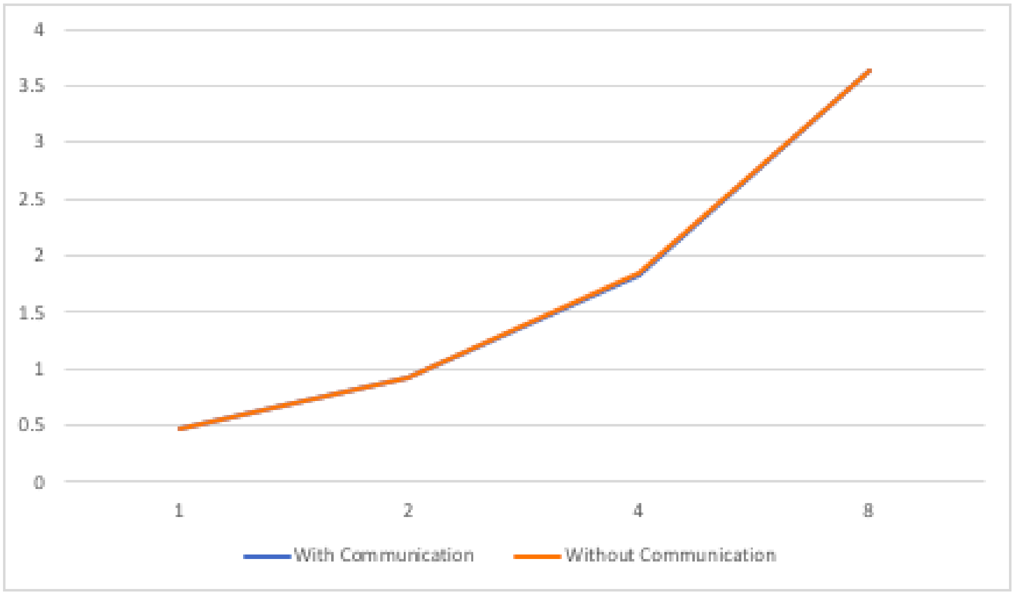
Assignment #3 - Optimizing the Aliev-Panfilov Model using MPI

Unnikrishnan Sivaprasad A53230407

Debosmit Majumder A53242244

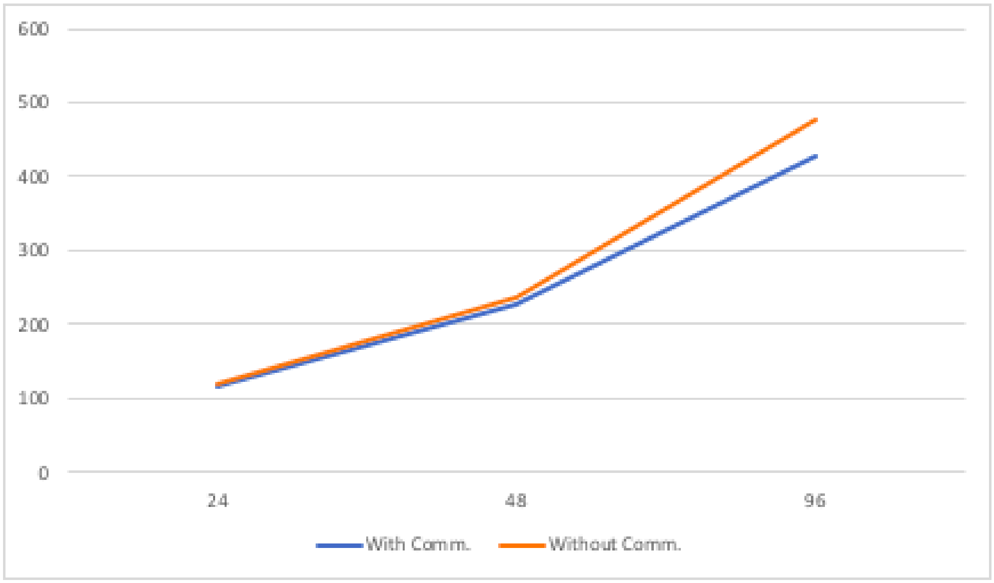
1. **Performance**
   a. **Scaling on bang(1 to 8)**

a) With our implementation mentioned above, we get strong scaling on Bang for 1 to 8 cores and keeping N as fixed. We use 1-dimensional geometry. There is very little communication overhead on bang as can be seen in the figure below. This may be because since the number of cores is small, the computation time dominated the communication time. Thus, the processors have to wait less for MPI Send and Receives. Also, it may be because on Bang, all the 8 processors are near to each other and thus sending and receiving messages through MPI does not take a lot of time. We run the strong scaling for N = 400 and iterations = 2000. X-axis plots the number of processors and Y-axis plots the performance in GFlops.



b. **Scaling on comet(24 to 96)**

d) We also scale on Comet using 2-D processor geometry. In this case, we determine the optimal geometry for each number of cores. But first, we study the scaling on Comet and find that it has stronger scaling as shown in the figure below, where X-Axis gives the number of processors and Y-Axis gives the performance in GFlops. We use 1800 as the matrix sizes and number of iterations = 2000.



