PA3 Aliev-Panfilov

Amardeep Ramnani - A59005452
Debaditya Basu - A59005472
Mohit Shah - A59005444
Monil Shah - A59012111
GitHub Repo Link : https://github.com/cse260-sp22/hw3-pa3-aramnani-dbasu-m1shah-m3shah.git

Section 1: Developmental Flow

1.a) How the program works:

- The program starts by first taking the user input for deciding the size of the computation box, number of iterations and the processor geometry and it proceeds to allocate memory for the computation box for E, E_prev and R. The user input size is padded by 1 in both the X and Y direction to enable computation on the boundaries of the box for the stencil approach.
- Then the computation boxes E_prev and R are initialized where the right half-plane of E_prev is set to 1.0 and the left-half plane is set to 0. The bottom half-plane of R is set to 1 and top-half plane is set to 0.
- It is important to note that the same piece of code works on all the processors specified in the geometry and each processor has its own copy of the computation boxes.
- After initialization, the program proceeds to solve the Alien-Panfilov equations on multiple processors parallely. This starts with the Processor with rank 0 distributing the initial conditions to the other processors.
- Rank 0 calculates the size of the submatrix to be handled by each worker and its corresponding start index in the global scheme. After the index and size is obtained the corresponding data is packed for E_prev and R and are sent to the corresponding rank using MPI_Isend() calls and then rank 0 waits for the MPI_Isend calls to return.
- Each worker apart from Rank 0 worker issues a MPI_Irecv() call to get the initial conditions from Rank 0 and then unpacks the data and updates its own copy of E_prev and R.
- After the initial distribution is completed, the process proceeds to find out the X and Y coordinates of the worker Rank in the processor geometry. These coordinates help the rank to decide in which direction it would have to communicate the ghost cell data. The communication of each rank with its neighbors happens using MPI_Isend/Irecv APIs.
- The submatrix size for each rank is determined by first equally distributing along X and Y directions. Then if the X or Y coordinates of the rank falls below the remainder produced by division of matrix size by size of processor geometries the submatrix size is increased by 1 and then the final size is padded in both directions to allow ghost cell communication.
- While communicating with left/right neighbors each rank would have to pack the data since memory is organized in row major fashion. And while communicating with top/bottom neighbors the process can just specify the start address and size to send.
- Since we are using non-blocking Isend/Irecv APIs we need to synchronize and wait for the communication to complete and then the local E_prev is updated with data received.
- Before proceeding to the computation the program updates the padding region by copying the data from the boundaries to avoid computing single sided differencing.
- Then it proceeds to complete the ODE and PDE calculations using the stencil method and there is an option of using the fused or the unfused code. We have also provided an option to manually vectorize the ODE and PDE calculations.
- After all the iterations are completed, the program calculates the local LInf and sum of squares for each rank. Then the program uses the MPI_Reduce API to calculate the Global maximum Linf among all ranks and the sum of Squares for all ranks. The global sum of squares is used to compute the L2Norm. The global LInf and L2Norm are returned to end the program.

