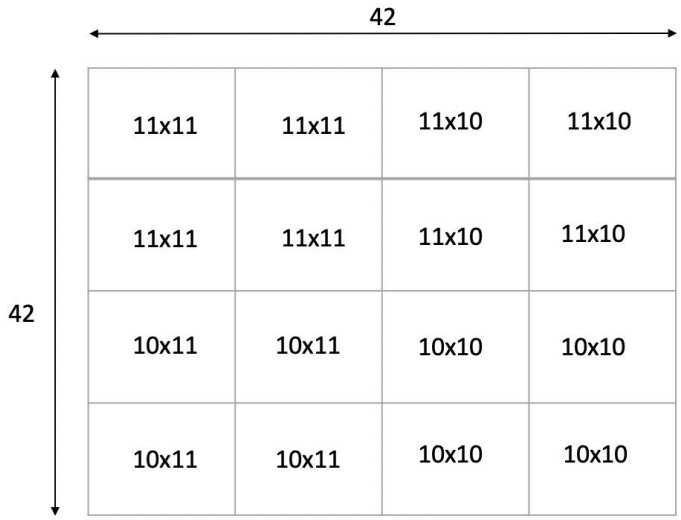
CSE 260 PA #3

Section (1) - Development Flow

Q1.a) Describe how your program works (pseudo code is fine, do not include all of your code in the write-up). [ghost cell exchanges, boundary cases, even distribution, MPI calls etc.]

For input mesh of dimension $M \times N$, we divide it into blocks of dimension $\frac{M}{p_x} \times \frac{N}{p_y}$ for each of $p_x \times p_y$ processor cores. When $M \times N$ is not integer multiples of $p_x \times p_y$, processors located at the right or bottom sides of the mesh will deal with smaller blocks accordingly. We use an example in the chart below to show how we evenly distribute the work so that no processor has more than 1 row or column more work than any other processor.



Before the actual computation iteration begins, the process 0 will distribute initial conditions to the other processes. This is conceptually a scattering operation but ordinary MPI_Scatter() function cannot deal with the sophisticated layout of subblocks in the overall mesh. Thus it flattens the mesh into vectors of equal length and scatters these vectors to all processes for them to retrieve subblocks. Each process may have its own buffer to store its subblock to improve cache locality.

Afterwards, the program simulates Aliev-Panfilov equations iteratively. Each iteration consists of three steps. First, each subblock will need boundary information from neighbors or ghost cells that copies the interior of the subblock itself. We copy memories within the process to set up ghost cells and use MPI asynchronous communication to exchange boundaries of neighboring processes. Here we apply MPI_Isend() and MPI_Irecv() for sending and receiving data and MPI_Waitall() is used for synchronization.

Second, it computes actual equations from the previous result to the current one. We decide to use the unfused versions where PDE and ODE are computed separately because this way each loop will access a smaller number of different memory locations, enabling the compiler to possibly load them all in to registers and vectorize the loops.

Lastly, the process 0 gathers statistics like L2, Linf norms from all processes. This is an ordinary reduction operation so we use MPI_Reduce() with MPI_SUM and MPI_MAX. And if command line option -p is set, it gathers all subblocks to plot the graph. Apparently, this gathering is the reverse of the initial distribution, so we reuse that logic.

Q1.b) What was your development process? What ideas did you try during development?

1. At first, we downloaded the starter code and managed to run it on bang, comet and our own machines. We also reviewed the code, especially about dimensions, indices of ghost cells and computations.

2. We followed the idea in the course slides. Each process computed a subblock of the mesh and exchanged boundary data with its neighbors. We used asynchronous communication for each process to communicate its top, bottom, left, right ghost cells with respective neighbors. Importantly, we figured out the dimensions, indices, strides of the input.

3. The code provided was not enough to check correctness of our parallel program, we needed a way to gather back L2, Linf statistics. We reviewed stats() function for the whole mesh and implemented one for subblocks. After each process computed its statistics, we used MPI_reduce() to automatically compute data for the whole mesh. With that, we debugged our current code, and tested on bang.

4. The assignment required to distribute initial data from the process 0. Actually the starter code computed data in every process, so we should pretend to ignore that. The naive way was to broadcast the whole mesh to every process though each would only use part of it. However, we asserted that would not impede performance as it did not affect the computing iteration.

