Development Flow

For every process, we first find the respective rank of the process. Upon finding the rank of a process, we calculate the position assignment of the process in the processor grid. Initially, we calculate the base number of rows and columns of the global mesh to be handled by a process, such that every process will at least handle the base number of rows and columns. We then calculate the number of additional rows 'r' and columns 'c' that need to be distributed to ensure the difference between the largest and smallest subdivisions is at most one. We distribute the remaining 'r' rows among the first 'r' processes. Each of these 'r' processes will handle one additional row. Similarly, we distribute the remaining 'c' columns among the first 'c' processes. Each of these 'c' processes will handle one additional column.

Algorithm 1 Even Distribution of Work

```
Input: myrank, px, py, n, m
Output: mesh_myrows, mesh_mycols

pgrid_mycol ← myrank mod px
pgrid_myrow ← ⌊myrank/px⌋

mesh_mycols ← ⌊n/px⌋
mesh_myrows ← ⌊m/py⌋

if m mod py ≠ 0 then
    if pgrid_myrow < m mod py then
        mesh_myrows ← mesh_myrows + 1
    end if
end if
if n mod px ≠ 0 then
    if pgrid_mycol < n mod px then
        mesh_mycols ← mesh_mycols + 1
    end if
end if

return mesh_myrows, mesh_mycols
```

Upon calculating the number of rows "mesh_myrows" and columns "mesh_mycols" of the global mesh to be handled by each process (ensuring the condition for equal distribution of work such that no processor has more than 1 row or column more work than any other processor), we allocate the three state variables arrays with size equal to (mesh_mycols+2)*(mesh_myrows+2) .

Initial Data Distribution :-

If the process rank is 0, for every other process rank, we iteratively calculate the number of rows and columns of the global mesh to be handled by each process and also the column offset and row offset as per the processor grid position of the process respective to the global mesh. Then we allocate only two temporary state variables arrays (proc_E_prev & proc_R) with the respective sizes for those processes. Using the respective row and column offset, we pack the inner block of the temporary state variable arrays of the respective processes from the global mesh. Then we send the two temporary state variables arrays using two MPI_Send calls. For process rank other than 0, using MPI_Recv calls, they receive their respective two state variable arrays into the buffers of the arrays allocated before.

```
if (myrank == 0) {
    for (int proc = 1; proc < nprocs; proc++) {
        Allocate proc_E_prev and proc_R arrays by calculating each process' count of rows and columns;
        Pack the inner block of the arrays from the global mesh using each process' offset;
        MPI_Send(proc_E_prev, tot_pts, MPI_DOUBLE, proc, 0, MPI_COMM_WORLD);
        MPI_Send(proc_R, tot_pts, MPI_DOUBLE, proc, 1, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(my_E_prev, mytot_pts, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(my_R, mytot_pts, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Ghost Cell Communication:-

We create two data types for handling communication of rows and columns of ghost cells between neighboring processes. We use MPI_Type_vector for handling communication of columns of ghost cells such that blocklength is 1 , count is 'mesh_myrows' and stride is 'mesh_mycols+2'. Similarly , we leverage MPI_Type_contiguous for handling communication of rows of ghost cells such that count is 'mesh_mycols'.

```
MPI_Datatype ghost_column;
MPI_Type_vector(mesh_myrows,1,mesh_mycols+2, MPI_DOUBLE, &ghost_column);
MPI_Type_commit(&ghost_column);

MPI_Datatype ghost_row;
MPI_Type_contiguous(mesh_mycols, MPI_DOUBLE, &ghost_row);
MPI_Type_commit(&ghost_row);
```

Before performing ghost cell communication, we check if either of the dimensions of the processor grid are not equal to 1. Then, depending on the position of the process in the processor grid, we perform ghost cell communication of the respective number of rows and columns using MPI_Isend and MPI_Irecv calls to the neighboring processes. For processors

```
if(pgrid_myrow == CASE){
MPI_Isend(my_E_prev + ..., 1, ghost_row, ..., 0, MPI_COMM_WORLD, &reqs[req_count++]);
MPI_Irecv(my_E_prev + ..., 1, ghost_row, ..., 0, MPI_COMM_WORLD, &reqs[req_count++]);
.