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

- We first compiled the starter code on sorken with the matrix size of 800, number of iterations 2000 and process geometry 1x1. This sets the baseline L2 norm and Linf values which would be helpful to verify our program.
- We started the process with reducing the number of computations to half for each core for a 1x2 geometry and we observed double performance on sorken. Earlier performance was 5GF/s for the size of 800, now we have the performance 11GF/s for the same size.
- Now, we started with distributing the data from rank 0 to other ranks (initial distribution). For this we used one MPI function called MPI_Scatterv, which distributes different size subarrays to all ranks defined. This routine is simple to implement, we created a vector data type for the subarrays with n/x and n/y size. Then, based on the processor geometry we distributed subarray elements using displacement and start index of MPI_Scatterv. We verified the same using print statements which confirms the correct output to local receive buffers of each rank.
- Now, we moved to ghost cell implementations. For the ranks having matrix edges, we copied the corresponding inner elements to the edges. Now, the edges have the correct data. Further, we used MPI_Isend and MPI_Irecv to send and receive ghost cells across, top, bottom, left, and right. Since Isend and Irecv are non-blocking calls, we need to use MPI_Waitall to wait for all ghost cells communication across all ranks before starting 5-point stencil computation.
- After ghost cell communication, we verified the local buffer outputs using print statements, and we observed the correct outputs. Then ,we wrote the code for 5-point stencil computation, on a matrix size of n/x +2 , m/y +2. We observed better performance on sorken with 2-D processor geometry, but the Linf and L2norm were not matching the reference outputs. This means there is something wrong with the code.
- While debugging, we used print statements to figure out the issue, and since multiple cores are printing at the same time. It was difficult to debug. We realized that there were some issues with the Scatterv section and we decided to move to MPI_Isend and MPI_Irecv for initial distribution of data to all ranks.
- Changing Scaterv to Isend and Irecv for distribution solved the problem. Now, the L2norm and Linf is matching with the reference outputs and the code was working fine for different matrix sizes.
- Now, we decided to implement our code on expanse. We first started with a shared node on expanse and we were getting a performance of 150GF/s for 16 cores, x 2 , and y 8, matrix size of 800 and number of iterations 100000. We tried different geometry but the highest was 150GF/s. We then moved to the compute node and we got the performance of 180GF/s for the same configuration.
- We decided to manually vectorize our code for performance. Adding AVX2 instructions would perform 4 double operations per cycle, which reduces the number of iterations, and helps in loop unrolling. We observed a performance boost on sorken as well with vectorization. Finally, we got the performance of 240GF/s for the same configuration on sorken along with auto vectorization, and a better performance for other sizes as well.

1.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.

- The initial condition distribution from worker 0 was initially using MPI_Send()/ MPI_Recv() APIs which are blocking calls. So it was waiting to send a submatrix of E to another rank and then start sending E_prev and subsequently R. So instead of using these blocking calls we used MPI_Isend/Irecv() non blocking calls to avoid waiting after sending calls for each E_prev and R.

- This same technique improves the performance during ghost cell communication. Initially we started with blocking MPI communication calls for each neighbor of a particular rank.
- We then replaced it with non-blocking MPI_Isend/Irecv() calls so that each rank could issue all the commands to communicate with all its neighbors and then wait together for all send/receive commands to complete. With this we observed better performance since all ranks were able to issue commands at the same time instead of waiting for each send/receive call to complete.
- For further performance improvement we implemented manual vectorization of the ODE and PDE computation using AVX2 intrinsics where we were able to process 4 doubles at once. We first verified the increase in performance due to vectorization on sorken and then tested it on the shared and compute nodes of Expanse.
- To finally reach 200GFs at 16 cores, every neighbor starts sending ghost cell data. To hide the communication latency we compute on the inner matrix first that doesn't require any ghost cell data to compute. Once that is complete we wait for the ghost cells data and compute on the outer ring. SO we are able to do meaningful work during the data transfer which speeds up our performance further.

Section (2) - Result

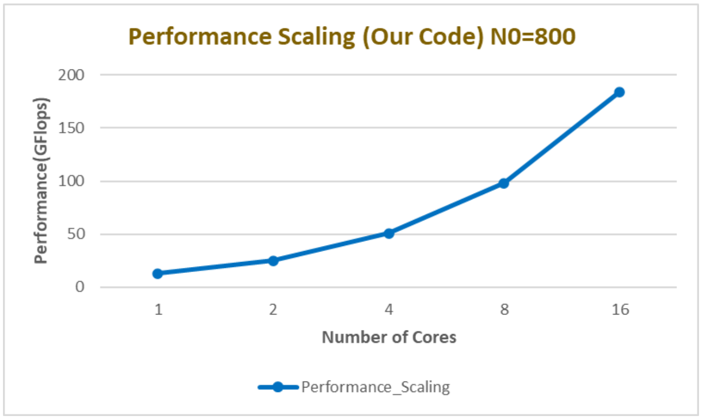
Q2.a) Compare the single processor performance of your parallel MPI code against the performance of the original provided code. Measure and report MPI overhead from 1 to 16 cores).