5. However, to improve cache locality, it was better in each process that the subblock was stored in a more contiguous manner. For now, subblocks had the same stride equal to the dimension of the overall mesh. Therefore we needed to improve the way we distributed data. One way was to simply send/receive sub blocks row by row, another was to reorganize the mesh for scattering operation.

6. Besides testing our program on bang and comet, we also finished Extra Credit a. We reversed the scattering for initial data and managed to gather subblocks back to plot in process 0.

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.

First improvement is to use MPI to broadcast initial data and do parallel computation. Second improvement is to use MPI to scatter initial data and store subblocks in smaller buffers. Compared to started code and MPI reference on the server, our code has a better performance. (N=400, iters=2000)

GFlop/sec	np=1	np=2	np=4	np=8	Starter code (w/o MPI)
MPI reference	0.4557	0.9067	1.811	3.593	0.4559(FUSED) 1.13(UNFUSED)

Broadcast	1.107	2.29	4.502	8.497	
Scatter+locality	1.15	2.298	4.545	8.895	

We know that the unfused version is better than the fused version so our code always uses the unfused version. It is interesting that MPI reference is worse than unfused starter code, so We guess the reference program uses fused computation.

Section (2) - Result

Q2.a) Compare the single processor performance of your parallel MPI code against the performance of the original provided code. Measure and report MPI overhead on bang(1 core and 8 cores).

When N=400, iters=2000, we divide our GFlop/s by the number of cores and compare it with starter code.

	np=1	np=2	np=4	np=8	Starter code (w/o MPI)
GFlop/sec/core	1.15	1.149	1.13	1.111	1.13
MPI overhead	-0.02	-0.019	0	0.019	0

It seems our code has better performance than the starter code so that some MPI overhead is relieved.

Q2.b) Conduct a strong scaling study: observe the running time as you successively double the number of cores, starting at p=1 core and ending at 8 cores, while keeping N0 fixed.

When N=400, iters=2000, for each number of cores we compute the scaling factors compared to that of half number and that of one core.

	np=1	np=2	np=4	np=8
GFlop/sec	1.15	2.298	4.545	8.895
Scaling against np/2	N/A	0.99	0.98	0.97
Scaling against one core	1	0.99	0.98	0.96

Our code has a scaling factor >0.95 consistently over np=1-8, so strong scaling is indeed achieved.

Q2.c) Measure communication overhead on 8 cores.

Command option -k will set cb.noComm, so we ensure when this is set no MPI communication is initiated. When N=400, iters=2000, we compare the communication overhead.

Running time/sec	np=1	np=2	np=4	np=8
With communication	7.765	3.900	1.971	1.021
Without communication	6.260	3.116	1.556	0.781
Communication overhead	0.19	0.20	0.21	0.23

It is shown empirically that communication overhead increases as the number of blocks increases. It is because more blocks results in more boundary data to exchange among processes.

Q2.d) Conduct a strong scaling study on 24 to 96 cores on Comet. Measure and report MPI communication overhead on Comet.

When N=1800, iters=2000, for each number of cores (24,48, 96) we compute the scaling factors. The result is as follow:

Scaling: 24->48 : 1.42 ; 48->96 : 0.99

As shown in the table below, the communication overhead of 24 cores , 48 cores, 96 cores is 0.15s, 0.17s and 0.07s. It's hard to find an obvious relationship between the number of cores and communication overhead because many factors may influence it. For example, when we use more cores for a fixed amount of computation, less data transmits between two different cores , saving some time. However, each node of the supercomputer owns 24 cores . When using 48 cores and 96 cores, we need to communicate across different nodes, thus increasing the communication cost.

Running time/sec	np=24	np=48	np=96
With communication	2.155	0.757	0.382
Without communication	2.005	0.587	0.312
Communication overhead	0.15	0.17	0.07

Q2.e) Report the performance up to 480 cores. (i.e. 96, 192, 240, 384 and 480 cores).(Use the knowledge from the geometry experiments in Section (3) to perform the large core count performance study.)