.
}
if(pgrid_mycol == CASE){
MPI_Isend(my_E_prev + ..., 1, ghost_column, ..., 0, MPI_COMM_WORLD, &reqs[req_count++]);
MPI_Irecv(my_E_prev + ..., 1, ghost_column, ..., 0, MPI_COMM_WORLD, &reqs[req_count++]);
.
.
}
```

Boundary cases on the edges of the global matrix:-

Depending on the position of the process in the processor grid, it may handle a subdomain which lies on the boundary of the global mesh. For such processes , we fill the respective rows and columns in the padding region by copying data from the boundary of the computational block of the respective process.

While waiting for the asynchronous MPI calls to complete, we performed computation of the differential equations of the inner block cells of the computational box as these cells do not depend on the ghost cells. After completing ghost cell communication, we complete the computation of the differential equations of the boundary cells of the computational box.

After running the program for a certain number of iterations, each process computes its local Linf and sumSq variables. We leverage MPI_Reduce calls to accumulate the results at the root process and perform proper calculations of L2 and Linf using the accumulated results.

Development Process

We started the development process with implementation of 1D processor geometry using OpenMPI on our local machines. Initially our implementation could only support processor geometries that were divisible by the mesh grid size and ghost cell communication along rows. We used blocking synchronous send operations to ensure there was no deadlock occurring in the program. Once the program was running without errors, we added the logic for handling even distribution of work even when the processor geometries were not divisible by global mesh size. Then we implemented asynchronous send and receive operations in the program to support ghost cell exchange. We verified the correctness of our program using the provided code. We achieved strong scaling for the processors ranging from 1 to 16. Then we started with implementation of 2D processor geometry. We implemented ghost cells communication along columns by loading a static array with ghost cells values along the column and sending the array using MPI. Upon receiving the array, we implemented logic to unpack the array and load the values along the respective column of the computational block. In order to boost performance, we leveraged MPI_Type_vector for communication of ghost cells along column and MPI_Type_contiguous for communication of ghost cells along row. We interleaved computation of the inner block of a process' computational block during ghost cell communication for performance improvement. Thus we broke the computation of differential equations for the inner block and boundary cells in two different parts. In the initial data distribution step, we realized it was not necessary to send the state variable array E as it was not being used and was being computed in the initial iteration. Hence, we reduced the amount of data being distributed from process 0 to other processes by just distributing the other two state variables. In order to reduce memory usage in processes with rank other than 0, we modified the alloc1D and init functions such that other processes did not need to allocate and initialize state variables with size greater than that of their respective computational block.

Optimizations Performed During Development

1. Using Asynchronous MPI calls for ghost cell communication
    a. Initially performed synchronous MPI operations for ghost cell exchange in 1D processor geometry.
    b. Performed asynchronous MPI operations for ghost cell exchange in 1D processor geometry and 2D processor geometry.
2. Using MPI_Type_vector
    a. Initially performed ghost cell exchange along a column by first loading the column in a static array and then performing MPI calls for sending and receiving arrays. Then unpacking the array received such that we can fill the ghost cells along a column in the state variable.
    b. Leveraged MPI_Type_vector to handle ghost cell exchange along a column.
2. Interleaving Computation with Communication
    a. Initially performed differential equations computation on the whole computational box of the respective process after performing ghost cell exchange.
    b. Interleaved differential equations computation on the inner block of the respective process' computational box with ghost cell communication. After completing ghost cell exchange, performing differential equations computation only on the boundary of the respective process' computational box. The performance improvement observed was very minor.
3. Reduced Memory Usage in Processes in other than rank 0

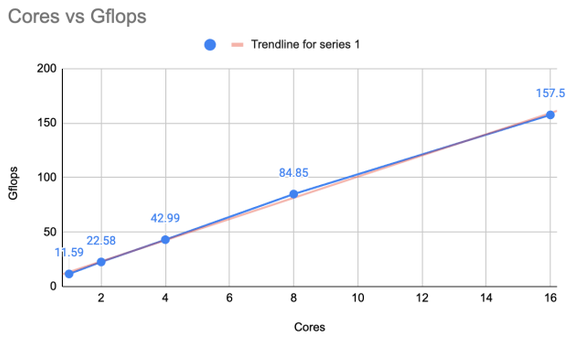
a. Initially, for each process , the state variables were being allocated and initialized with size equal to that of the global mesh.
b. Modified allocation and initialization functions such that for processes other than with rank 0, the state variables are being allocated and initialized with size equal to that of their respective computation box, reducing memory usage.

