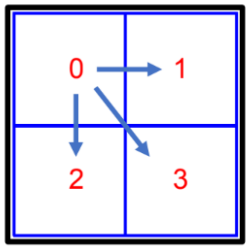
Section (1) - Development Flow

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.]

Our implementation is described as:

1. All processes initialize a local copy of a matrix that is the same size as the global matrix with ghost cells around them.
2. Process 0 sets up an initial condition that consists of one-half plane of 0's and one-half plane of 1's to the matrix of E and R, and distributes to all other processes as the figure(borrowed from discussion) below illustrates.



During the distribution, we ensure:

Even distribution: in order to divide the mesh as evenly as possible such that the amount of work assigned along a given dimension will differ by not more than one row (and/or column) of the solution array for all workers, we implement the partition such that:
For each worker process:

Given a matrix of size m * n and block grid of x * y, initialize the size of the partition to be the average(round down) size:

rows per process = m / y

cols per process = n / x

Then we distribute the the remainder rows and cols to worker processes:

For remainder y = m % y, we assign to the first remainder y rows of the processes/block grid. For remainder x = n % x, we assign to the first remainder x column of the processes/block grid.

MPI_Isend and MPI_Type: in order to distribute to different worker processes, we define a MPI_type(tile) which is the size of the largest partition to send. Then we use MPI_Isend

to send the partition to all the processes. **MPI_wait** is called on all send requests for synchronization.

3. All worker processes run **MPI_recv** to receive the partition sent by process 0. Upon receiving, the worker processes will place the partition at the top left corner of their local matrix(inside ghost cell).
4. After all processes have finished the initial process, they all enter the iterations of computation of their own partition, which includes:

```
if on the top boundary:
  fill the top ghost cell by copying
else:
  send topmost row to top adjacent process
  receive and fill row of ghost cells from top adjacent process

if on the left boundary:
  fill the left ghost cell by copying
else:
  send leftmost column to left adjacent process
  receive and fill column of ghost cells from left adjacent process

if on the bottom boundary:
  fill the bottom ghost cell by copying
else:
  send top row to bottom adjacent process
  receive and fill row of ghost cells from bottom adjacent process

if on the right boundary:
  fill the right ghost cell by copying
else:
  send rightmost column to left adjacent process
  receive and fill column of ghost cells from right adjacent process
```

Above is a pseudo code for the preparation each process performs for the computation. For each side (top, down, left, right) of the partition:

Ghost cell filling: each process first checks if their partition is at the boundary of the whole matrix. If yes, then fills the matrix by copying the symmetric cell of the boundary.

Ghost cell exchanges: if not, then initialize a MPI_Isend to send the row/column to the adjacent process and call MPI_Irecv to expect a row/column to be sent from the corresponding adjacent cell and placed at the ghost cell upon receiving. Note here we defined a MPI_Type for sending a column. The exchange of the right/left column to an adjacent process is illustrated in this figure(borrowed from discussion).

5. **MPI_wait()** is called on all processes so that all processes finish setting up the ghost cells before starting computation.

6. Then every process performs the computation on the partition of the matrix, where the loops are fused. Each process loops over each row j in the partition, for each element i, perform:
    a. Get the values of the current elements in the E, E_prev, and R.
    b. Update the E value by adding the weighted sum of the neighboring E_prev values and subtracting a term that depends on the E_prev and R values.
    c. Update the R value.
    d. Store the updated E and R values back into the E and R arrays.
7. **MPI_Barrier()** is called after the computation is done and before the loop goes into the next interaction for synchronization.
8. After all interactions are performed, every process calculates the square sum of their partition and max value. Then **MPI_Reduce()** is called with **MPI_MAX** and **MPI_SUM** so that the final statistics are calculated and updated by process 0 after receiving the results from all processes.

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

We start by implementing the simplest possible MPI mechanism using 1D tiling where cb.px = 1 and cb.px = y (commit1 f272d9), which works well for small numbers of cores.

Next we implement 2D to handle two-dimensional geometries(commit abb113a), where we incorporate MPI_Type for sending the columns.