When N=8000, iters=2000, for each number of cores (96, 192, 240, 384, 480) we compute the scaling factors. The result is as follow:

Scaling: 192->384 : 1.08 ; 384>480 : 1.17 ; np= 480: 1085 Gflops

Running time/sec	np=96	np=192	np=240	np=384	np=480

Best geometry	px=8, py=12	px=16,py=12	px=20,py=12	px=24,py=16	px=30,py=16
With communication	19.344	10.442	8.705	4.842	3.304
Computation	18.664	9.672	7.808	4.190	2.678
Communication overhead	0.68	0.77	0.90	0.65	0.62
Execution time x cores	1857	2005	2100	1859	1586

Besides, the general geometry strategy is to make $\frac{N}{p_y}$ in the range of 400~800,which we will discussed detailedly in Q3.

Q2.f) Report cost of computation for each of 96, 192, 240, 384 and 480 cores. In addition to it, run the 'batch-test.sh' script file to report the Queue Time (i.e. Wait Time for the batch job to allocate the requested number of cores) in the following table.

The cost of computation for each of 96, 192, 240, 384 and 480 cores are reported in the table of Q2.e. Besides, we report the queue time for tasks of different cores in the following table. We do 3 times for each and take the average. We understand that usually when we require more cores per task, we need to wait longer. However, this trend doesn't appear in our experiment because it highly depends on the traffic of super computers and it's very hard to get that conclusion within limited times of experiments.

Cores	Queue Time
96	48.98
192	56.67
240	48.89
384	54.44
480	79.55

Section (3) - Determining Geometry

Q3.a) For p=96, report the top-performing geometries. Report all top-performing geometries (within 10% of the top).

We report five toppest geometries(within 10%) when N=1800, iter =2000 and p=96. The best geometry is px=24, py=4.

Rank	1	2	3	4	5
Geometry	px=24, py=4	px=48, py=2	px=32, py=3	px=96, py=1	px=8, py=12
Running time/sec (Com)	0.382	0.383	0.391	0.393	0.420
Running time/sec (NoCom)	0.312	0.312	0.312	0.312	0.312
Communication overhead/sec	0.070	0.071	0.079	0.081	0.108

Q3.b) Describe the patterns you see and hypothesize the reasons for those patterns. Why were the above-mentioned geometries chosen as the highest performing? Use the knowledge from these geometry experiments to perform the large core count performance study.

As needed, devise models, describe experiments and plot data to explain what you've observed. In explaining optimal processor geometry, consider the two components of performance: the cost of computation and the cost of communication. In general, the optimal geometry effects a balance between these two costs. Keep in mind that the cost of communication may be a function of direction.

For general cases, each core should do computation for a $\frac{N}{p_x} \times \frac{N}{p_y}$ matrix; for each iteration, each core should send $\frac{2N}{p_x} + \frac{2N}{p_y}$ data and receive $\frac{2N}{p_x} + \frac{2N}{p_y}$ data.

First, we consider computation's influence. For each point (i, j) in the matrix, we need to use data from position (i, j+1), (i, j-1), (i- $\frac{N}{p_y}$ -2, j) and (i+ $\frac{N}{p_y}$ +2, j) . Here, when $\frac{N}{p_y}$ is large, we will meet more conflict miss. However, the computation time of each geometry is nearly the same. As a result, the computation's influence on deciding the best geometry is small, or not larger than communication's influence.

Second and the most important, we consider comunication's influence. As we said, each core should send two rows of data and receive(and then place) two columns of data with their neighbours. Here, reading two rows of data or placing two rows of data is not costly for the data is in contiguous memory. However, reading two columns of data or placing two columns of data is time wasting because it can easily cause conflict miss. Therefore, we should try to make the column size($\frac{N}{p_x}$) small and row size ($\frac{N}{p_y}$) large, which means to make px large and py small. Besides, the total wait time is MAX(column transmission time, row transmission time). In this case, we need to balance the time between column data transmission and row data transmission and couldn't let the size of column be too small . After experiments, with the aim of getting best performance, we make py a little smaller than px and try to make $\frac{N}{p_y}$ in the range of 400~800.

