PA3 Report

Section (1)

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.]

Out code runs in the following steps:

1. Memory allocation (helper.cpp): Same for (**E**, **E_prev**, **R**)
   In this step memory is allocated according to the process rank. The bigger matrix along with it's padding (cb.m+2✕cb.n+2) is allocated in process **0**. For all the other processes, we allocate only the tile that the process will compute (along with the padding to store the ghost cells).

   The calculation of tile size is done using the following formula (this is used everywhere else as well to compute tile size ny and nx):

   processRowIndex = my_rank / (cb.px);
   processColIndex = my_rank % (cb.px);
   nx = (cn.n/cb.px) + ((processColIndex - cb.n%cb.px) < 0 ? 1 : 0)
   ny = (cn.m/cb.py) + ((processRowIndex - cb.m%cb.py) < 0 ? 1 : 0)

   This makes sure that tile sizes for any two processors along a given dimension do not differ by more than 1.

2. Initialization (helper.cpp):
   Process **0** initializes the bigger matrix according to starter code. For **E_prev**, the right half-plane is set to 1 (not including the padding). For **R**, the bottom half-plane is set to 1 (not including the padding). All other cells are set to 0.

   For all other process ranks we set all the cells in both **E_prev, R** to be 0.

   Distribution:
   Once the matrices are initialized in process **0**, the tiles that will be computed by each matrix are sent to the respective process ranks. Following are steps:

   a. Calculate the tile dimensions for particular rank.
   b. Construct the **strided vector data-type** for the tile dimensions.
   c. Send it to the process rank.
   d. After the sending loop, process **0** overwrites its own **E_prev** and **R** to match its tile.

   All other processes receive the tile from **0** and write it in their **E_prev** and **R.**

E is not initialized, only allocated.

3. Solve (solve.cpp): Ghost cells communication.
   At this point every process will have its own tile to compute on (along with the padding on the side). Solve is done in the following steps:

    a. Calculate the 2D coordinates for the process rank.
    b. Calculate the tile size for the process.
    c. Check in which direction the process needs to send ghost cells. This is done by checking the 2D coordinates of the process rank.
        i. If the process rank is at the top boundary, it will not send ghost cells to the top and it will not receive ghost cells from the top. Instead to speed up the calculation, we copy the second row into the top padding. Similarly, if process rank is at the bottom boundary, left boundary or right boundary.
    d. For communication **MPI_Isend** and **MPI_Irecv** are used. For transferring the right and left ghost cells, a strided vector is used.
    e. Since non-blocking communication is used, the inner-matrix (all elements except at the boundary) are computed while the processes wait for Irecv.
    f. After Wait for receive completes, the boundary elements are computed (using loops similar to started code).
    g. Finally, **stats()** method computes the **mx** (max element) and **sumsq** (squared sum of each element) for each tile. **MPI_REDUCE** (sent to process **0**) is used with **MPI_SUM** to calculate the squared sum of all elements of the complete matrix. Once the sum arrives in process **0**, L2 is calculated. Similarly, **MPI_REDUCE** is used to send the max element to process **0** which is then stored in Linf.

