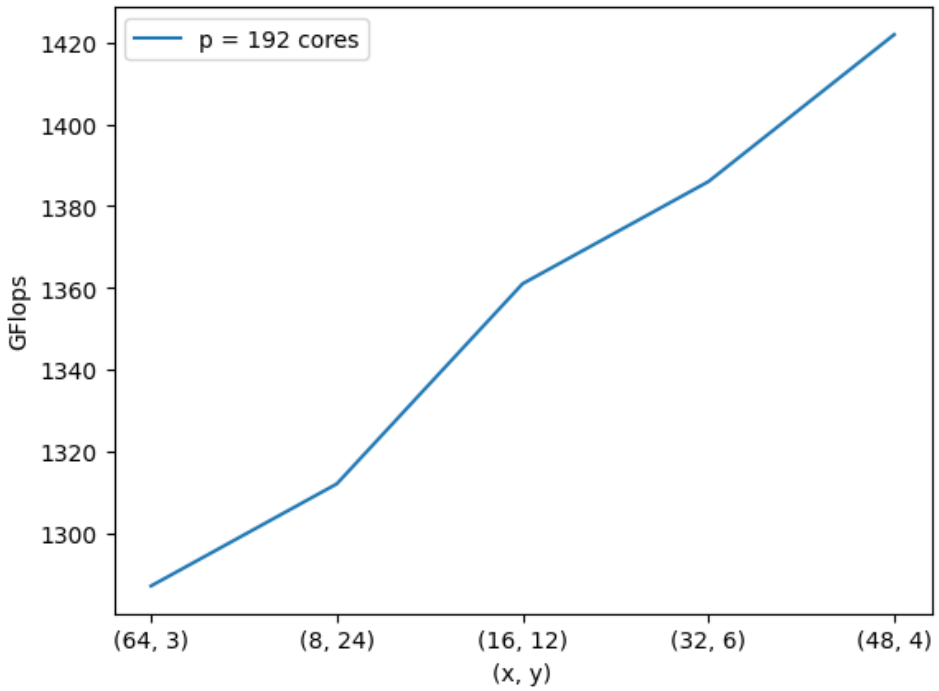
Increasing processors along y direction leads to more left-right direction communication and leads to more cache misses while sending-receiving ghost cell columns. This leads to an increase in communication cost. But, in this case, if each processor has small size rows, there will be cache locality when the PDE - ODE solver tries to access stencil cells. This leads to a decrease in computation cost.

Considering this tradeoff, we observe that we need to find a sweet spot in between where we can take the aforementioned advantages of having cores in x and y direction and have least cache misses. Our table below validates this hypothesis. Increasing cores along x direction reduces cache miss rate up to a point and then adding more cores along x direction increases the cache miss rate. Thus, the sweet spot in between leads to optimal cache miss rate and performance.

Geometries for p = 128 (N1)		GFlops	L1-d Cache Miss Rate (%)
x	y		
8	16	1081	5.9
16	8	1118	4.9
32	4	1097	5.0

Using the same insight shared above, we found top-performing geometries for large core counts.

Geometries for p = 192		GFlops
x	y	
48	4	1422
32	6	1386
16	12	1361
8	24	1312
64	3	1287