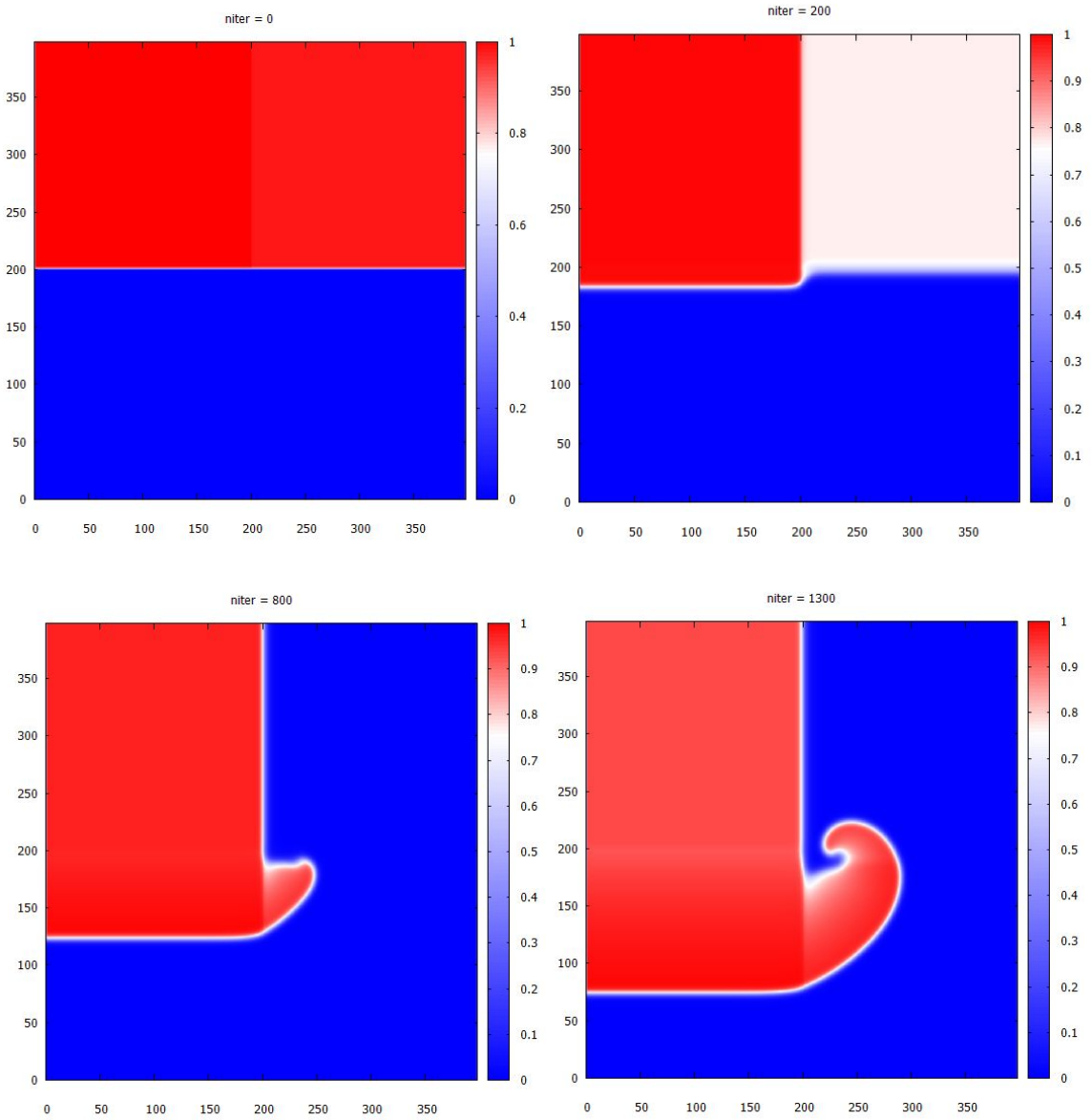
Section (4) Extra Credit

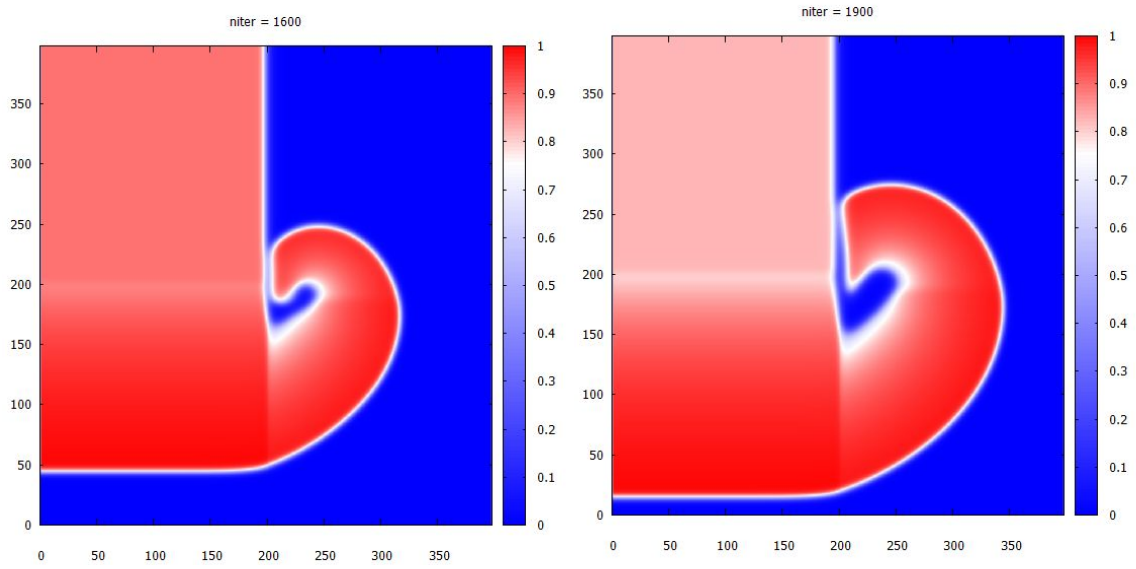
Q4.a) The current plotting routine only works on a single thread. Devise an efficient method of plotting results on a multi-core (MPI) application. Demonstrate your code by showing snapshot updates (at least 3) of an

evolving simulation. In your implementation, does the node that does the plotting also do computation? If so, is this a problem worth solving?

The starter code uses gnuplot and as far as we know it cannot accept multithreading input. So before submitting mesh to gnuplot, we need to somehow gather mesh from all processes. It is trivially the reverse of distributing initial conditions. So we reuse the code for the distribution and swap the input and output. Thus process 0 can get the whole mesh and submit to gnuplot.

In our implementation, the process 0 does computation as well as plotting. This would impose a serious overhead to performance as the gathering mesh involves a lot of communication, so maybe it is not worth solving. However, we think it would be better if there is another process which is dedicated to plotting and does asynchronous communication with computing processes to minimize the overhead to computation.





Q4.b) Either hardcode a SIMD implementation or using the intel compiler and various code restructuring and perhaps some pragmas, get the code to vectorize using AVX2 and report the performance increase. Describe what you need to do (flags, code changes, etc) to get the code to vectorize and show that it actually did vectorize. (Provide us with an easy way to turn these optimizations on or off - default should be off). Show the speedup on scalar code and on 48, 96 and 480 cores. Does the code scale as well as it did before aggressive vectorization? Explain why it does or does not. Referring back to the time * cores used product, does the result from Q2.f change?

