CSE 260: HW 3 Report Template

Section (1) - Development Flow

Q1.a) Program Overview

The program is designed to simulate a 2D wave equation by using a 5-point stencil method. A 2D square field is simulated by calculating the values at discrete points evenly distributed. To scale the field to a large size, the field is divided into several blocks, where each of them is computed by a core, also known as rank. Since waves propagate across blocks, cores involved in the program communicate with each other through MPI.

At the beginning of the simulation, a configuration file is read and information of sine wave stimuli is included. The information consists of position, start time, and duration. These stimuli are stored in a list. For each iteration, the program checks whether each sine stimulus in the list has started (if not, then bypass it) or ended (then remove the stimulus from the list). Then, each rank updates the value in the field which it is responsible for if the stimulus is located within it. After handling the sine wave stimuli, the program starts to update the value with the 2D wave equation that does the propagation part. Before the calculation, we do a so-called “ghost cell exchange". As mentioned in the previous paragraph, each rank has to communicate with its neighboring rank to get the value of cells on the neighboring edge. These cells from the neighboring cores are called ghost cells.

The ghost cell exchange utilizes MPI_Isend() to send data and MPI_Irecv() to receive data. Then, MPI_Waitall() is called to make sure the exchange is completed. When preparing the data to be sent, it is much easier to handle the top and bottom edge since the machine is in row-major order. We just need to specify the address and the data length. However, for left and right borders, we have to copy the value to the sending buffer one by one. After the ghost cell exchange and new value calculation, the last step in an iteration is to rotate the pointers of prev, curr, and next to get prepared for the computation in the next iteration.

The scenario described above is the case of interior blocks. For blocks that contain global edges, they don’t have ghost cells on those specific edges. Instead, there are cells that absorb the waves to make it look like there are actually infinite bounds rather than walls. This is done by adding absorbing boundary cells outside of the global matrix. They have a 2D wave equation different from the other cells.

To balance the workload of each rank, the program is also designed to evenly distribute the work such that no processor has more than one row or column more work than any other processor. This is done by adding at most one extra row and column to each rank besides the fields that are evenly distributed.

Finally, to print out the statistics maxV and SumSq, which are the maximum value and sum of squared values respectively, MPI_Reduce() is used. The function finds the maximum value among the local maxVs from every rank to get the global maxV and stores them in the root rank, which is set to rank number zero. Similarly, the MPI_Reduce() function calculates the sum of local SumSqs to get the global SumSq and store them in rank 0. After the MPI_Barrier() function is called, the results are ready to be shown.

Q1.b) Development Process

1. Decomposite dimensions

   At the beginning, we tried to implement the 2d version at once. But it does not work and it is really hard to debug.

   Then we first finished the 1d version and implemented the 2d version based on the 1d version. The debug process is then much easier.

2. Synchronic

   During the implementation of ghost cells exchange, we first tried MPI_Send and MPI_Recv but they did not work. Then we used non-blocking calls MPI_Isend, MPI_Irecv and MPI_Waitall for the exchange. In this way, it also allows processors to utilize the resource more efficiently and avoids potential deadlocks.

3. Debug examples

   a. Segment fault: One of the big bugs is segment fault. Since there are lots of dynamic memory allocations in our implementation, checking pointer validity before freeing it is one thing that we tend to miss.
   b. Using plotter: At the beginning, we didn't know that there are methods in buffer.cpp to be done and found that the SumSq is scaled with np increasing. Later I implemented the plotter and found the initial conditions are multiplied due to the incorrect mapping.
   c. Order of ghost cells exchange: We first put the ghost cells exchange for u.nxt after all calculations and before buffer swap. This ignored the modification

made by the initial condition on u.cur. So we put the exchange on u.cur after initial condition updates and before calculations.

Q1.c) Optimization Process

1. Synchronization
    a. In the ghost cells exchange process, we used two barrieres instead of one. It seems the first one is the only necessary one. Once one process finishes the communication (both sending and receiving), it can calculate the result with the updated ghost cells and update the interior cells without influencing the other process. The possible reason for this optimization is to avoid unnecessary waiting. Imagine if one processor is much faster than the other processors, it should always block at the first barrier while other processors are still communicating.

Section (2) - Scaling Results

Q2.a) Computation Performance vs Workload

The program measures the number of points processed per second. Calculate the effective GFlops/sec (computational work) as a function of points processed/sec. Show how you derived this number.