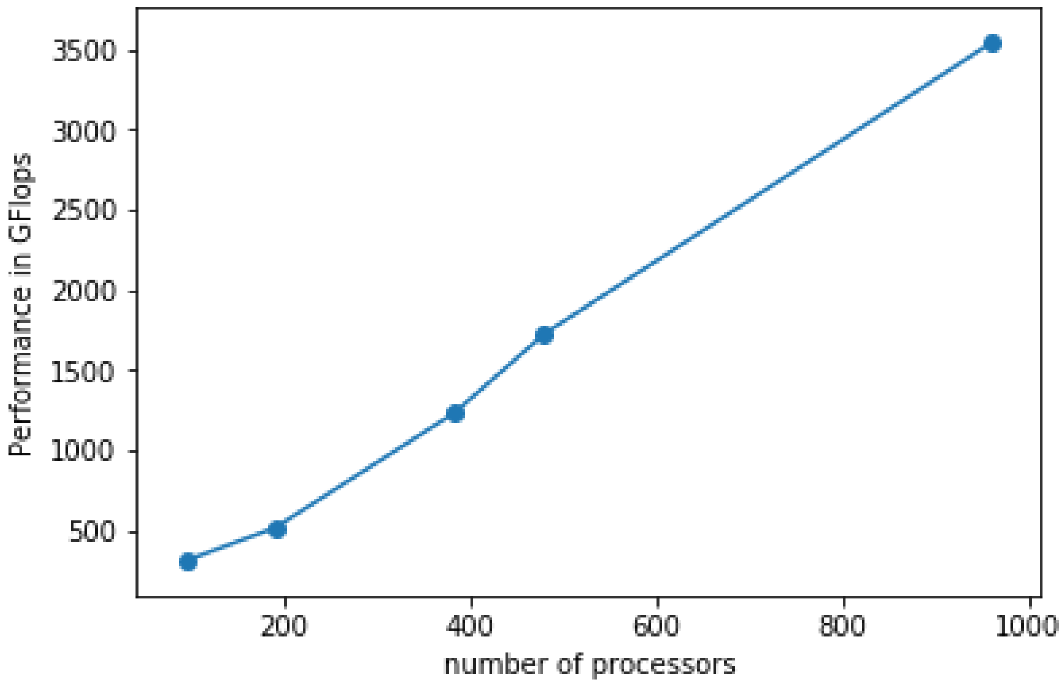
We discuss what should be the optimal geometries for each of the number of cores. It was observed that the optimal geometries for each of the number of cores was px = 2 and py = P/2, where P was the number of cores. So, for 24 cores, the optimal geometry was x = 2 and y = 12, for 48 cores, x = 2 and y = 24, for 96 cores, x = 2 and y = 48. The performance for these uneven geometries was found out to be better than close to square geometries. This may be due to the fact for the uneven geometries with x =2 and y = P/2, also mentioned in section 4, the ghost cells that are exchanged in between the rows have values in the same cache lines. Hence, there are less number of cache misses. While, for more square geometries, more number of ghost cells are exchanged between left and right which result in the cache line between columns getting evicted and hence, results in more cache misses.

Examining the cost of Communication

Also, as can be seen, there is a notable increase in the cost of communication. This can be analysed using compute cost and communication cost concepts. For larger number of cores, for the same matrix size, the perimeter of the sub-matrix grows relative to the area if the sub-matrix. Hence, we have to wait for more MPI communications from neighbouring ranks. And thus, the communication overhead is more significant than for lesser number of cores.

c. <u>Scaling on comet(192 to 960)</u>

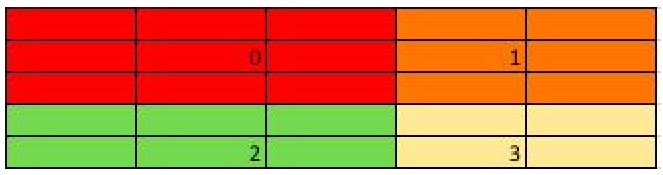
d) Next, we perform strong scaling for 192 to 960 cores. We use a matrix size of N = 8000 and number of iterations = 2000. The graph below shows that we still have strong linear scaling as shown in figure below. We use square processor geometry in this case to plot the results.