Result

Compare the single processor performance of your parallel MPI code against the performance of the naive algorithm.

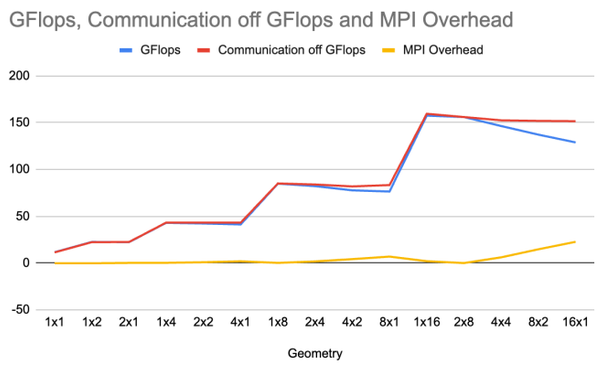
Reference:					
Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead	Computation Cost = #cores x computation time
1	1x1	11.25	11.27	0.02	3.18691
16	1x16	156.2	162.3	6.1	3.67192

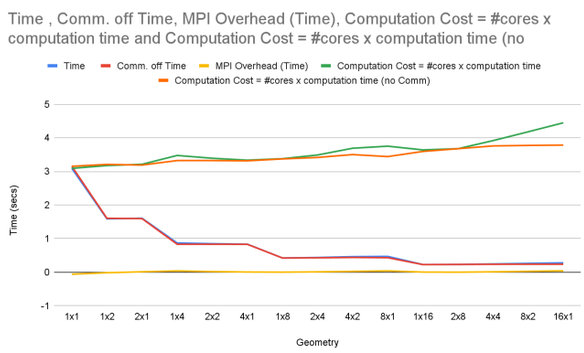
Cores	Gflops
1	11.59
2	22.58
4	42.99
8	84.85
16	157.5



Processor Performance on different geometries:

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead
1	1x1	11.59	11.37	-0.22
2	1x2	22.58	22.34	-0.24
2	2x1	22.32	22.48	0.16
4	1x4	42.99	43.24	0.25
4	2x2	42.26	43.14	0.88
4	4x1	41.25	43.14	1.89
8	1x8	84.85	85.02	0.17
8	2x4	82.18	83.93	1.75
8	4x2	77.7	81.88	4.18
8	8x1	76.42	83.31	6.89
16	1x16	157.5	159.5	2
16	2x8	156	156	0
16	4x4	146.4	152.5	6.1
16	8x2	137.2	151.9	14.7
16	16x1	128.9	151.6	22.7



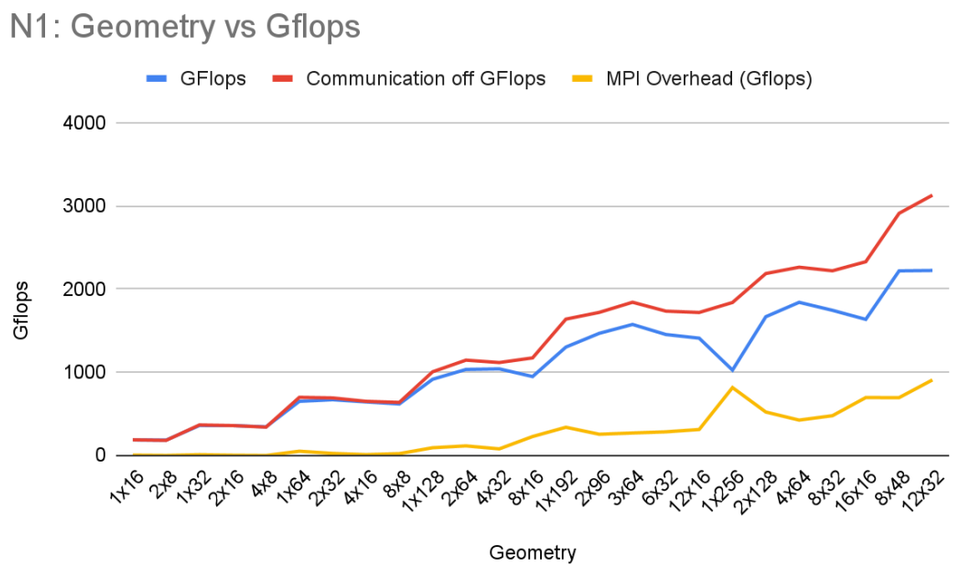


Cores	Geometry	Time	Comm. off Time	MPI Overhead (Time)
1	1x1	3.09111	3.15232	-0.06121
2	1x2	1.58706	1.60431	-0.01725
2	2x1	1.60606	1.59446	0.0116
4	1x4	0.86885	0.830796	0.038054
4	2x2	0.848048	0.830718	0.01733
4	4x1	0.833687	0.828883	0.004804
8	1x8	0.422401	0.421535	0.000866
8	2x4	0.436137	0.427018	0.009119
8	4x2	0.461278	0.437699	0.023579
8	8x1	0.468997	0.430212	0.038785
16	1x16	0.227521	0.224658	0.002863
16	2x8	0.229792	0.229778	0.000014
16	4x4	0.24483	0.234983	0.009847
16	8x2	0.261261	0.23587	0.025391
16	16x1	0.278025	0.236485	0.04154

Strong scaling study on 16 to 128 cores on Expanse with size N1.

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead (Gflops)
16	1x16	184	185.2	1.2
16	2x8	181.2	178.5	-2.7
32	1x32	357.2	363.8	6.6
32	2x16	356.3	356.4	0.1
32	4x8	340.9	337.7	-3.2
64	1x64	647.9	696	48.1
64	2x32	667	688.5	21.5

64	4x16	640.8	647.8	7
64	8x8	615.8	634.9	19.1
128	1x128	913.1	1003	89.9
128	2x64	1031	1143	112
128	4x32	1038	1114	76
