CSE 260 PA #3

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.]

a. First, allocate ranks to each processor. Then process 0 distributes the corresponding submatrix of initial condition to each core, achieved through a broadcast of the entire E and R matrices from the root process. To divide the mesh as evenly as possible, we distribute the remainder to the same number of processors. In this way, the amount of work assigned to every process along a given dimension will differ by not more than one row.

b. In each process, have the ghost cell ready to send and receive using mpi non-blocking call. Every iteration, each process needs to communicate with its neighbor process to exchange the ghost-cell values. Every process except those located at the edges of the global matrix will communicate with four neighboring processes, i.e. top, bottom, left and right. For the top and bottom ghost cells, we directly copy the data in the same array, for left and right cells, we use mpi type to perform copy-free sending and receiving.

c. Calculate the inner part of the submatrix by unfused kernel that does not need the information of the ghost cell while mpi is sending the receiving ghost cells. The unfused kernel is to compute the PDE and ODE in a separate loop. During the computation, we use cache blocking strategy to slide a tile over non-overlapping regions of the chunk in row-major-order. After the inner part calculation finishes, wait until all ghost cells are properly received. Then do the calculation of the outer bound of the submatrix.

d. Process 0 collects the statistics of inf norm and L2 norm by employing mpi reduce.

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

a. Initial implementation without mpi call

b. Synchronous communication and computation, inner cell computation waits for ghost cell communication.
c. Interleaving inner block computation and transmission, perform PDE and ODE computation while transmitting the data. Afterwards, calculate the outermost row and column after the transmission process completes.
d. Copy-free transmission. Instead of copying boundary cells into a temporary buffer and then send it to other nodes. Use the vector type of MPI to perform copy-free sending and receiving.
e. Further optimization includes cache blocking slide a tile over non-overlapping regions of the chunk in row-major-order, performing the entire computation in that tile. As well as hint the compiler to use registers, by creating temporary variables to hold the value of e_tmp, r_tmp and add prefetch and unroll matrices needed for the calculation.
f. vectorization to accelerate the calculation and further boost the performance (see Q4.b)

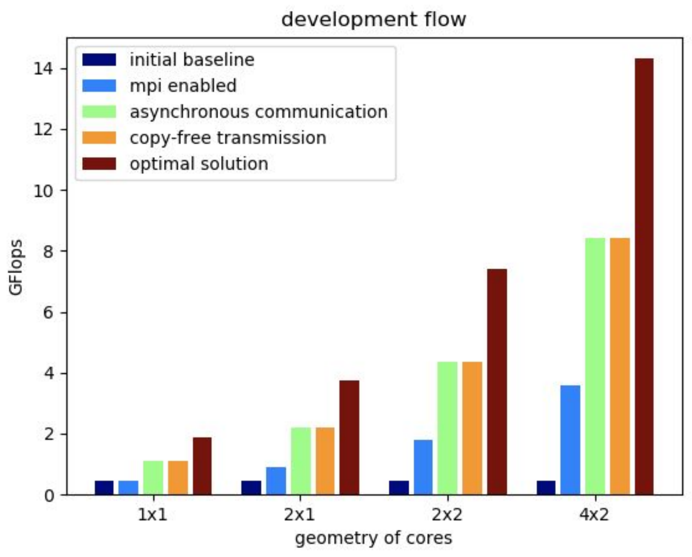
Q1.c) If you improved the performance of your code in a sequence of steps, document the process. It may help clarify your explanation if you summarize the information in a list or a table. Discuss how the development evolved into your final design.

We test the performance increasing on bang, under the condition that n = 400, i = 2000.

The progressing results are displayed as follows.

version	np	x	y	gflops
a.	1	1	1	0.456
	2	2	1	0.4538
	4	2	2	0.4552
	8	2	4	0.450

b.	1	1	1	0.4557
	2	2	1	0.9098
	4	2	2	1.812
	8	2	4	3.585
c.	1	1	1	1.109
	2	2	1	2.194
	4	2	2	4.335
	8	2	4	8.412
d.	1	1	1	1.112
	2	2	1	2.198
	4	2	2	4.338
	8	2	4	8.436
e.	1	1	1	1.863
	2	2	1	3.728
	4	2	2	7.388
	8	2	4	14.3



The figure above shows the five incremental baselines for our development progress.

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).

	Time execution for the whole program	T_p	T_p -k (disable communication)	Gflop/s	MPI_Overhead
Original provided code	19.762	19.65	19.66	0.4565	N/A
Mpi 1 core	4.827	4.708	4.76	1.903	Negligible
MPI 8 cores	1.747	0.6259	0.5964	14.32	0.0295

It can be seen from the above sheet form, the overhead brought by the mpi is small enough to be negligible.

