CSE 260 PA3 Report

Fan Jin Yucheng Huang

f1jin@ucsd.edu yuh023@ucsd.edu

Git Repo link: https://github.com/cse260-wi20/pa3-pa3-yuh023-f1jin

University of California San Diego

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.]

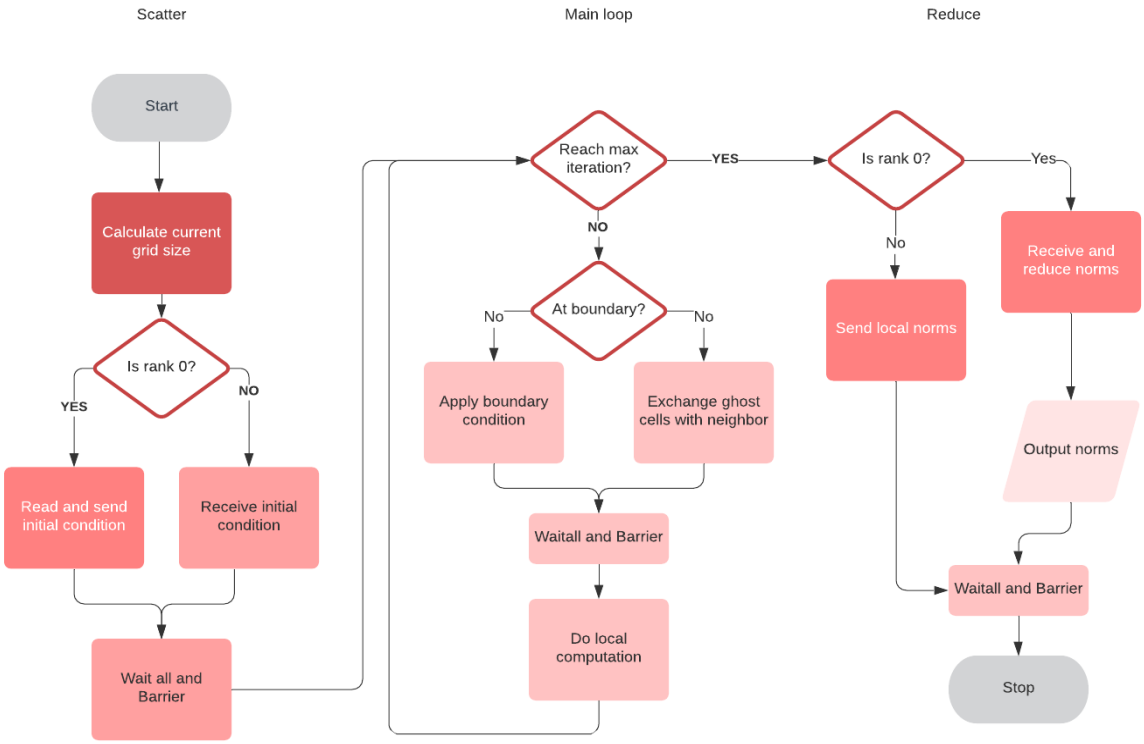
Notation (row, column)	Meaning
M, N	Global matrix size
m, n	Local matrix size, excluding ghost cells
r_x, r_y	Rank index
x, y	Processor geometry

For each process with rank r_x and r_y, local matrix size is calculated as

$$m = \text{floor}(M/x) + \text{condition}(M\%x < r_x),$$

$$n = \text{floor}(N/y) + \text{condition}(N\%y < r_y).$$

How MPI calls work: During the communication phase, each node sends and receives ghost cells asynchronously, using MPI_Isend and MPI_Irecv. A wait statement, MPI_Waitall, is added in the end, followed by a Wait and Barrier. This synchronizes nodes and make sure they never exchange ghost cells across iterations.



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

1. We added MPI code to compute in parallel. At first, we tried to use MPI_Send() and MPI_Recv(), which might block and result in sequential message passing in a particular order. We then chose to use asynchronous calls (i.e. MPI_Isend() and MPI_Irecv()) to avoid potential deadlock or particular message passing order.
2. Following the instructions, we prepared initial condition matrix in rank 0 and distributed these initialization conditions to all the ranks using MPI calls. To make sure our implementation is correct, we tried to compare the L2 norm of different processor geometry and different input size n.
3. 2-level blocking was implemented. This was intended to reduce cache misses and improve q (in the roofline model). Within the inner blocking, a 256-byte alignment memory was allocated to do data copy.
4. We conducted scaling study on Bang for 1-8 cores, and then we tested scaling on COMET for nprocs = 96 and N=1800, achieving 95% scale or even super linear scale. (See Q2.b and Q2.c) Scale drops under 95% when N=8000.
5. We compared the performance of fused code and unfused code. The results showed that fused code exhibits much better performance.

