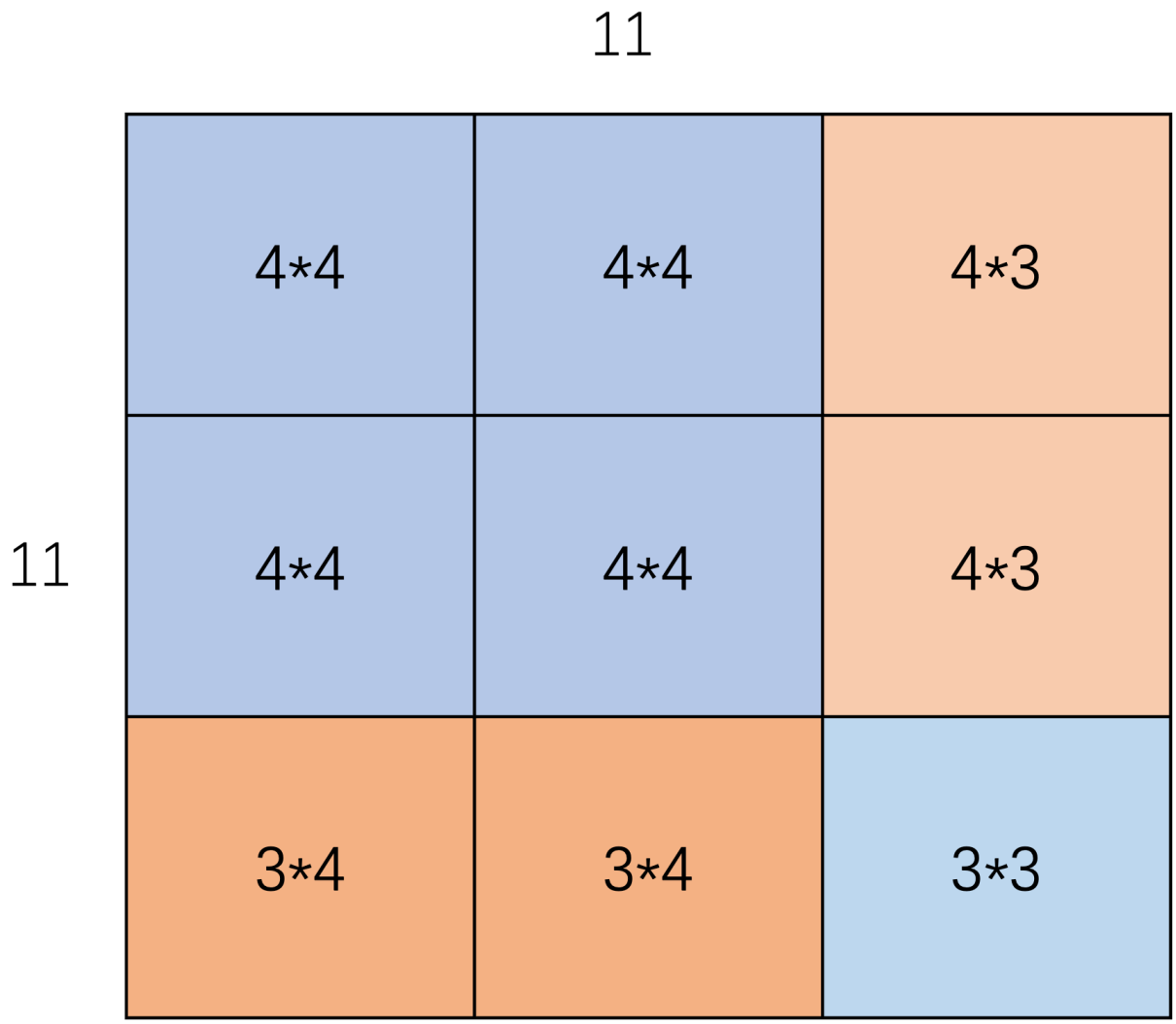
# CSE260 Assignment 3 - Aliev-Panfilov Cardiac Simulation