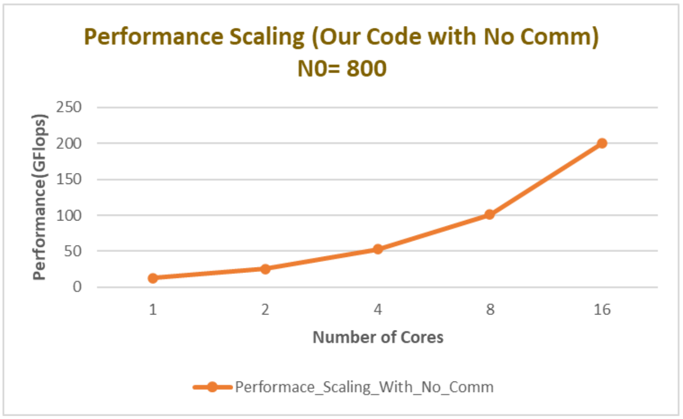
Cores	Geometry	GF/s (without Com)	GF/s(With Com)	MPI Overhead (%)
1	x=1,y=1	12.87	12.93	-0.4662004662
2	x=1,y=2	25.28	25.03	0.9889240506
3	x=1,y=3	38.67	38.46	0.543056633
4	x=1,y=4	52.35	51.04	2.502387775
5	x=1,y=5	65	62.8	3.384615385
6	x=1,y=6	77.5	74.5	3.870967742
7	x=1,y=7	89.6	86.52	3.4375
8	x=1,y=8	100.8	98.01	2.767857143
9	x=1,y=9	113	109.3	3.274336283
10	x=1,y=10	125	120.6	3.52
11	x=1,y=11	139.5	131.3	5.878136201
12	x=1,y=12	148.2	141.8	4.318488529
13	x=1,y=13	163.8	152.3	7.020757021
14	x=1,y=14	174.1	163.5	6.088454911
15	x=1,y=15	181.6	172.6	4.955947137
16	x=1,y=16	184	199	7.953976988

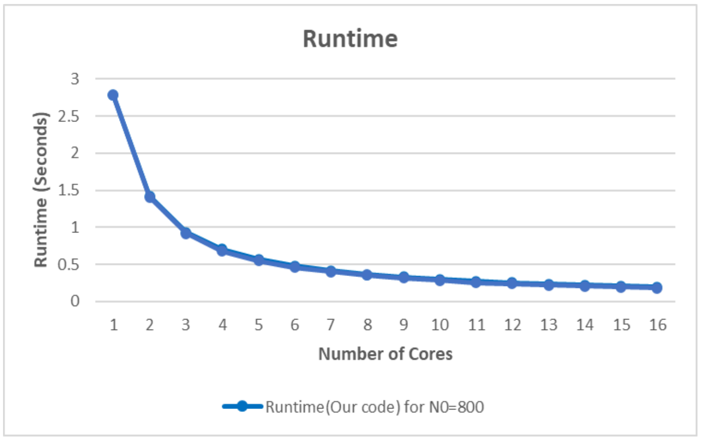
With the starter code, we observed a performance of 11.49GF/s for a matrix size of 800 and processor geometry of 1x1. We ran the same configuration on our code, and we achieved a slightly better performance with 12.93GF/s. The number of iterations for the run is 2000. The MPI overhead gradually increases from 1 core to 16. This can be attributed to the fact that improvement in speedup with adding multiple cores is lesser than increase in communication for initial distribution of data and ghost cell communication for smaller sizes of matrix. The marginal reverse overhead in case of 1x1 might be because of the additional branch statements for noComm case to ensure correct padding causing less auto vectorization.

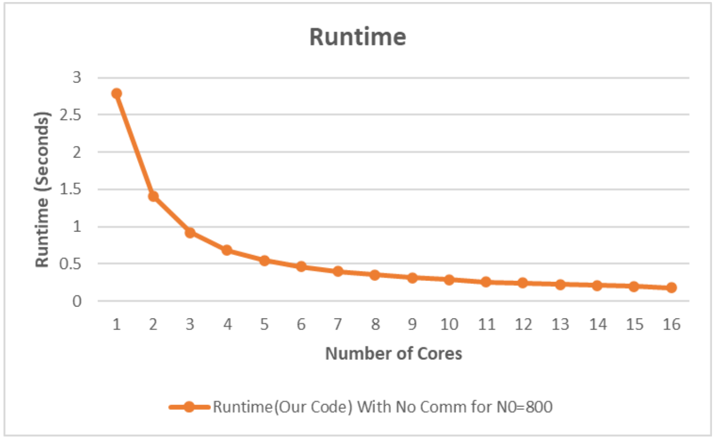
Q2.b) Conduct a strong scaling study: observe the running time as you successively double the number of cores, starting at p=1 core and ending at 16 cores on Expanse, while keeping N0 fixed.