6. We wrote our own gather() function to gather results (Linf/L2 norms) at first, which was slow especially for large n and large #cores. Instead, we turned to MPI_Reduce(), which turns out to have more than 1000GF/s, much faster.

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.

We did pair programming. Both of us know how our code works, and a lot of bugs were spotted once they came out.
Improvements came from both computation and communication.

1. We made computation faster by copying and padding when computing block by block. After copying, adjacent rows become closer in memory address, making it faster to access a column in a row-major ordered matrix.
2. We employ minimal communication. Among the three matices *E*, *E*_prev and *R*, everything is updated in place except *E*, which uses neighboring *E*_prev. So, only communication of *E*_prev is necessary in ghost cells. We use asynchronous send/receive calls as well.

We made several mistakes and setbacks.

3. We failed to notice that matrix R is updated on its own, so we don't need to send ghost cells to its neighbors.
4. We wrote our own send/recv code within a for-loop to gather norms at the end, which turned out much slower than using MPI_Reduce directly.
5. We confused MPI_Waitall with MPI_Barrier and therefore failed to synchronize.

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 on bang (1 core and 8 cores).

Nprocs.	Geo	GF	GF w/o MPI	%MPI	Note
1	1x1	0.4587	0.4589	0%	Starter Code
1	1x1	1.418	N.A.	N.A.	w/ L1 Blocking
1	1x1	0.4590	N.A.	N.A.	w/o L1 Blocking
8	8x1	10.490	N.A.	N.A.	L1 Blocking

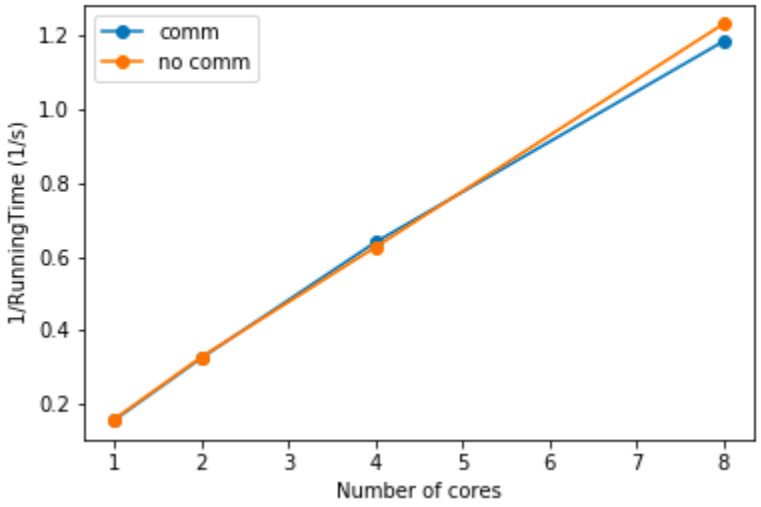
MPI overhead is negligible.

We conducted experiments above using input size n=400 and #iterations=500. For the parallel MPI code, we used FUSED code in the innermost for loop.

Here, our parallel MPI code with L1 Blocking exhibits a much higher GFlops value than the original code. This may due to the fact that 1-level blocking can reduce cache misses. We also copied data into a small block of 256-bit aligned memory. This makes it further improved.

The MPI overhead for L1 blocked code is almost 0%.

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 8 cores, while keeping N0 fixed.



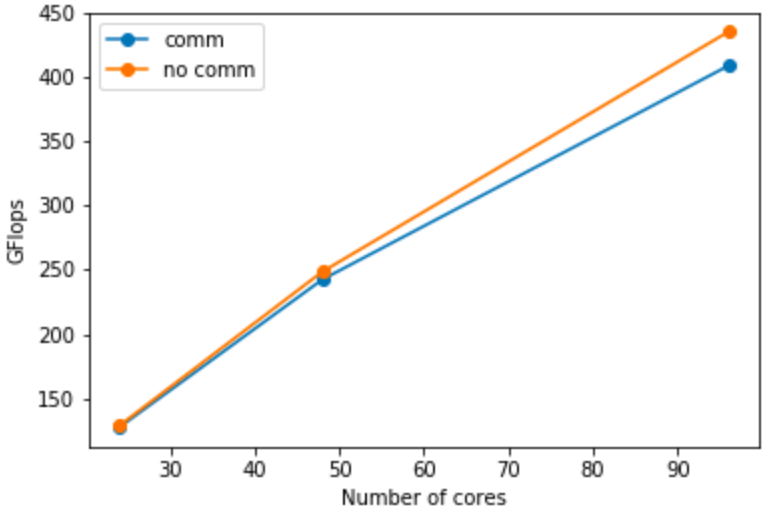
We set n=400 and #iterations=2000 and use fused code. Experiments above shows results for code with L1 blocking. Block size is 50. Clearly, we observed a strong scaling.