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.

np	Running time
1	4.716
2	2.357
4	1.209
8	0.6236

It can be observed above, as the number of processors double , we could also achieve double performance, so our code has strong scaling performance.

Q2.c) Measure communication overhead on 8 cores.

We report the output from our codes as follows.

When communication is enable:

```
  M x N   px x py Comm?   #iter  T_p, Gflops       Linf, L2
@ 400 400   2 4     Y    2000  0.6249 14.34   9.92447e-01 7.16013e-01
```

When communication is disable:

```
M x N   px x py Comm?   #iter  T_p, Gflops       Linf, L2
@ 400 400   2 4     N    2000  0.5974 15   9.92446e-01 3.10838e-01
```

The communication overhead for 8 cores is 0.6249-0.5974, which is 0.0275.

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

np	MPI communication overhead
24	0.119
48	0.0967
96	0.072

It can be seen from above, as the number of processors increase, the communication overhead is increasing.

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.)

np	time	gflops
96	1.599	560.4
128	1.264	709
240	0.8178	1096
384	0.6577	1362
480	0.5593	1602

It can be seen from above, our mpi implementation could achieve more than a Teraflop/sec by using 480 cores. At the same time, we also achieve the strong scaling performance. This demonstrates the superior performance we achieve.

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 the requested number of cores) in the following table.

Cores	Queue Time
96	2.197
192	32.789
240	176.961
384	116.637
480	116.294

Requesting more cores would result in a higher waiting time. The queue time is also heavily dependent on the congestion on the server.

Section (3) - Determining Geometry

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

Geometries	Performance (gflops)
1 x 96	509
2 x 48	506.3
4 x 24	491.5
6 x 16	447.7
8 x 12	420.1

The top-performing geometries are 1 x 96 and 2 x 48.

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 order to achieve optimal geometry, we would need to consider communication overhead and computation overhead. The less processors, the less communication cost. In the meanwhile, we should maximize cache utilization in the computation. From over observation, the geometries with x lower and y higher always yield better performance. When x has lower value, meaning each row would store a bigger number of elements along the process matrix, the program would utilize more cache locality since these elements are stored in contiguous row-majored memory locations.

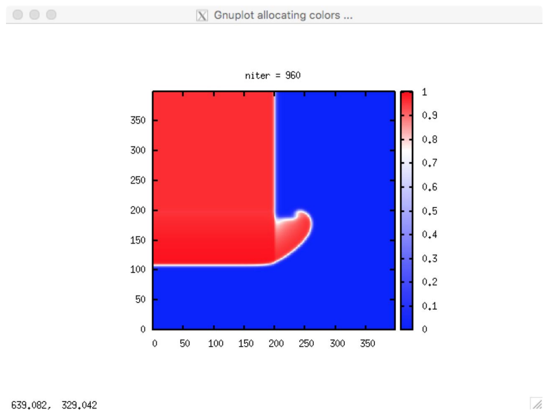
But this also has a limit. When increasing the elements in a row goes beyond some threshold that is not a multiple of cache line size, this would negatively impact the performance.

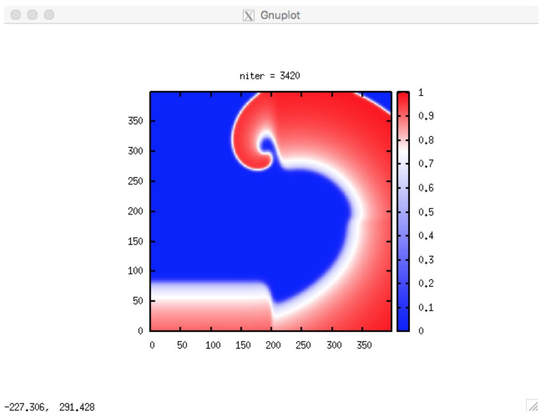
Section (4) Extra Credit

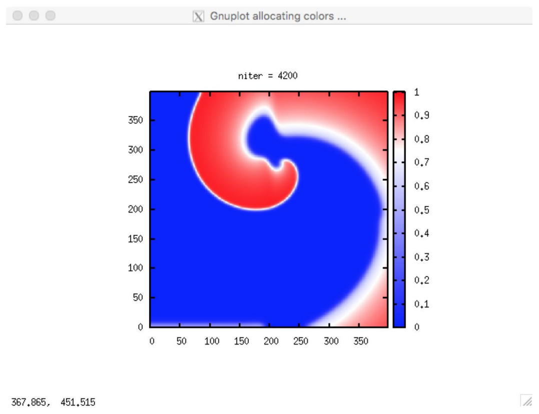
Q4.a) The current plotting routine only works on a single thread. Devise an efficient method of plotting results on a multi-core (MPI) application. Demonstrate your code by showing snapshot updates (at least 3) of an evolving simulation. In your implementation, does the node that does the plotting also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