Cores	**Geometry**	**Run time (seconds)**	**GFlops**	**Run time with no communication (seconds)**	**GFlops(no Comm)**
1	x=1,y=1	2.78	12.93	2.78	12.87
2	x=1,y=2	1.42	25.03	1.41	25.28
4	x=1,y=4	0.702	51.04	0.68	52.35
8	x=1,y=8	0.365	98.01	0.355	100.8
16	x=1,y=16	0.194	184	0.181	199.9



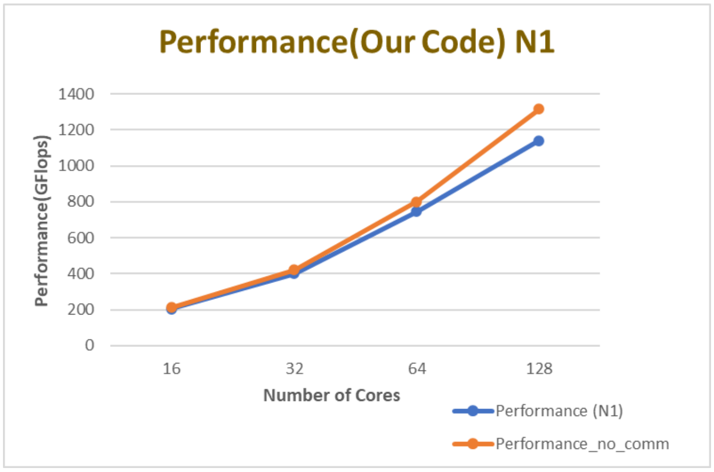




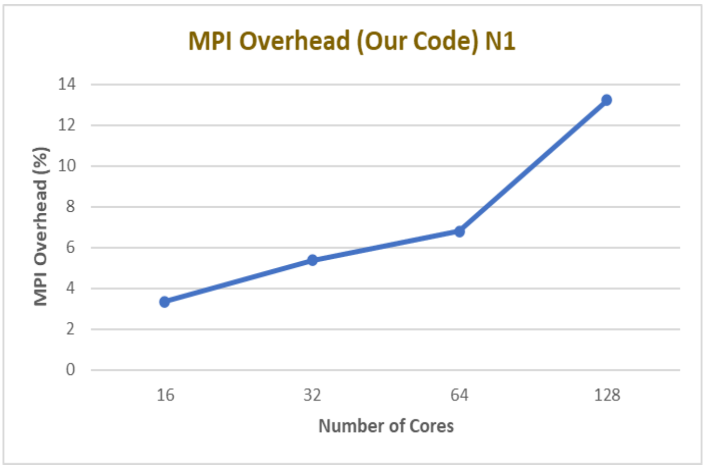


As can be seen in the table above, the decrease in runtime almost saturates for the same matrix size and number of iterations when increasing computing cores. The decrease in runtime is more at the start than at the end. The overall benefit in runtime decrease is the difference between time saved from using multiple cores and the time spent in ghost cell/init communication. The rank 0 distribution and ghost cell communication for smaller matrix sizes starts to outweigh the speedup achieved using multiple cores with increasing cores. The performance can be seen scaling fairly straight with a number of cores and performance is higher without communication vs with communication.

Q2.c) Conduct a strong scaling study on 16 to 128 cores on Expanse with size N1. Measure and report MPI communication overhead on Expanse. Supplement your discussion of scaling/overhead (i.e. what is the cost of communication).