We apply “#pragma simd” before inner loop in aliev_panfilov function. Also, we use a new makefile named Makefile.exp. In order to run the AVX version, you should use command “make -f Makefile.exp” instead of “make `cat OPTIONS.TXT`”.To prove we achieve AVX, we compare the assembly code before and after adding “#pragma simd” using command “objdump -d -M intel -S solve.o”. We find after adding “#pragma simd”, new instructions like “vaddpd” and “vaddsd” appear in the assembly code, which is the evidence of using AVX successfully.

We list the speedup results in the following table.

	np=48	np=96	np=480
Original Gflops	244.4	475.6	1085
AVX version Gflops	294.5	538.2	1306
Speedup	1.20	1.13	1.20

As we can see, we get similar speedup on different numbers of cores. We think the reason is that computation takes up most part of total time. AVX makes computation scale, then it can make the total time nearly scale.

Section(5) - Potential Future work

What ideas did you have that you did not have a chance to try?

1. We found that inner loops of unfused PDE and ODE computation are automatically vectorized by the compiler, which has increased performance compared to fused version, but we may still write AVX code by hand to further optimize it.

2. Initial condition distribution is not a bottleneck as it does not take place in the iteration but currently our code has to do a lot of memory copy/write to MPI_Scatter subblocks. Maybe we can simply send/receive those subblocks one by one.

3. We may use a separate process dedicated to plotting to minimize the overhead when one process has to compute and plot alternately.

Section (6) - References (as needed)

1.Open MPI Documentation https://www.open-mpi.org/doc/