128	8x16	945.4	1170	224.6
192	1x192	1300	1636	336
192	2x96	1465	1717	252
192	3x64	1572	1839	267
192	6x32	1451	1732	281
192	12x16	1407	1716	309
256	1x256	1023	1836	813
256	2x128	1665	2184	519
256	4x64	1838	2260	422
256	8x32	1742	2217	475
256	16x16	1633	2326	693
384	8x48	2216	2908	692
384	12x32	2221	3126	905



The communication overhead is higher for geometries with higher x. Scaling Efficiency:

16 to 32 cores: GFlops roughly doubles, indicating good scaling. MPI overhead remains low, with some negative values suggesting possible inefficiencies in smaller configurations. **32 to 64 cores**: GFlops again nearly doubles, but MPI overhead increases, especially in the 1x64 configuration. **64 to 128 cores**: GFlops shows significant increase, but MPI overhead becomes substantial, particularly in the 8x16 configuration, indicating diminishing returns due to communication overhead.

- **Increasing MPI Overhead**: As the number of cores increases, MPI communication overhead grows significantly. This is expected as more cores lead to more inter-process communication, which increases the latency and synchronization costs. Configurations like 8x16 on 128 cores show very

high MPI overhead, which indicates that the benefits of adding more cores are being offset by the communication costs.

- **Optimal Configurations**: The 2x64 configuration on 128 cores shows a reasonable balance with high GFlops and moderate MPI overhead compared to the 8x16 configuration. Smaller configurations such as 1x16 or 2x8 on 16 cores and 1x32 or 2x16 on 32 cores provide better scaling efficiency with minimal communication overhead.
- **Negative Overhead Values**: Negative values for MPI overhead in some configurations (e.g., 16 cores 2x8, 32 cores 4x8) could indicate anomalies or optimizations where parallelism actually helps reduce some computational bottlenecks.

Performance study from 128 to 384 cores with size N2.