N1 performance vs cores



N1 MPI overhead (%) vs cores

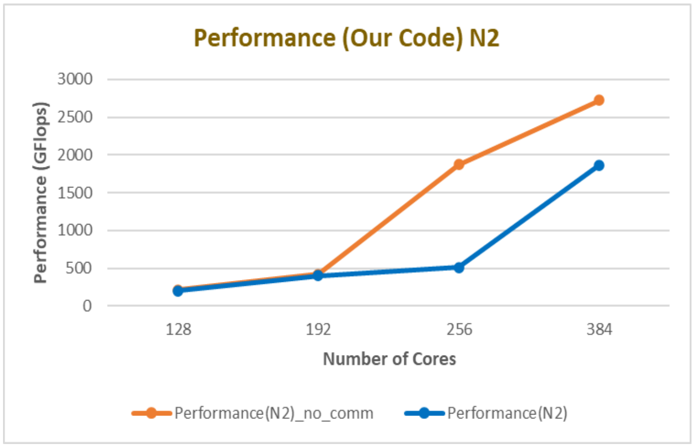
Cores	Geometry	Throughput without Comm(GFlops)	Throughput with Comm(GFlops)	MPI Overhead (%)
16	x=2, y = 8	212.4	205.3	3.342749529
32	x=4, y=8	422	399.3	5.379146919
64	x=4, y=16	800.4	746	6.796601699
128	x=8,y=16	1315	1141	13.23193916

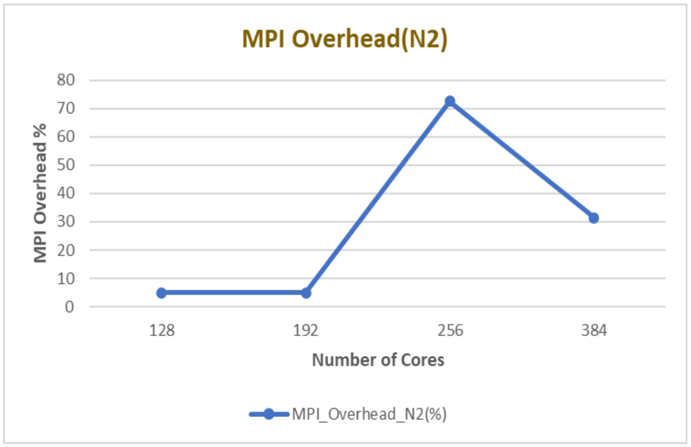
As can be seen from the two grap1

Q2.d) Report the performance study from 128 to 384 cores with size N2. Measure the communication overhead. Use the knowledge from the geometry experiments in Section (3) to perform these large core count performance studies. Don't do an exhaustive search of geometries here as that will eat up your allocation.

Cores	Geometry	Throughput without Comm(GFlops)	Throughput with Comm(GFlops)	MPI Overhead(%)
128	x=8, y=16	212.7	202.1	4.983544899
192	x=12, y=16	420	399.4	4.904761905
256	x=16, y=16	1873	512.8	72.62146289

384	x=3, y=128	2723	1866	31.47264047





N2 performance vs cores

N2 MPI Overhead (%) vs cores

As can be seen the performance increases with increasing cores. There is a sharp increase in performance at 256 cores for no communication case but not for the communication case. This also leads to a lot of MPI overhead which can be seen in the other graph. One possible reason for this is the geometry size. With a geometry size of 16x16 in case of 256, the communication required to pack and unpack ghost cells and left/right cells for a square geometry is reasonably higher than the computation speedup that is achieved through higher cores. The same is not true for 384 cores because the geometry favors more row major communication that would be repetitive for top/down ghost cell communication.

Q2.e) Explain the communication overhead differences observed between <=128 cores and >128 cores.

As can be seen from the Tables above, the communication overhead for cores <=128 and N1 as 1800 is increasing linearly with the number of cores, whereas for N2 as 8000, the overhead first increases from 192 to 256 and then decreases from 256 to 384**.** The benefit from adding the number of cores in case of 128 shows that the added cores are able to utilize the computation more than the communication overhead. The distribution of nodes for 384 cores is what governs the communication overhead.

Q2.f) Report cost of computation for each of 128, 256 and 384 cores for N2. What is the most optimal (from a cost point of view) based on resources used and computation time? Explain.