Geometries for p = 256		GFlops
x	y	
32	8	1665
16	16	1583

Geometries for p = 384		GFlops
x	y	
48	8	2213
64	6	2073

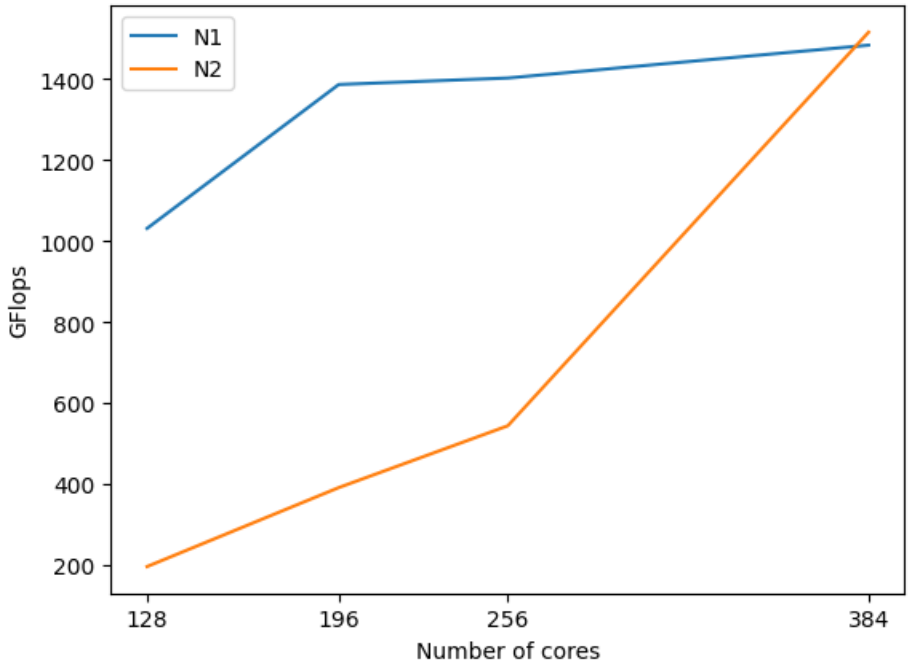
Q4. Strong and Weak Scaling Study

Q4.a. Best N1 Geometry for large core counts

All the experiments in this section have been performed using the provided N1 size configuration, i.e., `m = n = 1800` and `i = 150000` while using a compute queue on expanse.

Number of Cores	Geometry		GFlops
	x	y	
128	16	8	1031
192	32	6	1386
256	64	4	1402
384	96	4	1483

N1 configuration leads to significantly higher performance than N2 for less than equal to 256 cores geometries. This is because each core now has a significantly smaller submesh and there is a lot of cache locality available.



Q4.b. Differences in Strong Scaling Behavior: N1 vs. N2

For N1, we see a linear increase in performance with an increase in the number of cores for less than equal to 196 cores. For greater than 196 cores, there is minimal increase in performance. This is because at 196 cores, we have sufficiently small submesh for each processor and thus sufficient cache locality. Increasing cores beyond this point will not give much cache locality advantage and actually give more communication overhead. Thus, the performance gain we achieve by increasing cores is not too high and that's why we are unable to achieve a linear increase in performance for cores greater than 196.

For N2, we see a close to linear increase in performance with an increase in the number of cores. This is because the submesh that each core gets is big enough and as the number of cores increases, there is an increase in the cache locality. Moreover, there is increase in throughput (i.e. number of instructions executed overall per cycle in the system) with the increase in number of cores. These benefits outweigh the communication overhead and thereby give us nearly linear increase in performance with an increase in the number of cores.