2. Analysis

a. Distribution

The matrix distribution assumes that all the initial values have been received on process 0. Then, the code for helper distributes all the initial values of the matrices in such a way so that each process receives egalitarian distribution of work. So, if the processor geometry evenly divides the size of the matrix, we make sure that each process gets equal size of a chunk of the matrix. We calculate the rows and columns using n/cb.py and n/cb.px. Now, if the processor geometry does not evenly distribute the matrix, then we increment rows and columns of each distributed chunk in such a way such that each processor does not get allotted more work than a row or a column of elements. So, a matrix of size 5*5 with a processor geometry of 2*2 will have distribution which goes like the following:



b. MPI Implementation

The Aliev-Panfilov Model requires the value of the cells in the nearby region to calculate the gradient. When we run in multiple processors, the matrix is divided into smaller sub matrices and each processor runs the algorithm in the smaller sub matrix. So, each processor requires value from the nearby processor for calculations for cells on the edge. We can find the rank of the matrix for communication in the horizontal direction by adding and subtracting 1 from the current rank number. We can also find the rank of the matrix for communication in the vertical direction by adding and subtracting row number. This is how we handle the ghost cell exchanges.

For the inter processor communication, we used the commands MPI_Isend and MPI_Ireceive which are non-blocking calls. We used this to get data from the top, bottom, left and right.

When the edge of the sub matrix coincide with the edge of the matrix, we padded the matrix with extra rows and columns of ghost cells. To find whether the edges coincide, we used variable x which shows the x-coordinate of the processor and variable y which finds the y coordinate of the processor, where x = rank % cb.px and y=rank/cb.px. If x=0, then the sub matrices top edge is coinciding with the global matrices top edge. On the other hand if x=cb.x-1, then the sub matrices bottom edge is coinciding with the global matrices bottom edge. Similarly we can find whether the sub matrix coincide with the edge of the matrix for the y direction also.

We distributed rows and column for the sub matrices in such a way that no processor has more than 1 row or column more work than any other processor. We implemented this using the following formula:

rows = cb.m / cb.py + (y < (cb.m % cb.py))
columns = cb.n / cb.px + (x < (cb.n % cb.px))

Where cb.m is the number of rows in the global matrix,
cb.n is the number of columns in the global matrix,
cb.py is the y dimension of the processor,
cb.px is the x dimension of the processor

This ensures that processors towards the left and processors towards the right has 1 more row or column if the matrix size and the processor dimensions are not divisible.

We used MPI_Isend to send the data, MPI_Irecv to receive the data. We used these non blocking commands so that the performance will not be hampered by blocking. We used MPI_Wait after calling MPI_Isend and MPI_Irecv, so that the processor waits till all the packets are received before implementing the Aliev-Panfilov model

<u>Pseudo code for MPI Implementation</u>

If edge in sub matrix boundary is overlapping with edge in global matrix boundary: Add ghost cells

The below code fills in the TOP Ghost Cells if the sub matrices top edge overlaps with the top edge of the global matrix

```
if (y1==0)
{
for (i = 0; i < (n+2); i++) {
E_prev[i] = E_prev[i + (n+2)*2];
}
}
```

Similarly, we did for bottom, left and right.

Add edges of sub matrix to buffers to be send by MPI:

The below code adds left column of the matrix in send left buffer

```
 if (x1!=0)
  {
   int j=0;
   for(i=n+3; i<(m+1)*(n+2); i+=n+2)
     {        buffer_left_sen[j] =E_prev[i];
    j++;
           }
       }
```

Repeated this for right, top and bottom

```
MPI_Isend( buffers)
MPI_Irecv( buffers)
MPI_Wait( )
```

Copy the value in buffers to respective positions in the matrix

The below code copies the value in buffers to respective positions in the matrix

```
if (x1!=0)
  {
//left
   int j=0;
```

```
        for(i=n+2; i<(m+1)*(n+2); i+=n+2)
            {      E_prev[i]=buffer_left_rec[j];
             j++;
            }
    }
```

Using the indirect method as mentioned on the assignment, we shut off communication for the run on 8 cores and measure the communication overhead that is obtained. We found the overhead to be around 1.9%. This should be due to ghost cell exchanges while doing MPI_Send and MPI_Rcv. But the overhead is close to negligible. This is due to the fact, that the number of cores and the size of the matrix are less, hence, the sub-matrices do not have to wait a lot for message passing from the neighbouring matrices.

First N = 400 and iters = 500, we check the single processor performance and it is found out to be 0.4647 GFlops. While, our parallel MPI has a performance of 0.4642 GFlops. The overhead is negligible, close to 0.1%. The overhead is probably due to the MPI_Init and MPI_Finalize calls, that take some part of the run time. But since the overhead is negligible, there is no need to check the code again to eliminate this overhead which maybe be due to extra conditional statements introduced while doing ghost cell exchanges.