Cores	**Geometry**	**Execution Time(s)**	**Computation Cost = #cores x computation time (N x seconds)**
128	x=8, y=16	70.95	9081.6
192	x=12, y=16	35.89	6891.5328
256	x=16, y=16	27.95	7155.2
384	x=3, y=128	7.60	2919.70944

We can observe from the table that as the number of cores are increasing, the computation cost, which is the number of cores multiplied by the execution time, is also decreasing. For the matrix size of 8000 with number of iterations also set to 8000, 384 cores computation time is just 7.6 seconds as compared to the computation time of 27.95 seconds with 256 cores. With 384 cores, we achieved the optimal performance with geometry x=3, and y=128. In the left/right direction, we have to do packing and then send the ghost cells, and then unpack before the core can use the ghost cells. In the top/down direction, since the matrices are stored in row-major order, we can send the whole row as ghost cells without the need of packing and unpacking. So, in 384 cores with minimum x, the cost of communication would be less as compared to other processor geometries.
Also, as we increase the number of cores, the amount of computation that each core needs to perform per unit time decreases. So, 384 cores can do more computation for the same matrix size as compared to other cores in the same unit time. Hence, the computation cost with 384 cores is minimum and optimal for N2.

Q3.a) For p=128, report the top-performing geometries at N1. Report all top-performing geometries (within 10% of the top).
It is a completely non vectorized case.

N (Matrix Size) = 1800

Core (P = 128)	**Geometry**	**Performance(GFlops)**
128	x=2, y=64	671
128	x=16, y=8	678
128	x=4, y=32	701
128	x=8, y=16	729

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?

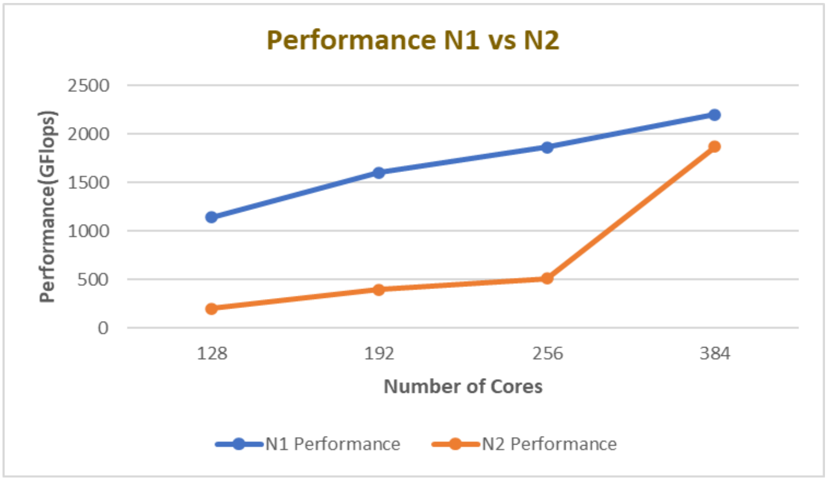
The optimal geometries will manage the trade off between communication and computation well. The computation time will decrease with an increasing number of processors but the communication time changes with the number of cores. We observed that the optimal geometries for a higher number of processors was moving more towards square shapes where it achieves a high ratio of computation to communication cost. We don't see an exact square shape for 128 since it is not a perfect square. Square shape geometry is also optimal for 256 cores.

In the stencil method, for one iteration elements from a given row and the row to its top and bottom are accessed. So if these three rows were to be put into the L1 cache at the same time there would be lots of reuse and enable faster computation. In Expanse, the L1 Cache Size is 32 KB which means it can hold 4096 Doubles. So for every submatrix which is calculated by one processor the maximum X direction size would be (4096 / 3 rows) which is approximately 1365 without incurring misses. So geometries with a number of processors more than (N / 1365) in X direction will have better computation costs. So we have observed that geometries (4,32) and (8,16) had lesser cache miss rates than (2,64).
But we also need to consider the communication costs. If there are more processors in X direction then there needs to be more packing and unpacking done since the data sent to left and right neighbors is in columnar fashion from row-major order data which reduces performance. So the best performing geometries turned out to be Px=4,Py=32 and Px=8,Py=16 with comparable performance for the number of processors P = 128.

Strong and Weak Scaling (4)

Q4.a) Run your best N1 geometry (or as close as you can get) at 128, 192, 256 and 384 cores. Compare and graph your results from N1 and N2 for 128, 192, 256 and 384 cores.