Q5. Extra Credit

Q5.EC-a. Plotting on a Multi-Core Application

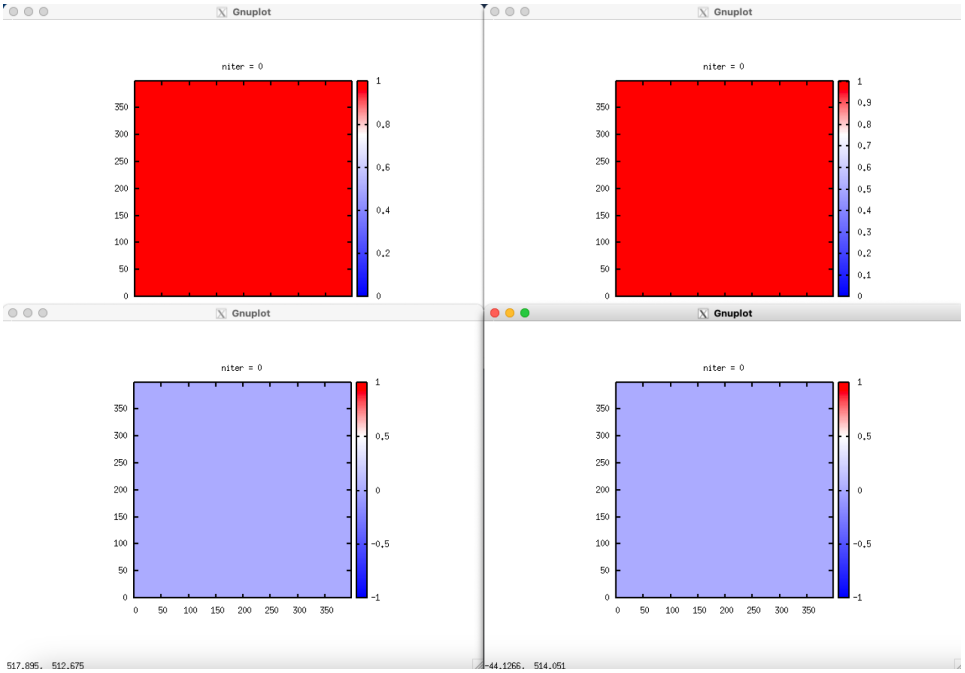


Figure 4a: Snapshot updates of evolving simulation: N0 Iteration #0

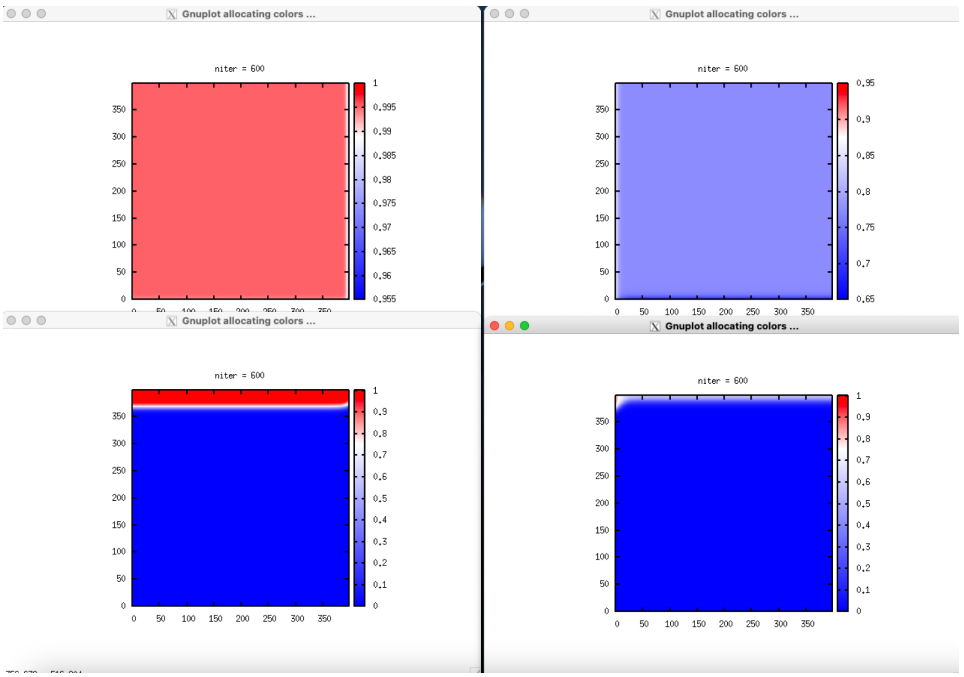


Figure 4b: Snapshot updates of evolving simulation: N0 Iteration #600

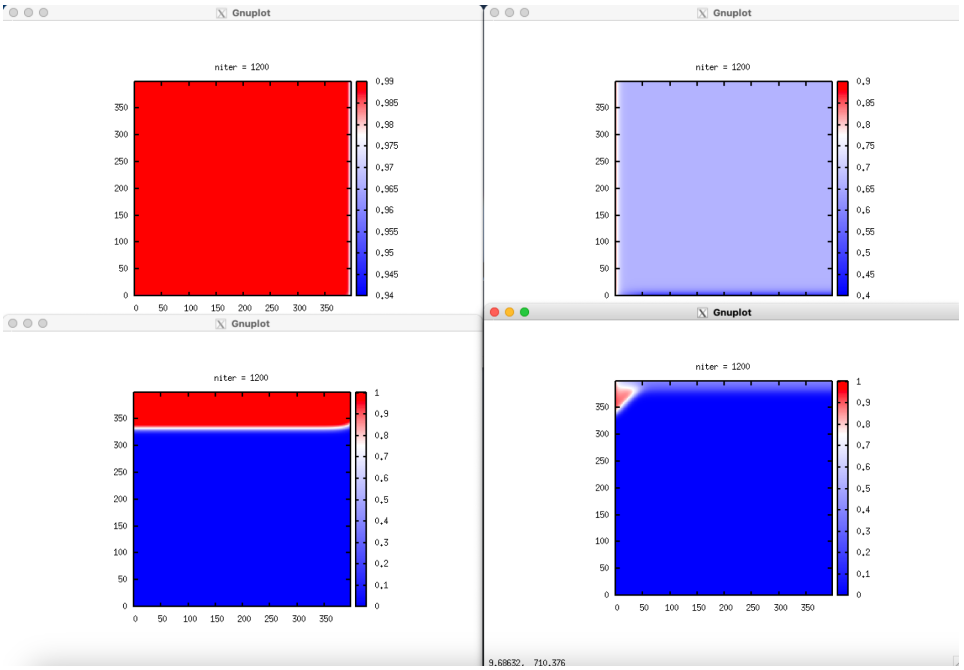


Figure 4c: Snapshot updates of evolving simulation: N0 Iteration #1200

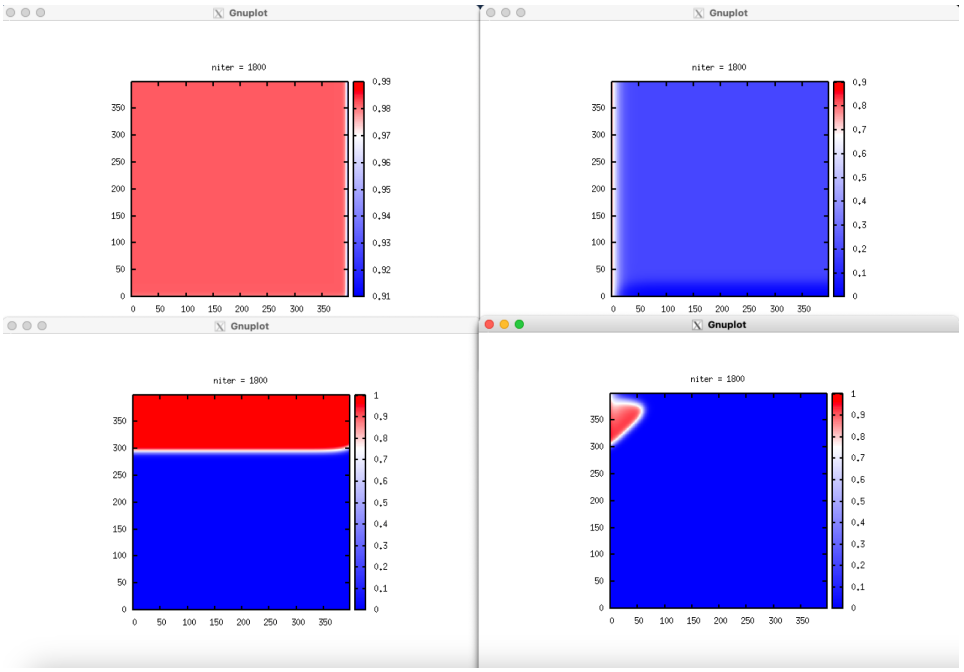


Figure 4d: Snapshot updates of evolving simulation: N0 Iteration #1800

The code for MPI plotting is present on the plotting branch. As per our implementation of plotting on a multi-core application, each core that plots also performs computation. This causes an overhead and hinders performance. This is definitely a problem worth solving - one way we thought of solving this problem was:

1. Choose a processor at random (say one of the white processors from Figure 1)
2. The chosen processor's neighboring processors will be filled with rows and columns to compensate for the current processor's dimensions.
3. This will produce other issues such as non-square matrix multiplication and will have to be handled accordingly to ensure correct results.
4. The processor chosen in step 1 can now work only on plotting and not computation.

EC-b) Vectorization

The code for MPI hand vectorization is present on the avx branch.
To vectorize the code, we store the scalar values (constants like alpha, dt, kk, and so on) in 128-bit registers and perform the computations of solving ODE and PDE using vectorized AVX instructions [4]. Since the vector contains two elements of a submesh (256 bits = 4 doubles), we need to iterate through the computations of PDE and ODE with a step size of 4 (which was 1 before vectorization, since a single value was computed in