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead (Gflops)
128	2x64	196.5	200.3	3.8
128	4x32	195.9	200.5	4.6
128	8x16	197.8	202.5	4.7
192	4x48	388.9	407.1	18.2
192	8x24	391.6	406.3	14.7
192	12x16	392.6	401.9	9.3
256	4x64	466.5	1202	735.5
256	8x32	478	1188	710
256	16x16	482.9	1160	677.1
384	2x192	1841	2248	407
384	3x128	1738	2458	720
384	4x96	1865	2243	378
384	8x48	1717	2436	719
384	12x32	1437	2432	995

Explaining the communication overhead differences observed between <=128 cores and >128 cores.

The communication overhead between cores shows a noticeable difference when scaling from ≤128 cores to >128 cores. The main difference observed in our data is that communication overhead increases significantly when the number of nodes exceeds one. This increase is primarily due to the following reasons:

1. **Inter-node Communication**: With more than one node, data transfer between nodes introduces latency and bandwidth limitations, as compared to intra-node communication where cores share faster, local memory.
2. **Network Contention**: As the number of nodes increases, more communication occurs over the network, leading to congestion and increased contention for network resources, thereby increasing overhead.
3. **Synchronization Costs**: Larger numbers of nodes require more complex synchronization mechanisms to ensure data consistency, adding to the communication overhead.

In summary, the shift from ≤128 cores to >128 cores involves moving from primarily intra-node communication to inter-node communication, which significantly increases communication overhead due to latency, bandwidth limitations, network contention, and synchronization costs.

Cost of computation for each of 128, 256 and 384 cores for N2.

Cores	Geometry	Computation Cost = #cores x computation time	Computation Cost = #cores x computation time (no Comm)
128	2x64	9338.4832	9160.6656
128	4x32	9369.0496	9151.1296
128	8x16	9275.0336	9062.4512
192	4x48	7077.3312	6761.0304
192	8x24	7028.256	6774.9312
192	12x16	7010.1312	6848.9088
256	4x64	7867.6992	3053.7728
256	8x32	7677.7984	3089.9456
256	16x16	7600.7168	3163.5968
384	2x192	2989.45152	2449.14048
384	3x128	3168.2304	2239.27296
384	4x96	2952.50304	2453.85984
384	8x48	3205.66272	2259.67872
384	12x32	3829.64736	2264.00256

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

Larger N/P has more favorable computation to communication ratio:

128 Cores Configurations: **8x16 (9275.0336)** is the most optimal due to the lower computation cost. The smaller grid size results in a more favorable surface area to volume ratio, reducing communication overhead. **192 Cores Configurations**: **12x16 (7010.1312)** is the most optimal. The higher number of divisions (12x16) likely reduces the perimeter, balancing the computation and communication more efficiently. **256 Cores Configurations**: **16x16 (7600.7168)** is the most optimal.. **384 Cores Configurations**: **4x96 (2952.50304)** is the most optimal. The grid size ensures that communication overhead is minimized due to an efficient surface area to volume ratio.

Most Optimal Configuration: 384 cores, 4x96: Computation cost of 2952.50304.

This configuration demonstrates the lowest computation cost, indicating it balances computation and communication most effectively. The grid structure minimizes communication overhead by optimizing the surface area to volume ratio, making it the most resource-efficient and time-effective configuration for this

Determining Geometry

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

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead
128	1x128	913.1	1003	89.9
128	2x64	1031	1143	112
128	4x32	1038	1114	76
128	8x16	945.4	1170	224.6

Why were the above-mentioned geometries chosen as the highest performing?

As a rule of thumb with higher px the communication cost increases, however it can be seen from the table that when communication is turned off the Gflops increases as px is increased. These two effects cancel each other when px = 4 and py = 32, when the total number of cores is 128. So as x increases, final Gflops increase till 4x32 and then final Gflops start decreasing.