For each point, we need 9 floating operations to update it according to the ODE and PDE function. Therefore, we have
GFlops/sec = GFlops/point * point/sec = (9*10^(-9)) * point/sec = (9*10^(-3)) * Mpoint/sec

Q2.b) Single-core Performance

n=500

Running time of original provided code (i = 2000): 13.788 sec

Running time of our MPI code (i = 10000):

Core geometry	Running time (sec)	Speedup
1x1	68.437	1.000
1x2	35.756	1.914
2x1	35.776	1.913
2x2	18.441	3.711

1x4	18.631	3.673
4x1	18.138	3.773
2x4	9.935	6.888
4x2	9.160	7.471
1x8	10.479	6.531
8x1	9.086	7.532
4x4	4.611	14.842
1x16	4.803	14.249
16x1	4.631	14.778
2x8	4.598	14.884
8x2	4.747	14.417

Q2.c) Strong Scaling Study on small number of cores

n=1000; i = 3000

Core geometry	Running time (sec)	Speedup	Efficiency
1x1	82.415	1.000	1.000
1x2	41.871	1.968	0.984
2x1	42.401	1.944	0.972
2x2	21.665	3.804	0.951
1x4	22.257	3.703	0.926
4x1	22.543	3.656	0.914
2x4	11.084	7.435	0.929
4x2	11.585	7.114	0.889
1x8	11.978	6.881	0.860
8x1	10.451	7.886	0.986

4x4	5.251	15.695	0.981
1x16	5.620	14.665	0.917
16x1	5.452	15.116	0.945
2x8	5.332	15.457	0.966
8x2	5.270	15.639	0.977

Using -k option:

Core geometry	Running time (sec)	Speedup	Efficiency
2x4	10.681	7.716	0.965
4x4	5.442	15.144	0.947

The MPI overhead is 0.403 seconds (3.64%) for p=8. For the 4x4 core geometry, the performance even degrades a little bit after turning off the communication. It probably results from the fluctuating performance. To sum up, the difference is small and we believe the reason is the use of non-blocking communication, so the latency is hidden. Also, these cores are located within the same node. The communication cost is lower.

Q2.d) Strong Scaling Study on medium number of cores

n=2000; i = 5000

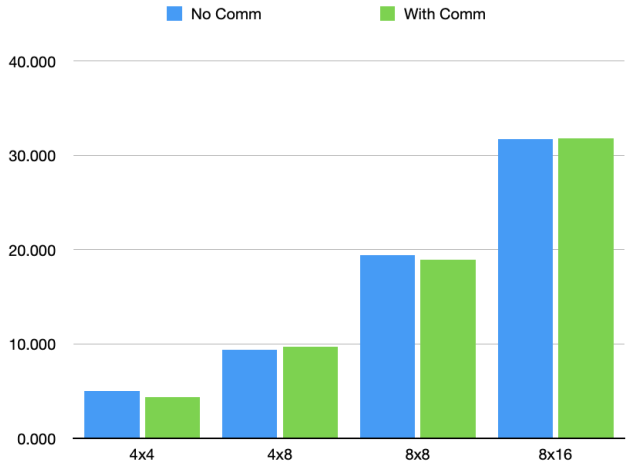
Cores	Geometry	MPoints Per Second.	GFlops/sec
16	4x4	480.009	4.320
32	4x8	1079.564	9.716
64	8x8	2105.042	18.945
128	8x16	3537.944	31.841

Using -k option:

Cores	Geometry	MPoints Per Second.	GFlops/sec
16	4x4	554.816	4.993
32	4x8	1042.807	9.385

64	8x8	2160.994	19.449
128	8x16	3527.959	31.752

We find little overhead when the number of cores is not in a square geometry. While when the geometry is in square, the overhead remains nearly constant at about 55~75 Mpoints/sec, which is the cost of communication.



Q2.e) Strong Scaling Study on large number of cores

n=4000; i=2000

Cores	Geometry	MPoints Per Second	GFlops
128	8x16	3584.229	32.258
192	12x16	5633.142	50.698
256	16x16	6832.740	61.495
384	16x24	9141.116	82.270

Using -k option:

Cores	Geometry	MPoints Per Second	GFlops
128	8x16	3570.632	32.136
192	12x16	5618.635	50.568

256	16x16	6760.563	60.845
384	16x24	9854.239	88.688

A huge difference is only seen when the number of cores is 384. We expected the communication overhead would be largely increased as the number of cores is increased because of the need for communication between nodes. However, it didn't happen when n=128, 192, and 256.

Q2.f) Communication Overhead Differences

The communication overhead is all pretty small when using fewer cores. The only huge difference occurs when p=384. It is reasonable that there are more nodes involved so more communication overhead is observed.

Section (3) - Determining Geometry

Q3.a) Extensive study on small p

n=1000; i=5000; p=32

Core geometry	Running time (sec)
1x32	5.021
4x8	4.863
8x4	4.688
32x1	5.012

n=1000; i=5000; p=64

Core geometry	Running time (sec)
1x64	2.782
4x16	2.509
8x8	2.472
16x4	2.556
64x1	2.736

From both tables, the running time is like a valley shape, which means balancing the number of cores on the two axes is better. This is intuitive since the symmetry nature is more favorable, and more latency hiding opportunities are provided. We also expected that more cores along the y axis would yield better results due to the nature of row-major order machines. However, it seems that whether the row or column should have more cores is still an open option.

Q3.b) More efficient study on medium p

n=2000; i=5000; p=128

Core geometry	Running time (sec)
8x16	5.621
16x8	5.631
4x32	5.806

From our observation in Q3.a, we omit the geometry of having only one or two cores on one of the dimensions. Among the other geometries, these three have the best performance. It follows the observation from the Q3a that balancing the number of cores along the two dimensions yields better results.