each inner iteration). These optimizations can be toggled on/off by the initially provided SSE_VEC flag and requires the `-mavx` flag while compiling (this change has been added to the Makefile on the `avx` branch). To compile the vectorized MPI code, running `make vec=1 mpi=1` is used. Also, we disable the compiler's auto-vectorizer by uncommenting `__attribute__((optimize("no-tree-vectorize")))`. The results below are for experiments conducted with `m = n = 800` and `i = 100000`.

Cores	Geometry		Scalar Performance (GFlops)	Hand Vectorized Performance (GFlops)
	x	y		
64	16	4	540	715
128	32	4	641	740
256	64	4	730	760
384	48	8	724	1062

We also tried to add AVX2 flags (`-mavx2 -mprefer-vector-width=256`) along with `-mfma` and change the slurm files to use aocc (`module load aocc`) to get the code to vectorize without SSE hand vectorization. However, for the above geometries, we were only able to achieve ~1.6 GFlops. Due to a time constraint and prioritizing the required non-extra credit readings, we were unable to optimize the code further with sufficient evidence for an improved performance.

Q6. Potential Future Work

1. We would like to overlap the computation and communication phase and see how much performance enhancement we can achieve.
2. We would like to try implementing 9-D stencil that was discussed in class and figure out how we can implement ghost cell communication for it.
3. We would like to do an implementation that works with non-square computation meshes.

Q7. References

1. MPI_Type_contiguous Data Type
   https://www.mpich.org/static/docs/v3.3/www3/MPI_Type_contiguous.html
2. MPI_Type_vector Data Type
   https://www.mpich.org/static/docs/v3.3/www3/MPI_Type_vector.html
3. Intel Intrinsics Guide
   https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html