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

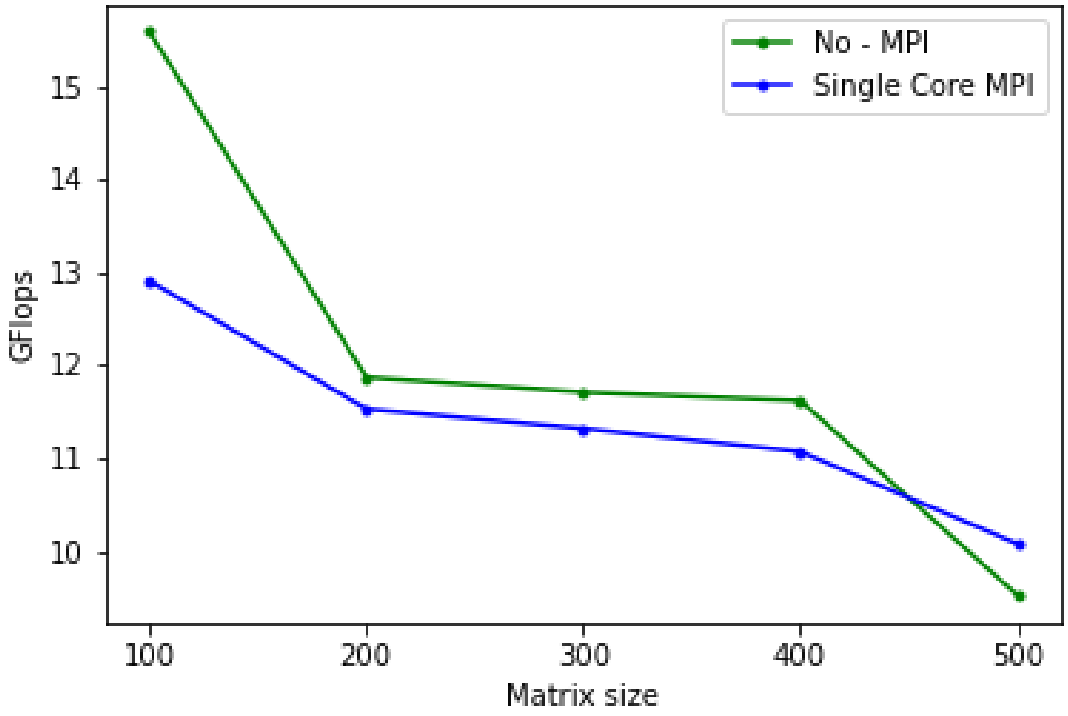
- For the allocation and initialization part, first we tried to only allocate and initialize the tiles to be computed by each process. This created some issues with calculations and even distribution. So we decided to just initialize the complete matrix in 0 and send the tiles to each process.
- For the solve part, we started with using MPI_SEND and MPI_RECV without using the strided vector data-type for columns. This helped us to make sure send buffers and receive buffers are calculated correctly. This was done in general for 2D geometries. Also, MPI_Reduce was added.
- Once the above worked, we changed communication to MPI_Isend and MPI_Irecv to enable computation of the inner matrix while communication is taking place. We also tried changing the loop structure for both fused and unfused but our experiments did not give us substantially better results.

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.

1.	Distribution of the bigger matrix without using strided vectors and creating temporary buffers. Not using Isend and Irecv	This gave us correct results and helped us to check our buffers but did not give good scaling.
2.	Use of strided vectors in communication (both initialization and ghost cells).	Gave us somewhat better results especially since now a lot of temporary buffers used in distribution we re removed
3.	Using Isend and Irecv and doing inner matrix computations while waiting for ghost cells to arrive.	This gave us much better performance which was still slightly below the performance goals.
4.	Adding the Pragma for GCC target avx2	Adding the compiler target improved our performance by over 20%.

Section (2) (Note: px -> number of columns in processor geometry, py: number of rows)

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).



As we can see, the single core performance with MPI enabled is very similar to the performance when MPI disabled (and original code is used) for different matrix sizes. With MPI disabled, single core performance is slightly higher which can be attributed to compilation time, since the new code has a bunch of macros which need to be parsed by the compiler.

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.

N0 = 800

Cores	Geometry (px*py)	GFlops	#Iterations	Time with communication	Time without communication	MPI overhead
1	1*1	12.15	10000	13.4593	13.47	~0 (as expected)
2	1*2	29.85	20000	14.1849	14.153	~0
4	1*4	57.46	40000	13.7219	12.1153	1.6s
8	2*4	113.4	80000	12.5421	12.5324	~0
16	2*8	206.2	120000	10.7379	10.053	0.7s

We can see that the performance increases almost proportionally to the increase in the number of cores. There is ~100% scaling as the processor count is doubled. This can be attributed to the fact that since the processor count is low, there is little communication happening.

In the MPI overhead, we can see that since the communication is happening in a single node, the overhead is not very high. However, since the number of cores are very low, they can still be fragmented inside a node which can lead to a little overhead, as seen for the 1*4.