Q3.c) Hypothesis and more efficient study on large p

n=4000; i=2000; p=192

Core geometry	Running time (sec)
12x16	8.521
16x12	8.639
8x24	8.640
24x8	8.642

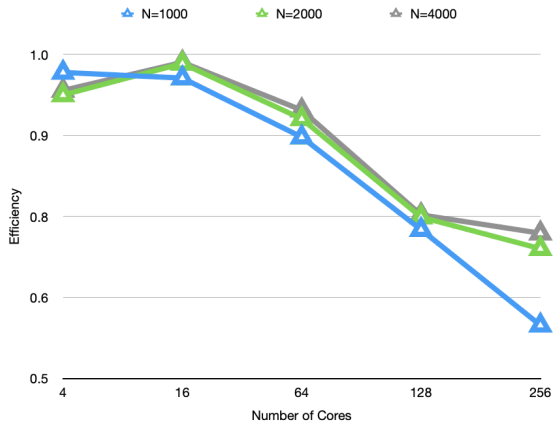
The results are consistent with the previous ones. We need to try our best to balance the number of cores along the two dimensions. Also, we found out that in most cases, when there are less rows and more columns of cores, the results are slightly better. We believe this is a result of the balance between cost of computation and cost of communication. When the number of cores are imbalanced, there will be more cost on communication. On the other hand, the ghost cell exchange would love to do more exchanges on top and

bottom edge because they don't need to pack the data. However, from our observation, it seems that it's the opposite since less cores on the y-axis get better results. We guess the reason could be the cache line alignment. Packing helps data to be aligned with the cache line.

Section (4) - Strong and Weak Scaling

Q4.a) Scaling Behavior

The chart below shows the efficiency of different numbers of cores and problem sizes. The weak scaling phenomenon can be observed. When the problem size grows with the number of cores, we get an efficiency closer to one. The efficiency we could get when N=1000 and P=256 is even lower than 0.6, therefore we could say that this is more like an example of weak scaling than strong scaling.



Section (5) - Error Analysis

Q5.a) Potential Errors

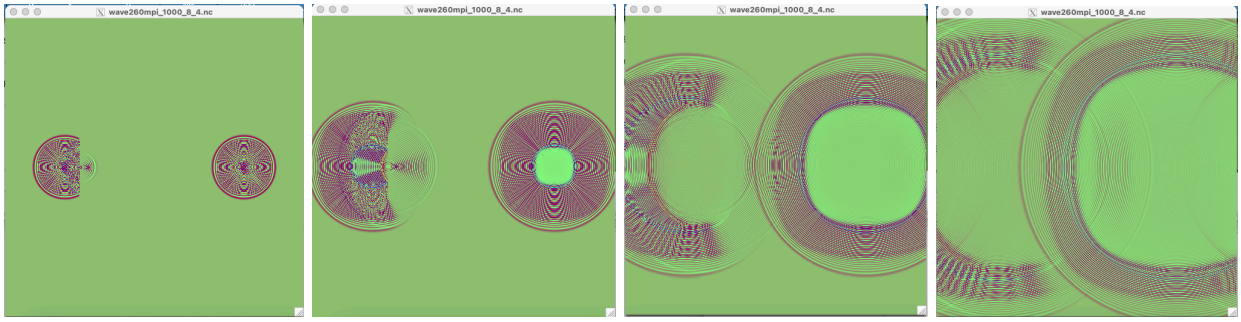
1. Out of Memory
    a. Our solution is not a space optimal solution. So if the given space is limited, our program may raise Out of Memory error.
2. Allocation
    a. We used memalign to ensure data addresses compatible with MPI. There may be errors about the allocations, even though we used assert statements to catch them.
3. 5. Numerical Precision and Floating-Point Issues:
    a. Rounding and Stability:
    PDE computations are sensitive to numerical stability. Floating-point

round-off errors can accumulate if the discretization parameters (e.g., c, kappa) and timestep t or grid spacing h are chosen poorly. Although not a coding bug per se, insufficient spatial/temporal resolution or inappropriate parameter choices can produce solutions that appear incorrect or "off" from expected behavior.

b. Initial Conditions and Stimulus Placement:
   If initial conditions or stimulus inputs do not consider ghost regions or are incorrectly applied only on part of the domain, the discrepancy in wave amplitude or speed at boundaries could be misdiagnosed as a parallelization error.

Section (6) - Extra Credit

EC-a) (SEE INSTRUCTIONS)



Currently, our plotter is making rank 0 collect the global plot then plot it. This is a problem worth solving since it takes extra time to communicate the global plot and does not balance the work of plotting between all nodes. The best solution is to make a node to plot its' local plot.

Section (7) - Potential Future work

Higher-dimensional extensions; Incorporating more sophisticated or problem-specific boundary conditions—such as perfectly matched layers (PMLs) for wave equations; Dynamic load balancing, which would ensure that each MPI process receives a fair share of computational work, eg. monitoring run-time performance metrics.

Section (8) - References

Course materials including lecture, discussion, and assignment instructions; Expanse website