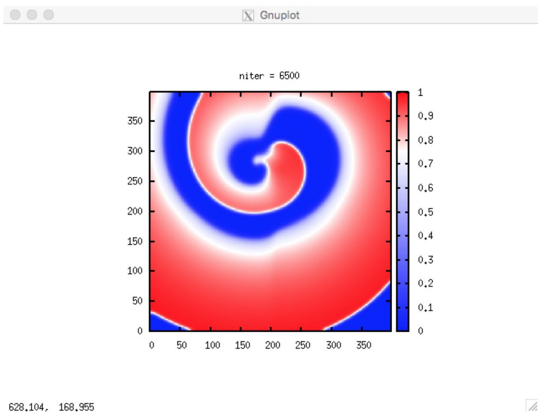
In our implementation, we use the root processor 0 to do the plotting. After every iteration, processor 0 will collect sub-matrix E from all the cores using MPI_Recv operations and push the whole complete matrix into the 'Plotter' object. Other processors simply send its own data matrix to processor 0. Since processor 0 also does computation, and at the end of each iteration, we use mpi_barrier to synchronize all the processors, so that would make the whole process longer than usual. So we don't think it's worthwhile to do multi-core plotting. The plotting function can be enabled by setting the plot_freq flag which sets the plotting frequency.

The snapshot of our evolving simulation can be seen below. We test on N=400 and simulate for 8000 iterations. We pick 4 typical snapshots for illustration.

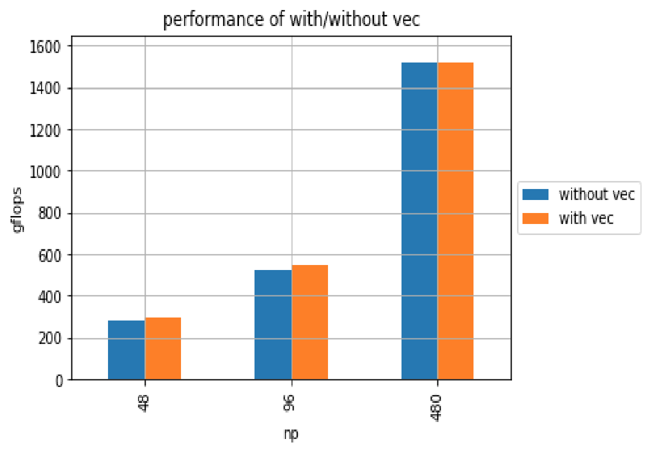


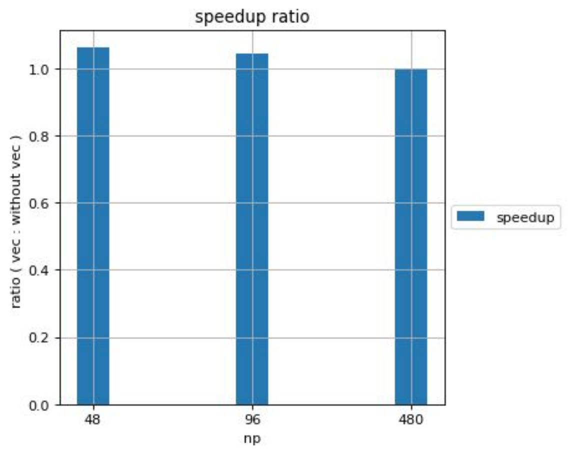






Q4.b) Get the code to vectorize using AVX2 and report the performance increase. Describe what you need to do (flags, code changes, etc) to get the code to vectorize and show that it actually did vectorize. Show the speedup on scalar code and on 48, 96 and 480 cores. Does the code scale as well as it 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?





The result shows that the vectorization of the code does not have a better scaling than without vectorization does. This is because auto vectorization by intel compiler performs much better than self vectorization. Meanwhile, by either the auto vectorization or using AVX2, we achieve greater than 1.5TF at 480 cores.

We implement the AVX2 by using the optimization #pragma ivdep, which is an Intel compiler directive which forces the compiler to vectorize any loop by ignoring vector dependencies. The AVX vectorization is enabled when setting the vec=1 flag .

Section(5) - Potential Future work

In future, we would like to try to optimize distributing matrix from process 0 more efficiently instead of having multiple buffers. In addition, optimizing loops by using more simd methods is also what we expect.

Section (6) - References (as needed)

https://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf
https://computing.llnl.gov/tutorials/mpi/
https://software.intel.com/en-us/articles/vectorization-essential
http://cseweb.ucsd.edu/classes/fa15/cse260-a/static/Bang/mpi.html