Core	Geometry	N1 GF/s	N2 GF/s
128	x=8, y=16	1141	202.1
192	x=12, y=16	1604	399.4
256	x=16, y=16	1863	512.8
384	x=3, y=128	2200	1866



As we can see from the plot, the performance increases for both N1 and N2 with an increase in the number of cores. The curve for N1 is approximately linear , while the curve for N2 is linear till 256 cores, and then increases sharply from 256 to 384 cores.
N2 size is 8000, while N1 size is 1800. So, if we run N2 on a processor geometry, there would be more number of elements per core as compared to that of N1. Since, for N2 each core needs to perform more computations as compared to that of N1, we see N1 has better performance for the same processor geometry as compared to N2. Also, with increase in matrix size, the transfer of ghost cells among cores increases, which makes the communication expensive. So, the performance slows down with more communication. Hence, in N2 we have expensive communication and less computation per unit time as compared to N2. That's why we see a huge performance difference between N2 and N1 for the same processor geometry. Thus we see there is strong scaling in performance for N1 from 128->384 but weak scaling for N2 from 128->384

Q4.b) Explain or hypothesize differences in the behavior of both strong scaling experiments for N1 and N2? As needed, devise models, describe experiments and plot data to explain what you've observed and justify your hypotheses.

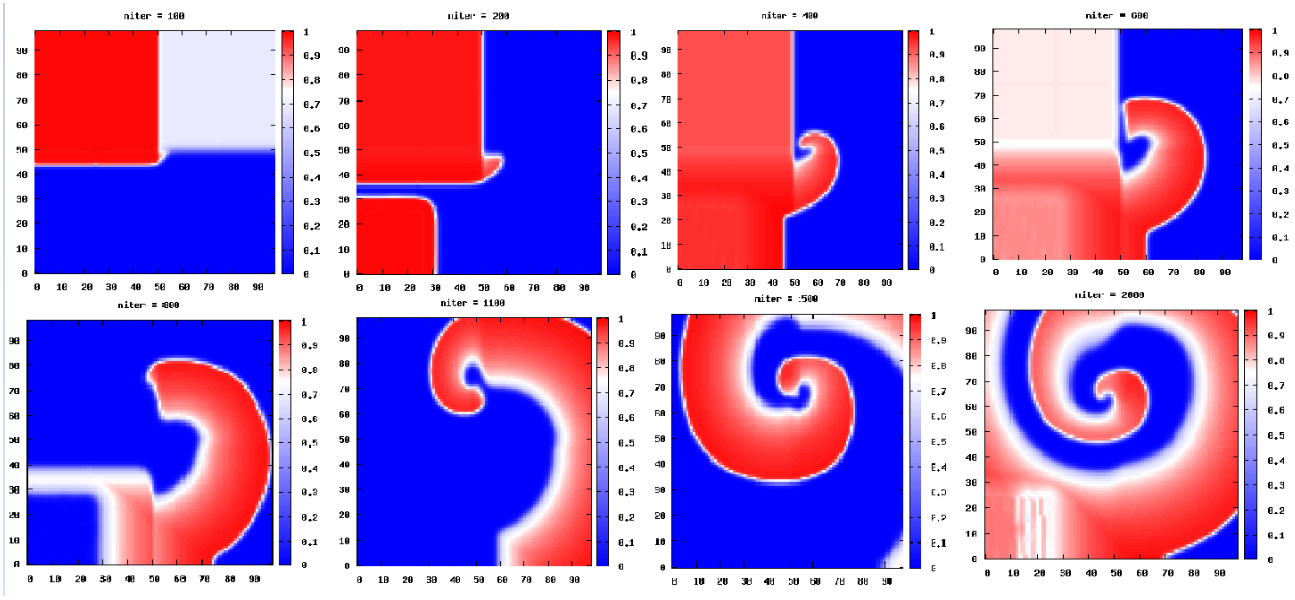
We can observe from the previous table and plot that as we increase the number of cores, the performance increases. With more cores, the amount of data a core needs to compute decreases. It means the computation per unit time would increase with the number of cores, that's why performance improves.

For N1=1800, as the number of cores increases, the performance increases in a linear fashion. The plot is approximately linear. In expanse, we have 128 cores in a node. As we move from 128 to 192, we are using two nodes. The nodes communicate over infiniband, which increases the communication overhead. So, moving from one node to another increases communication overhead. But, also with more cores, we get more computation per core. Hence, we observe an approximate linear performance with N1=1800.