MPI Calls that were used

MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, MPI_Wait, MPI_Reduce.

3. Development Process

We developed code starting the with following sequence of steps, where we ran into several bottlenecks:

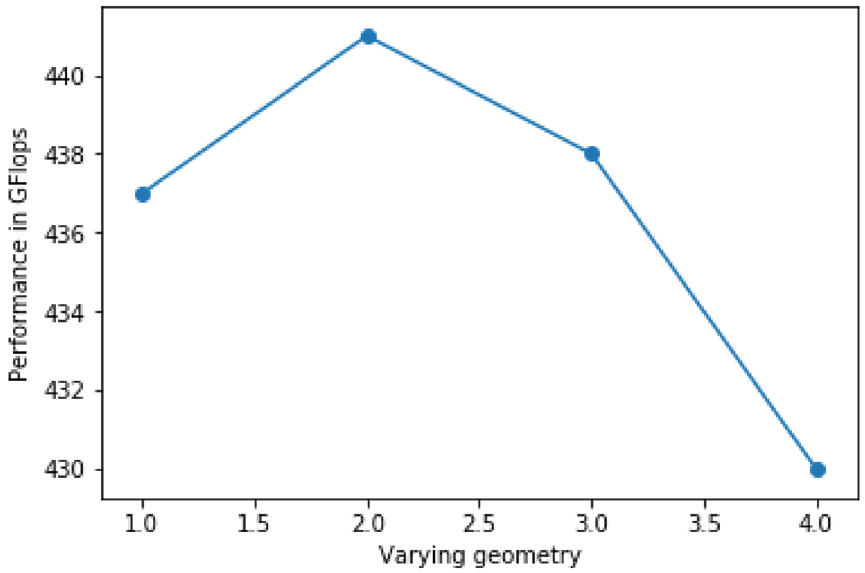
First, we developed code for helper.cpp which was used to distribute the initial values of E,E_prev and R across all the processors from rank 0. But at first, we were sending the whole matrices to solve.cpp. This was a mistake on our part because sending the whole matrix would have reduced the performance a lot. Because, the ghost cell exchanges would have taken for the huge E_prev matrix. Thus, we developed the code in such a way so as to send the chunk of the three matrices. The dimension of the chunk being determined by the rank of the process. Thus, MPI was taking place for smaller chunk of the matrices and also the boundary conditions were determined for the smaller chunks.

To enable this to take place, we created another file called "arblock.h" where we declared a structure which consisted of dimension of the rank and also the chunks of matrices that were being generated. Next, we went on to implement solve.cpp.

In this case, we developed the code for ghost cell exchanges. We didn't face a bottleneck here as it was quite straightforward. We just had to make sure that the messages were being passed in between the correct ranks. We used MPI_ISend and MPI_IRecv which are non-blocking calls. Thus, we had to use MPI_Wait for each of the message sends.

The bottleneck that we faced was in calculating the l2-norm and l_infinity because each rank was calculating a part of the final matrices E and R. Thus, we had to calculate sum of squares and absolute maximum value for l_infinity for each rank and send it to process 0 for the final calculation. We are using MPI_Reduce (with MPI_SUM) to add up the sum of squares values and send them to process 0. Also, we are using MPI_Reduce (with MPI_MAX) to send the maximum value to the rank 0.

4. Determine geometries



First point corresponds to geometry 1 x 96, second point corresponds to geometry 2 x 48, third point corresponds to geometry 4 x 24, fourth point corresponds to geometry 8 x 12.

As can be observed from the graph above, the optimal geometry for which we get the highest performance is x = 2 and y = 48. So, as expected the performance reduces as the processor geometry gradually attains more of a square form. This is due to the fact, that the processor in x-dimension should be less than that in the y-dimension for optimal performance. This is because of more ghost cell exchanges taking place between top and bottom than between right and left.. If the x-dimension is less, then there is more cache locality and the date can be found in one cache line. Thus, there will be lesser number of cache misses. But in the case when x-dimension gets larger and comparable to y-dimension, there are more exchanges happening in between right and left and thus, we have to copy each of the elements in each of the rows from the neighbouring processes and copy in a buffer to be sent to the neighbouring rank. Thus, data in a cache line will be evicted due to lesser cache locality. Within 10% of the top performance is also obtained from geometries x = 1, y = 96 and x = 4, y = 24.

5. Future Work

In the future if we have had enough time we could have implemented the following:

- AVX and SSE : We could have vectorized the code using AVX or SSE registers. This is expected to give higher performance because computation operations can be performed on multiple data in parallel. Thus, the instruction count could have significantly reduced.

6. References

http://mpitutorial.com/tutorials/mpi-send-and-receive/
https://www.mpich.org
https://en.wikipedia.org/wiki/Advanced_Vector_Extensions