Message passing time = alpha + n/beta, where n is message length, alpha = message latency, beta = peak BW

For 1D geometry : $Tcomm = 4(\alpha + 8\beta^{-1}N/\sqrt{P})$ For 2D geometry: $T_{comm} = 2(\alpha + 8\beta^{-1}N)$

Observed Patterns: Highest Performance at Intermediate Dimensions: The geometry 4x32 shows the highest GFlops performance, closely followed by 2x64. **Increasing Communication Overhead with More Dimensions**: The 8x16 geometry, despite having high "Communication off GFlops", shows significantly higher MPI overhead compared to 4x32 and 2x64. **Single-Dimension Geometry (1x128)**: This geometry has the lowest GFlops and higher MPI overhead compared to 2D geometries.

Hypotheses:

1. **Balancing Computation and Communication**: Optimal geometries balance the cost of computation and communication. For 4x32, the balance seems optimal, leading to the highest performance. The increase in x and decrease in y dimensions initially reduces communication cost up to a point (4x32), beyond which the communication cost increases due to more frequent message passing.
2. **Impact of Communication Direction**: In a 1D geometry (1x128), the communication cost is high because messages need to travel longer distances serially. In 2D geometries, communication can be more evenly distributed, reducing the cost up to an optimal geometry (4x32).
3. **Message Passing Time**: The message passing time, represented as α+n/β, indicates that both latency (alpha) and bandwidth (beta) play crucial roles. In geometries with higher dimensions, the number of messages (n) and hence total communication overhead increases.

Strong and Weak Scaling (4)

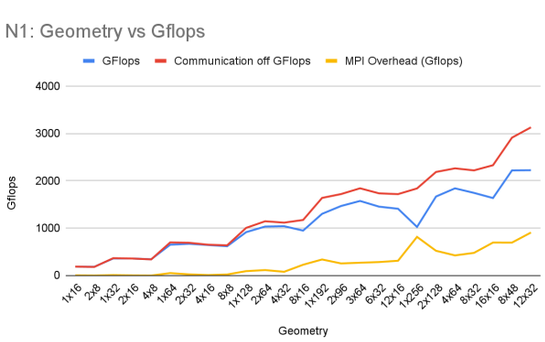
For N1:

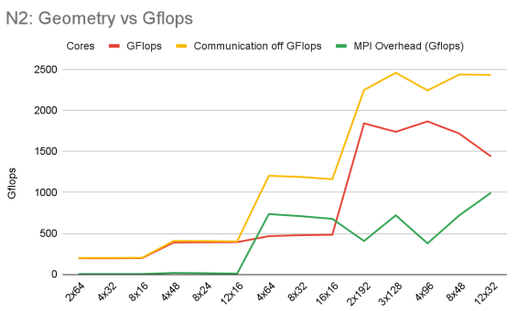
Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead
128	1x128	913.1	1003	89.9
128	2x64	1031	1143	112
128	4x32	1038	1114	76
128	8x16	945.4	1170	224.6
192	1x192	1300	1636	336
192	2x96	1465	1717	252
192	3x64	1572	1839	267
192	6x32	1451	1732	281
192	12x16	1407	1716	309
256	1x256	1023	1836	813
256	2x128	1665	2184	519

256	4x64	1838	2260	422
256	8x32	1586	2145	559

For N2:

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead (Gflops)
128	2x64	196.5	200.3	3.8
128	4x32	195.9	200.5	4.6
128	8x16	197.8	202.5	4.7
192	4x48	388.9	407.1	18.2
192	8x24	391.6	406.3	14.7
192	12x16	392.6	401.9	9.3
256	4x64	466.5	1202	735.5
256	8x32	478	1188	710
256	16x16	482.9	1160	677.1
384	2x192	1841	2248	407
384	3x128	1738	2458	720
384	4x96	1865	2243	378
384	8x48	1717	2436	719
384	12x32	1437	2432	995





Differences in the behavior of both strong scaling experiments for N1 and N2:-

N1 (larger problem size) scales better with more cores, but communication overhead becomes significant at very high core counts.

N2 (smaller problem size) suffers from higher communication overhead even at lower core counts, limiting effective scaling.

The key difference lies in the computation-to-communication ratio, with larger problems masking communication costs more effectively until network contention and synchronization costs become dominant.