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).

Cores	Geometry (px*py)	GFlops	iteration	Time with communication(s)	Time without communication(s)	MPI communication overhead (s)	scaling
16	2*8	235.9	26000	10.004	10.068	~0	100% (base)

32	4*8	445.7	50000	10.1692	10.1643	~0	94.47% (from 16)
64	8*8	805.1	100000	11.2681	11.0825	~0	85.32% (from 32)
128	2*64	1313	120000	13.82	13.79	0.03	81.6% (from 64)

The running time fluctuates very much. Sometimes, it is likely that time without communication is even more than that with communication, which is impossible as the communication overhead should be positive. One reason for this fluctuation is maybe physically, due to differences in resource allocation within the node.

We can see that the scaling has an expected rate each time the number of cores is doubled. It is expected because with the increase in the number of cores, the amount of communication increases.

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.
Please report your geometries in this Google Form: (as well as this table in your report)
https://forms.gle/2CNW9Kzr9mNTEnCp8

Cores	Geometry	GFlops	iteration	Time with communication(s)	Time without communication(s)	MPI communication overhead per iteration(s)	scaling
128	2*64	452.7	3000	10.881	10.883	~0	100%
192	12*16	478.2	3000	11.24	11.23	0.01	70% (from 128)
256	16*16	651	4000	12.71	12.62	0.09	90% (from 192) 72% (from 128)
384	16*24	1658	12000	12.97	12.6134	0.3566	173% (from 192) 122% (from 128)

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

From 128 to 192 cores, the computation cost increases. We propose that as we increase the number of cores more than 128, we allocate more than 1 node which has the maximum of 128 cores. Therefore, allocating 192 cores will cost inter-node communication and initialization, which will definitely cost extra computation cost. However, this extra computation cost will be amortized as the number of cores increases further, e.g. 256.

Q2.f) Report cost of computation for each of 128, 256 and 384 cores for N2.

Cores	Computation Cost = #cores x computation time(per iteration) (s)	time(s)	iteration
128	0.46	10.881	3000
192	0.719	11.24	3000
256	0.813	12.71	4000
384	0.415	12.87	12000

What is the most optimal (from a cost point of view) based on resources used and computation time? Explain.

384 cores is the best option with lowest computational cost.
From the result, we can see that:
1. From 128 to 192 cores, the computation cost increases. We propose that as we increase the number of cores more than 128, we allocate more than 1 node which has the maximum of 128 cores. Therefore, allocating 192 cores will cost inter-node communication and initialization, which will definitely cost extra computation cost. However, this extra computation cost will be amortized as the number of cores increases further, e.g. 256.
2. From 192 or 256 to 384 cores, the computation cost decreases. We propose that as we increase the number of cores, the tile for each core to take care of will have a smaller size. Usually, a smaller size of tile means the ratio between its border elements and its internal elements will increase, which means the ratio of their time spent will also increase. Since N=8000 is very huge and thus, usually there are more internal elements, this ratio is initially very low and as the tile size becomes smaller, the ratio will increase and become more balanced to 1. Since we use IRecv and ISend function, the computation of internal elements and communication of border elements are going at the same time. Thus, this more balanced time spending will definitely increase the overall efficiency increase. Also, in 192 cores, we found that when the geometry change from 8*24 to 12*16, sizes of 4 sides of the border will become more balanced and the computation cost will improve384 cores is the best option with lowest computational cost.

From the result, we can see that:

1. From 128 to 192 cores, the computation cost increases. We propose that as we increase the number of cores more than 128, we allocate more than 1 node which has the maximum of 128 cores. Therefore, allocating 192 cores will cost inter-node communication and initialization, which will definitely cost extra computation cost. However, this extra computation cost will be amortized as the number of cores increases further, e.g. 256.