In order to further optimize the code, we first try incorporating MPI_reduce() to do the validation(commit 420feb9). Then, we try implementing asynchronous (non-blocking) sending during initialization (commit 550f238). In addition, we try some compiler optimization during which we create temporary variables to hold the value of the local matrix (commit d1911c8) and try prefetching and decide to not include the prefetching due to performance drop.

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.

Development Step	Performance (GFLOPS=)
Starter	10
1D (1 x 16)	135.8

2D (4 x 4)	28.41
2D (4 x 4) w/o comm	190
Non-blocking initialization (1 x 16)	137.4
Compiler optimization	173.8
Prefetch	91.8

We measured the performance of our development using a matrix N0, where N = 800 and the number of iterations = 2000. The initial starter code, which lacked any parallelization, achieved a baseline performance of 10 GFLOPS. The development process began with a simple parallelization using a 1D implementation (1 x 16), which boosted performance to 135.8 GFLOPS by introducing basic parallelism.

Next, we transitioned to a 2D implementation (4 x 4). However, this change significantly reduced performance to 28.41 GFLOPS, primarily due to increased communication overhead among the processors. To address this, we tested the 2D implementation by removing the estimated communication overhead, which led to a substantial performance increase to 190 GFLOPS. This demonstrated the significant impact of communication efficiency on overall performance.

We then tried several optimizations. Non-blocking initialization, allowing process 0 to send partitions asynchronously, slightly improved performance to 137.4 GFLOPS, indicating marginal gains from reduced synchronization delays. Instead of accessing the array every time, we created temporary variables to hold the values of e_tmp and r_temp, which reduced memory access time and improved performance to 173.8 GFLOPS. Lastly, we experimented with adding prefetch instructions to optimize memory access patterns. Contrary to expectations, this step reduced performance to 91.8 GFLOPS, suggesting that the overhead introduced by prefetching outweighed its benefits in this specific context.

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	GF w/o comm	Overhead
1	NA	10	NA	NA

1	1 x 1	12.87	12.94	0.5%
2	1 x 2	25.23	25.44	0.8%
4	1 x 4	50	51.9	3.7%
8	1 x 8	95.1	101.8	6.6%
12	1 x 12	135.6	149.2	9.1%
16	1 x 16	173	196	11.7%

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	Running Time (sec)
1	1 x 1	2.784
2	1 x 2	1.42
4	1 x 4	0.717
8	1 x 8	0.377
12	1 x 12	0.264
16	1 x 16	0.208

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	GF	GF w/o comm	Communication Overhead
16	1 x 16	211.1	216.6	2.5%
32	2 x 16	396.5	418.9	5.3%

64	8 x 8	695.9	744.1	6.5%
128	8 x 16	1129	1317	14.3%

The strong scaling study shows that while the application scales reasonably well up to 32 cores, the scaling efficiency deteriorates as the number of cores increases further, primarily due to the increasing cost of communication. As the number of cores increases, the amount of communication required between processes also increases, leading to higher communication overhead. Moreover, the geometry change in both dimensions can induce more required communication as the inner processes need to communicate four sides each iteration.

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/fbUd5FJeHUBHBNf89

Cores	Geometry	GF	GF w/o comm	Overhead
128	8 x 16	420.3	431.1	2.5%
256	4 x 64	610.2	1209	49.5%
192	1 x 192	506	536.9	5.8%
384	3 x 128	1571	2453	36.0%

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

The higher communication overhead for core counts beyond 128 cores can be attributed to the inter-node communication. Since 1 node has at most 128 cores, a larger core requirement will need more than one node to perform. When the computation involves

multiple nodes, data needs to be transferred across the interconnect network, which can introduce significant latency and overhead compared to intra-node communication.

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

Cores	Computation Cost = #cores x computation time
128	4364.8
256	6014.7
192	5439.3
384	3504.0

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

The most optimal from a cost point of view is using 384 cores with a geometry of 3 x 128. Although the computation overhead percentage for 384 cores (36%) is higher than for 128 cores (2.5%) and 192 cores(5.8%), the computation time savings achieved by using 384 cores is significant enough to outweigh the computation overhead cost. Since the reduction in computation time is substantial, the overall cost savings still make 384 cores more optimal.