Q2.c) Measure communication overhead on 8 cores.

nprocs	GF	GF w/o comm	%comm
1	1.434	1.431	0%
2	2.921	2.927	0%
4	5.666	5.738	1.2%
8	10.580	11.090	4.6%

Communication overhead remains less than 5% on 8 cores.

Q2.d) Conduct a strong scaling study on 24 to 96 cores on Comet. Measure and report MPI communication overhead on Comet.



nprocs	Geo	GF	GF w/o comm	%comm
24	12x2	127.3	128.6	1%
	6x4	124.4	125.7	1%
	4x6	124.2	125.6	1%
	2x12	122.3	124.3	2%
	8x3	117.3	117.0	0%
48	12x4	242.5	248.6	2%
	4x12	236.7	246.1	4%
	8x6	224.6	230.8	3%
	6x8	221.4	227.9	3%
96	8x12	409.2	435.7	6%
	12x8	403.0	425.1	5%
	48x2	382.1	384.4	1%

The results shown above indicate that processor geometries that have similar size on two dimensions (i.e. x and y are close) performs best.

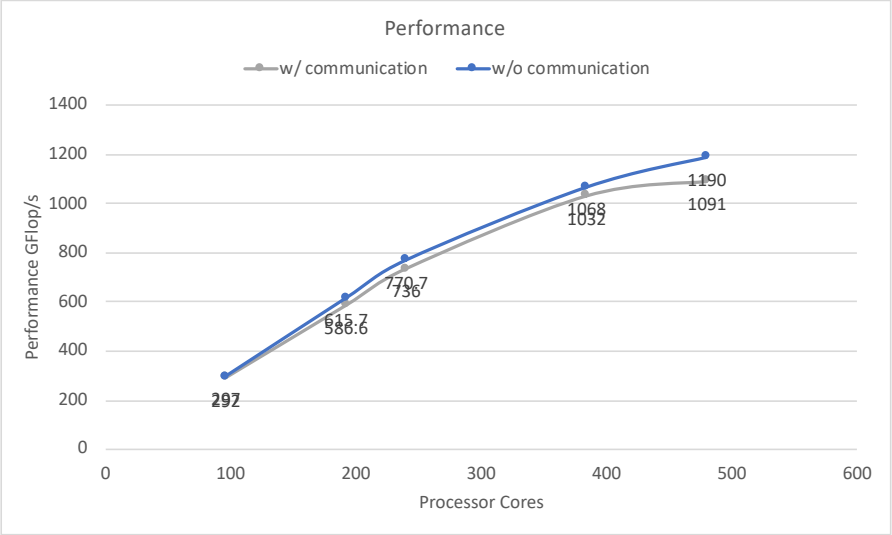
From 24 cores to 96 cores, the communication overhead is noticeable, because the ratio of time spent for communication, which decreases quadratically, and time spent for computation on each processor, which decreases linearly, increases. Moreover, the synchronization cost increases as the number of cores increases.

Difference between Comet and Bang: Here, we got 5% communication overhead on Comet for 96 cores. We got 5% overhead on Bang for only 8 cores. We learned that communication overhead is highly platform dependent.

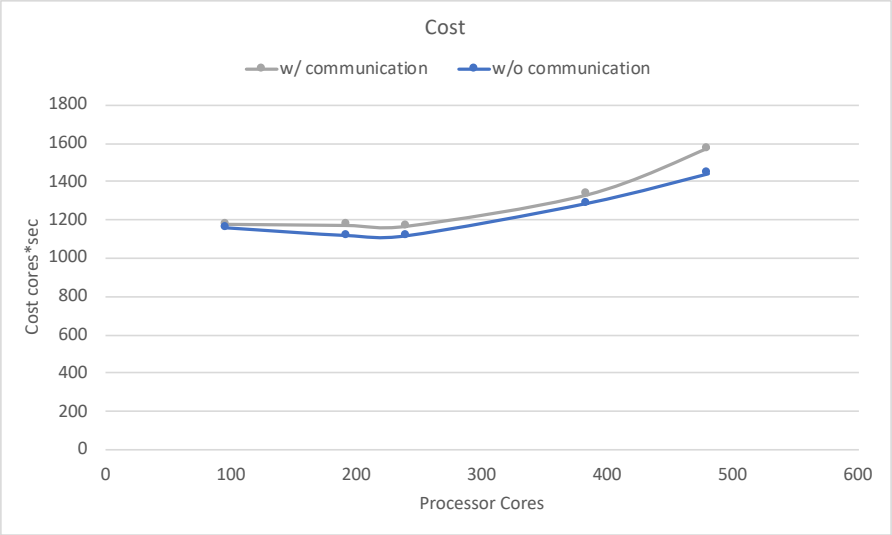
Q2.e) Report the performance up to 480 cores. (i.e. 96, 192, 240, 384 and 480 cores). (Use the knowledge from the geometry experiments in Section (3) to perform the large core count performance study.)