2. From 192 or 256 to 384 cores, the computation cost decreases. We propose that as we increase the number of cores, the tile for each core to take care of will have a smaller size. Usually, a smaller size of tile means the ratio between its border elements and its internal elements will increase, which means the ratio of their time spent will also increase. Since N=8000 is very huge and thus, usually there are more internal elements, this ratio is initially very low and as the tile size becomes smaller, the ratio will increase and become more balanced to 1. Since we use IRecv and ISend function, the computation of internal elements and communication of border elements are going at the same time. Thus, this more balanced time spending will definitely increase the overall efficiency increase. Also, in 192 cores, we found that when the geometry change from 8*24 to 12*16, sizes of 4 sides of the border will become more balanced and the computation cost will improve slightly.

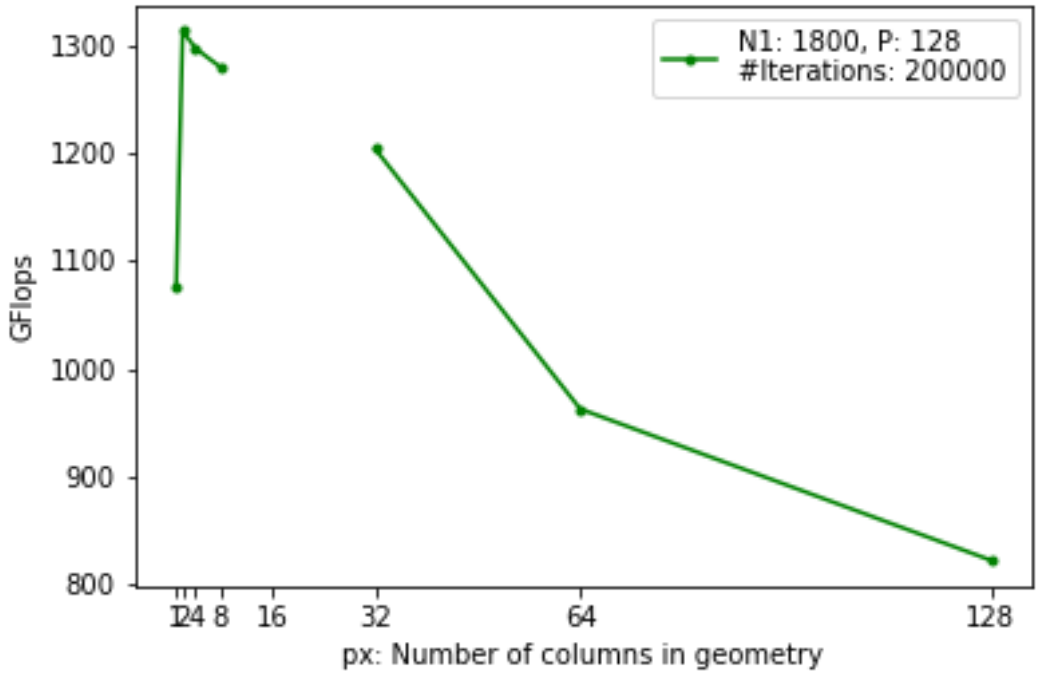
Section (3)

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

Top performing geometries:

Geometries	GFlops
2*64	1313
4*32	1297
8*16	1279

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 affects a balance between these two costs. Keep in mind that the cost of communication may be a function of the communication direction.