Section (3) - Determining Geometry

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

Geometry	Gflops
1*128	896.2
2*64	**1039**

4*32	**1088**
8*16	**1123**
16*8	708
32*4	625
128*1	443

All performances are recorded in the table above. The top-performing geometry is (2 x 64), (4 x 32), and (8 x 16), which is within 10% of the top performance of 1123 GFLOPS.

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.

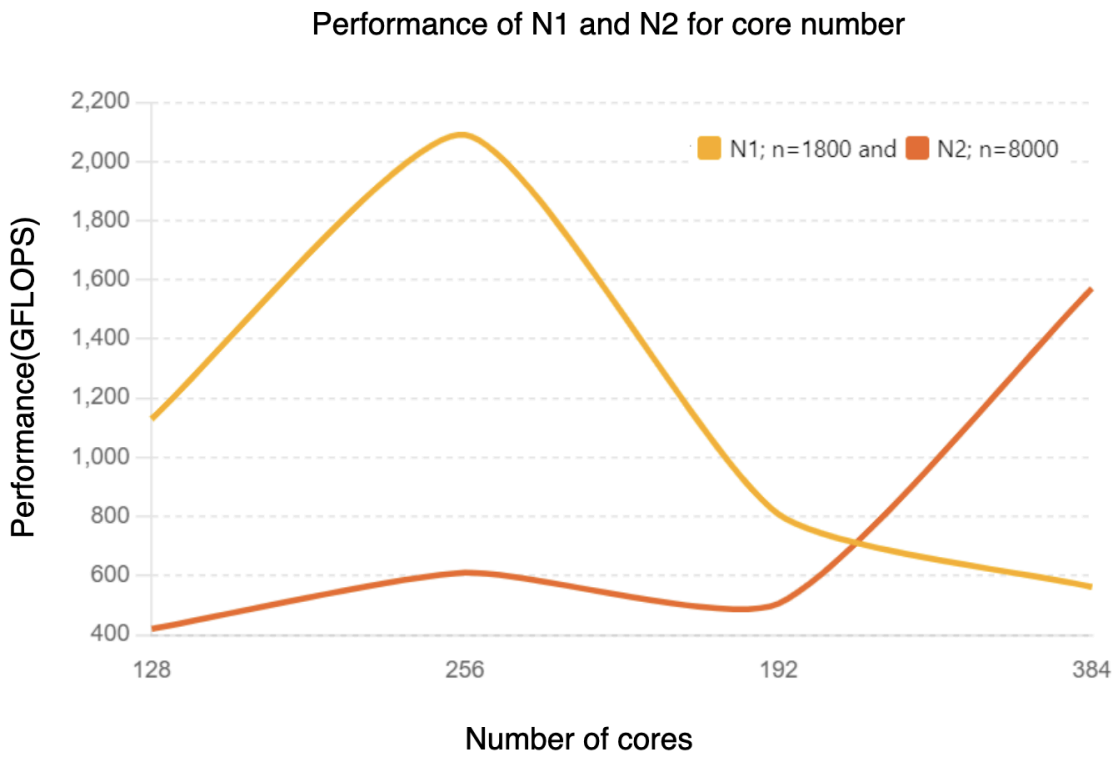
In the provided data, the top-performing geometry is (8 x 16), which achieves the highest Gflops (1123). The pattern we observe is that the performance initially increases as we move from a geometry of 1 x 128 to 8 x 16, and then decreases as we continue to 128 x 1.

1. **Computation cost:** we attribute the different computation cost due to geometry to cache behavior. For large x values such as 128, the data access pattern is such that only one element from the row is accessed at a time. Given the row-major order of the matrix, accessing one element at a time results in frequent cache misses. For large y values though, which perform the continuous linear memory access, the problem is very cache friendly. However, increasing the elements in a row to a value that is not a multiple of the cache line size or overflowing the cache line (64 bytes for our machine) can also negatively impact the performance.
2. **Communication cost:** given a matrix of size N and a geometry of (x , y) where xy = p, in our 2D partition, the problem is divided into tiles of (N/x, Np/x). Assuming the communication cost is same for all cores, the communication overhead is in the order of O(N/x + N/(p/x)). Taking the derivative of it gives (x^2-p/px^2). Setting it to zero and solving for x gives us an optimal value of x. Therefore, for an optimal x we would have x = p ^ (1/2) .
3. **Balance between:** for a small number of processes, computation cost dominates (for example p = 16 the overhead is ~23 GFLOPS), where we can observe that the geometry of 1 x 16 has the best performance over other geometries. However, as the process increased to p =128, the communication cost (~ 188 GFLOPS)

dominates. Hence the better geometry for communication cost has the best performance.