From Section 3, we infer optimal geometry should be x smaller than y and x close to y.

Cores	x	y	Performance (GF/s)	Relative Scale
96	8	12	292	1
192	12	16	586.6	1.004*
240	15	16	736	1.008*
384	16	24	1032	0.876
480	20	24	1091	0.846



Q2.f) Report cost of computation for each of 96, 192, 240, 384 and 480 cores. In addition to it, run the 'batch-test.sh' script file to report the Queue Time (i.e. Wait Time for the batch job to allocate requested number of cores) in the following table.



Cores	Cost of Computation (cores * sec)	Queue Time (sec) in 3 tries
96	1178.88	23.987, 304.126, 36.164
192	1172.928	102.399, 5.449, 53.405
240	1168.56	4.358, 34.941, 9.805
384	1332.864	1.090, 116.613, 20.704
480	1576.32	26.161, 46.887, 4.358

Conclusion: Queueing time is highly variable and depends on server traffic. It could take seconds or minutes to wait, randomly.

Cost of computation is, however, a different story. We pay roughly the same cost when number of cores is low, and we have to pay extra cost when number of cores is high, which implies lower efficiency. This can be explained by the fact that computation cost remains constant while communication cost goes linear with number of processors for some fixed matrix size. We pay cost for extra communication and therefore it is less efficient.

Section (3) - Determining Geometry

Q3.a) For p=96, report the top-performing geometries. Report all top-performing geometries (within 10% of the top).

x	y	N	GFlop/s
1	96	1800	185.9
2	48	1800	362.6
3	32	1800	279.4
4	24	1800	343.2
6	16	1800	352.0
8	**12**	1800	**411.7***
12	**8**	1800	**404.6***
16	**6**	1800	**375.2***
24	4	1800	362.5
32	3	1800	315.8
48	2	1800	358.7
96	1	1800	230.4

Conclusion: We noticed performance is best when x is close to y, especially when x is slightly smaller than y.

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

To simply our model, assume the matrix size N is a multiple of x and of y, where $x * y = P$.

Cost of computation: Under the assumption above, each process gets N^2/P expressions to compute for each element. It has 3 expressions to compute for each position.

Cost of communication: Communication happens between two adjacent processes, which is proportional to $2N(x-1+y-1)$ for each element. It requires 4 sends and receives each time.

Model: Given N and P, computation overhead is fixed in theory. Given $x * y = P$, communication overhead is determined by $x + y$, which is minimized when $x = y = \sqrt{P}$. This model explains our findings that peak performance is reached when x is close to y.

Section (4) Extra Credit

Q4.a) The current plotting routine only works on a single thread. Devise an efficient method of plotting results on a multi-core (MPI) application. Demonstrate your code by showing snapshot updates (at least 3) of an evolving simulation. In your implementation, does the node that does the plotting also do computation? If so, is this a problem worth solving?

Q4.b) Either hardcode a SIMD implementation or using the intel compiler and various code restructuring and perhaps some pragmas, get the code to vectorize using AVX2 and report the performance increase. Describe what you need to do (flags, code changes, etc) to get the code to vectorize and show that it actually did vectorize. (Provide us with an easy way to turn these optimizations on or off - default should be off). Show the speedup on scalar code and on 48, 96 and 480 cores. Does the code scale as well as it did before aggressive vectorization? Explain why it does or does not. Referring back to the time * cores used product, does the result from Q2.f change?

Q4.c) Implement an MPI cluster at Amazon. Run up to 8 instances and measure the communication overhead (compare with Comet's communication overhead between nodes). Describe the process of creating the cluster.

Section(5) - Potential Future work

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

1. Try to assign non-square data to each processor and explore the best non-square configuration.
2. Try to use tree-like geometry to distribute initial conditions from rank 0.
3. Try to use one-side communication instead of two-side communication.
4. To reduce communication cost, neighboring processors may exchange MPI messages every two iterations. To achieve this, we need a thicker layer of ghosts cells (i.e. A nxn matrix will need (n+4)-by-(n+4) memory to hold both data and ghost cells).
5. Use AVX instructions to increase ILP for each processor.
6. Optimize (parameter tuning) 2-level block sizes parameters.
7. Devise an efficient method of plotting results on a multi-core (MPI) application.
8. Use AWS to create an MPI cluster and build HPC cluster in the cloud.

Section (6) - References (as needed)

- OpenMPI 1.6 Documentation. https://www.open-mpi.org/doc/v1.6/