For N2, the performance first improves in a linear fashion from 128 cores to 256 cores. But, with 384 cores we see an approximate exponential increase. With 384 cores, we achieved the optimal performance with geometry x=3, and y=128. As mentioned in the cost of computation question (2.f), when we have less number of cores in x-direction, the cost of communication per node for 384 cores would be less as compared to other cores configurations. This boosts the performance with 384 cores for N2. Also, the matrix size for N2 is huge, and the number of elements per core may not fit in the processor's cache. This will cause a lot of capacity misses and slow down the performance, which we don't observe in N1 plot. When we make the number of cores 384, the number of elements per core decreases , and there would be less capacity misses with 384 cores as compared to 128, 192, or 256. This would also improve the performance. And finally, as the number of elements per core decreases, the computation per unit time increases, which further improves the performance. That's why we observe exponential plots when moving from 256 to 384 cores for N2.

Extra Credit (5)

1. **Plotting**
   To change the code which would support multi threaded plotting, we implemented gather logic that is almost the reverse of our scatter logic. By sending back data to Rank 0 every plot_frequency number of iterations, we are able to observe the plot below. We send the data back to rank 0 using normal MPI_Send and MPI_Receive and unpack the received data at appropriate indexes within rank 0. In our implementation rank0 did computation, distribution as well as gathering which is why it is not a very fruitful problem to solve. If we had one additional rank that just collects data for plotting that would be faster. This can be reasoned by the fact that every rank would send messages using asynchronous calls and the gathering node would wait until it receives all the messages and then plots. The problem is hence not worth solving if one core is doing computation as well as plotting.



2. **Vectorization**
   We saw a huge performance improvement with auto vectorization as shown below. Without vectorization statistics were obtained by passing "-fno-tree-vectorize" to the C++FLAGS argument in

Makefile. Manual vectorization statistics were obtained by adding "-msse -msse2 -mavx2 -DSSE_VEC -fno-tree-vectorize" to Makefile and our manual intrinsics were ifdef'ed under "SSE_VEC". Auto vectorization was run with "-mavx -march=core-avx2"

*objdump -d apf | grep xmm | wc -l (**763 instances**) (make mpi=1 vec=1 and auto vectorize on)*
*objdump -d apf | grep xmm| wc -l (**290 instances**) (make mpi=1 vec=1 and auto vectorize off)*
*objdump -d apf | grep xmm| wc -l (**0 instances**) (make mpi=1 and auto vectorize off)*

The performance gain from auto vectorization is more than manual vectorization because our manual vectorization was only done on the PDE and ODE solver computation portion, whereas the compiler has optimized other loops within the code as well like ghost cell communication, Array packing routines during MPI_Send etc. We also observed the effect of auto vs manual vs no vectorization by checking the assembly code as shown above.

Cores	**Matrix Size**	**Geometry**	**Iteration**	**Without Vectorization (GF)**	**Manual Vectorizati on (GF)**	**Auto vectorization (GF)**
128	1800	x=8,y=16	100000	729	1420	1950
192	1800	x=12,y=16	100000	978	2000	2727
256	1800	x=16,y=16	100000	1175	2090	2607
384	1800	x=8,y=48	100000	1676	2980	3842

Possible Future Work (6)

There were a couple of things that we could have tried and will be part of future works

1. Possibly try out logarithmic scattering as mentioned in the discussion section and see its impact.
2. Play around more with compiler optimizations like prefetching hints.
3. Some more design space exploration of the optimal geometry sizes.
4. Some experiments with objdump to know what code gets generated. We were able to do preliminary analysis to see the effect of auto vectorization vs manual vectorization.
5. Some analysis of cache usage using cachegrind was done but it could not help us a lot
6. We tried casually checking performance counters and wanted to use them in some analysis
7. We also wanted to see how MPI applications can be profiled using TAU but the report generated was difficult to comprehend without some parser.

References (7)

1. https://www.open-mpi.org/doc/v3.1/man3
2. https://cseweb.ucsd.edu//~baden/Doc/docs/MPI_Guide.pdf
3. https://www.codingame.com/playgrounds/283/sse-avx-vectorization/autovectorization
4. https://www.rookiehpc.com/mpi/docs/mpi_isend.php
5. https://www.cs.uoregon.edu/research/tau/home.php