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.



N1; n=1800

Cores	Geometry	GF	execution time
128	8 x 16	1130	34.1
256	4 x 64	2090	4.35
192	1 x 192	808	1.12

384	1*384	562	1.61

N2; n=8000

Cores	Geometry	GF	execution time
128	8 x 16	420.3	34.11
256	4 x 64	610.2	23.50
192	1 x 192	506	28.33
384	3 x 128	1571	9.12

Q4.b) Explain or hypothesize differences in the behavior of both strong scaling experiments for N1 and N2?

N1 Calculations (n=1800)

1. From 128 to 192 cores: Scale(128, 192) = (34.1061 / 1.12281) * (128 / 192) = 30.3868 * 0.6667 = 20.258:

   This high scaling factor means very good performance improvement, it is because the computation is distributed evenly across the increased number of cores. The problem size (n=1800) may still be large enough to effectively use the additional cores without excessive communication overhead.

2. From 192 to 256 cores: Scale(192, 256) = (1.12281 / 4.34546) * (192 / 256) = 0.2585 * 0.75 = 0.1939

   This bad scaling means bad performance. The addition of cores may introduce significant communication overhead and synchronization costs, which outweigh the benefits of adding more cores. Additionally, the workload may not be large enough to fully use 256 cores efficiently, leading to underutilization of resources.

3. From 256 to 384 cores: Scale(256, 384) = (4.34546 / 1.61446) * (256 / 384) = 2.691 * 0.6667 = 1.794

   The moderate scaling factor means although we can add more cores, but the benefit is minimal. The problem size (n=1800) may be too small to benefit

significantly from such a high number of cores, leading to increased overhead and less efficient scaling.

N2 Calculations (n=8000)

1. From 128 to 192 cores: Scale(128, 192) = (34.1061 / 28.3298) * (128 / 192) = 1.204 * 0.6667 = 0.803

   This scaling factor means a bit of improvement. The larger problem size (n=8000) helps in better utilizing the additional cores, but communication overhead and synchronization is still a big problem. The workload is kind of similar compared to N1, but there are still inefficiencies that prevent better scaling

2. From 192 to 256 cores: Scale(192, 256) = (28.3298 / 23.4951) * (192 / 256) = 1.206 * 0.75 = 0.904

   A scaling factor close to 1 indicates good scaling. The larger problem size (n=8000) allows the system to benefit more from the increased number of cores, with improved load balancing and reduced overhead. However,due to communication costs and the limits of parallelism, the performance is not perfect

3. From 256 to 384 cores: Scale(256, 384) = (23.4951 / 9.12498) * (256 / 384) = 2.575 * 0.6667 = 1.717:

   This high scaling factor suggests very good performance improvement. The problem size (n=8000) is large enough to fully utilize the additional cores, with little bit of communication overhead. The larger problem size benefits more from parallelism, leading to really good scaling.

Impact of Problem Size:

- **N1 (n=1800):** The smaller problem size results in poor scaling, especially as the number of cores increases beyond 128. This is due to increased communication overhead that outweigh the benefits of parallelism.
- **N2 (n=8000):** The larger problem size exhibits better scaling across all intervals. The computation-to-communication ratio is more favorable, allowing the system to benefit more from parallelism and efficiently utilize the additional cores.

Communication Overhead:

- **N1 (n=1800):** Higher communication overhead relative to the computation results in poor scaling performance, particularly noticeable in the transition from 192 to 256 cores.
- **N2 (n=8000):** Lower communication overhead relative to the larger computation load leads to better scaling performance. The impact of communication overhead is less significant compared to the larger problem size.

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?

Due to the limited resources, here are some methods we have not tried.

1. Interleaving computation for ghost cell communication.
2. Allocate specific memory for each process instead of a global copy.
3. Hand vectorization.

Section (7) - References (as needed)

https://sites.google.com/ucsd.edu/cse260-spring-2024/assignments/assignment-3-background?authuser=2&pli=1