For N1=1800: The highest GFlops is observed at 256 cores with a 4x64 geometry. Significant MPI overhead is observed as the number of cores increases. Intermediate geometries (like 4x32) tend to balance computation and communication costs well.

For N2=8000: Higher GFlops are observed at higher core counts. Geometries like 4x64 and 8x32 show high MPI overhead at 256 cores, indicating significant communication costs. Large geometries (like 2x192 and 3x128) start showing diminishing returns in GFlops due to high MPI overhead.

Hypotheses on Differences in Behavior

1. **Problem Size Impact on Scaling Efficiency**:

    - **N1 (1800)**: Smaller problem sizes do not benefit as much from strong scaling because the overhead of communication quickly outweighs the gains from parallel computation. This results in lower efficiency and higher MPI overhead as core counts increase.
    - **N2 (8000)**: Larger problem sizes benefit more from strong scaling because the computation-to-communication ratio is more favorable. As the problem size increases, the overhead of communication has less impact on overall performance, leading to higher efficiency and better GFlops performance at higher core counts.
2. **Optimal Geometry Dependence on Problem Size**: For smaller problem sizes (N1), more compact geometries (like 4x32) balance communication and computation costs effectively. For larger problem sizes (N2), larger geometries (like 8x32 and 4x64) can be utilized effectively despite higher communication costs because the increased computational load justifies the additional communication overhead.
3. **Communication Overhead Patterns**: Communication overhead increases significantly with core counts, especially for larger geometries. This indicates that for small problem sizes, the network latency and message passing costs dominate. While communication overhead still increases with core counts, the effect is mitigated by the larger computational workload, making the higher core counts more efficient.

Models to Explain Observations

1. **Communication Cost Model**:
    - Total Communication Cost: Tcomm=alpha + n/beta, where alpha is the message latency, beta is the bandwidth, and n is the message length.
    - For smaller problem sizes, n is smaller, making latency a larger component of the communication cost. For larger problem sizes, n is larger, making bandwidth the dominant factor, reducing the relative impact of latency.
2. **Computation vs. Communication Balance**: For small problems (N1), Time of communication increases faster than Time of computation leading to inefficiencies. For large problems (N2), Time of computation increases faster than Time of communication, maintaining efficiency even at high core counts.

Extra Optimizations

After Vectorization using flags: -ftree-vectorize -ftree-vectorizer-verbose=2 -march=native -DSSE_VEC

Results are:

For N1:

Compute N0: Ours	# iters = 8000	m x n = 8000 x 8000		
Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead (Gflops)
1	1x1	23.23	24.75	1.52
2	1x2	43.2	47.78	4.58
2	2x1	44.92	48.14	3.22
4	1x4	88.93	96.61	7.68
4	2x2	85.1	95.66	10.56
4	4x1	82.34	96.92	14.58
8	1x8	167.7	184.8	17.1
8	2x4	161.2	185.6	24.4
8	4x2	154	187.3	33.3
8	8x1	142.6	181.2	38.6
16	1x16	295.3	330.4	35.1
16	2x8	285.3	336.7	51.4
16	4x4	269.9	338.2	68.3
16	8x2	250.1	334.3	84.2
16	16x1	229.9	286.5	56.6

For N2:

Cores	Geometry	GFlops	Communication off GFlops	MPI Overhead (Gflops)
128	16 x 8	195.3	201.6	6.3
384	2 x 192	1809	3189	1380
384	3 x 128	1980	3348	1368
384	4 x 96	1966	3026	1060

The computation performance almost doubled for mesh size 800 for less number of cores but as the mesh size increased and number of cores was increased the effect of vectorization diminishes.

Potential Future work

1. Performing hand vectorization using SIMD intrinsics to help improve performance
2. Compiler Flags can be turned on for automatic vectorization (-mavx2, -03)

References (as needed)

1. https://sites.google.com/ucsd.edu/cse260-spring-2024/assignments/assignment-3-background?authuser=0
2. http://www.mpich.org/static/docs/latest/www3/
3. http://cseweb.ucsd.edu/~baden/Doc/mpi.html
4. https://web-backend.simula.no/sites/default/files/publications/Simula.acdc.8.pdf