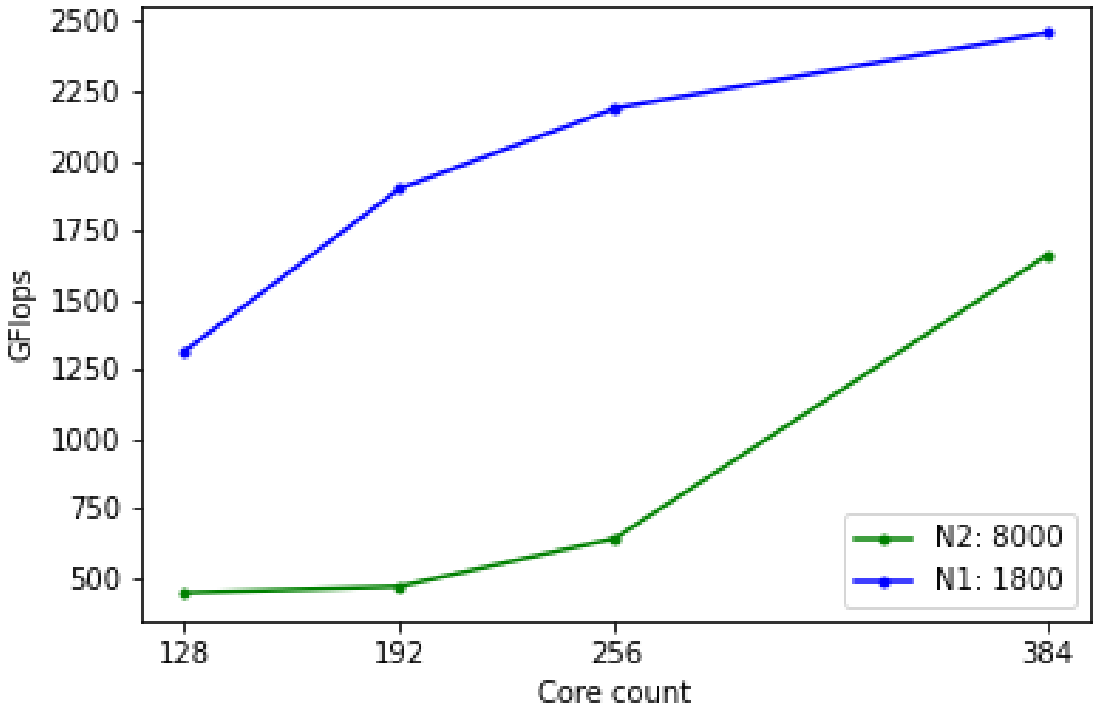
The above plot shows the change in performance for core count 128 (N1: 1800), as the number of columns increases in the geometry. Note that there is a missing point for geometry 16*8. For some reason we could not get the task to complete within the time limit for this geometry.

From this pattern it is clearly visible that having a high column count in geometry is not good. This is explained by the fact that the higher the number of columns we have in the geometry the more is the amount of columns that need to be communicated across processors. Since, the matrices are stored in row major order, creating a buffer for column (to send) takes more time. Also, storing this in the padding takes more time as well. On the other hand, sending rows is cheaper since the blocks to be sent are contiguous.

However, we see that increasing the columns in the beginning increases the performance. This can be attributed to lowering in the number of cache misses since now there are lower elements in a row that can be stored together in the cache.

Section (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.



The above plot compares the best performance achieved for N1 and N2 for processor counts [128, 192, 256, 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.

In N1 we can see that the scaling drops from around 95% (16 to 32) to around 81% (64 to 128). This is inline with the expectations since the amount of communication is increasing with the increase in the number of processors. Also, since the matrix sizes are the same, the computation time decreases, which also does not contribute to the scaling.

In N2, however, scaling happens in a little haphazard way. Going from 128 to 256 we get a scaling of 72%. The reduction in scaling might be explained by the introduction of a new node which increases the communication time (due to increase in the distance between cores). From 128 to 192 we also see a scaling of around 70% which can be again due to the increase in the number of nodes. However, even with the further increase in the number of nodes, scaling from 128 to 384 or 192 to 384 is very high (above 100%). Although this seems a little out of line, we hypothesize that this might be due to the good pipelining of communication and computation time (since the computation of the inner matrix is now taking much shorter time).

The differences between the scaling patterns can be attributed to the increase in the number of nodes which leads to increase in communication time between some cores. Also, with the increase in the number of cores, we get optimal geometries which have higher number of columns which can lead to a difference in scaling

Section (5) Extra Credit

EC-a) (SEE INSTRUCTIONS)
EC-b) (SEE INSTRUCTIONS)

Section(6) - Potential Future work

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

- Try SIMD vectorization for computing fused and unfused loops.
- Try different compiler architectures.
- Try this on a multi-core(MPI) application.

Section (7) - References (as needed)

- Class slides
- MPI documentation: https://rookiehpc.github.io/index.html