Author: Le Yang, Yongming He, Zihan Zhang, Zehui Jiao

## 1 Section (1) - Development Flow

### 1.1 Q1.a. How Program Works

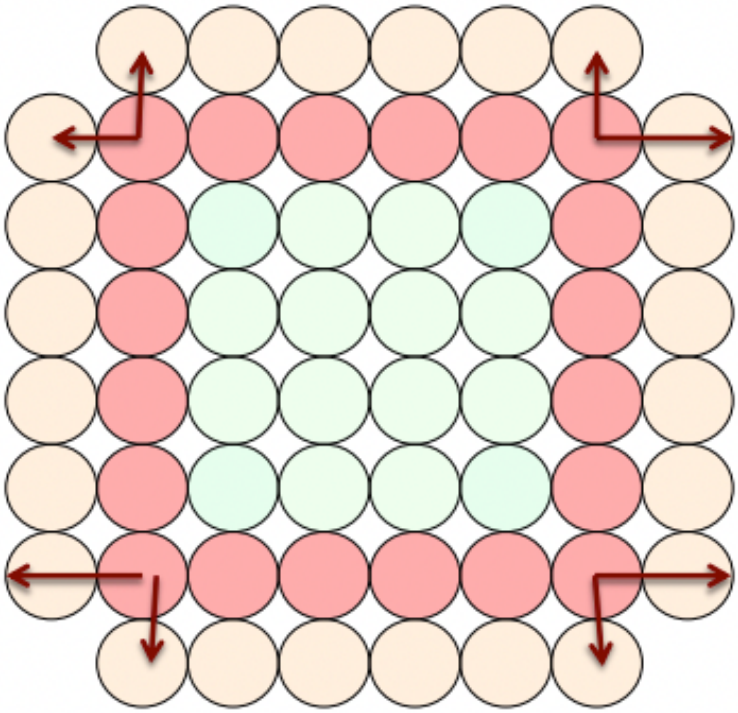
Given a matrix of dimension `cb.m x cb.n` and processor geometry `size = cb.py x cb.px`, we will divide the matrix evenly into blocks. In this situation, the block size is $\frac{cb.m}{cb.py} * \frac{cb.n}{cb.px}$. But when the processor geometry doesn't divide the mesh evenly, we will have $cb.m \mod cb.py$ and $cb.n \mod cb.px$ matrices have one more row or col respectively. In this situation, we will assign smaller blocks in the bottom-right corner. And we will use an example to illustrate it.



From the figure above, $cb.m = cb.n = 11$, $cb.px = cb.py = 3$. We could find that the larger blocks are in the top-left corner and smaller blocks in the bottom-right corner. And we guarantee that the amount of work assigned to the most heavily and most lightly-loaded process along a given dimension will differ by not more than one row and column of the solution array.

So far we have assigned the blocks, now we need to distribute the initial data from process 0 to all processes. And we use `MPI_Scatter` to send needed data to all processes. Before scatter, we reorganize the initial matrix such that elements in each block are stored sequentially in memory(array). After scatter, every process should place the received data in a local array. And `MPI_Scatter` sends data with the same length to all processes. So we need to set the send data length as the size of larger block and pad with 0. In this way, it won't interfere next steps.

Then we simulate Aliev-Panfilov equations iteratively. First, each block needs boundary information from neighbors or ghost cells that copies the interior of the block itself. We use `MPI_Isend` and `MPI_Irecv` to send and receive data to/from neighbors. These functions are asynchronous and we could do some computation when waiting for synchronization. So we do computation of green cells(fused version) when waiting for the communication. Computation on these cells do not depend on the boundary. After communication we use `MPI_Waitall` to wait for synchronization.



After `MPI_Waitall`, we could compute the red cells(fused version) in the graph. In fused version, PDE and ODE are computed in the same loop.

Finally, after all iterations, process 0 uses `MPI_Reduce` to gather L2, Linf norms from all processes. And we also implement the method of gathering all data into process 0 and plotting it.

### 1.2 Q1.b. Development Process

1. At first, we find out corresponding dimension, indices and stride and use asynchronous communication for each process to communicate around ghost cells with their neighbors.
2. In order to gather L2, we implement `stats_helper` function to sub blocks. Then we use `MPI_reduce` to gather the data. So far, the program is correct.
3. But we should pretend only process 0 knows the initial data and it should distribute it to other processes. So we design a function using `MPI_Scatter` to send data with same length to other processes. And every process will place data correctly after scatter.
4. To improve performance, we try to do some computation when waiting for synchronization. We find that the computation of inner cells in each block don't depend on other blocks. So we could do such computation when sending. This could improve performance apparently.
5. Then we try fused and unfused Aliev-Panfilov equations and different geometry of processors. We could achieve strong scaling on required size.
6. Finally, We try to implement EC-a by using `MPI_Gather` which is Inverted function of `MPI_Scatter` to gather E of each block. The process 0 will scatter the data and gather the data and then plot it.
7. And we also try logarithm scatter. It's time-consuming if only process 0 sends data to others. We want to achieve hierarchical scatter so process 0 could send data to some processes and then they could send data to next layer. But it still has some bugs and we don't have enough resources so we don't apply it.

### 1.3 Q1.c. How the Development Evolved

The most important idea we tried is doing computation when waiting for synchronization. And we record the GFLOPS of N2.

| Core | Before(GFLOPS) | After(GFLOPS) |
|---|---|---|
| 256 | 1747 | 2118 |

We could see that there is a huge improvement cause we utilize the waiting time to do computation.

## 2 Section (2) - Result

### 2.1 Q2.a. Performance Study on 1 to 16 Cores with N0

| Core | MPI(GFLOPS) | Origin(GFLOPS) |
|---|---|---|
| 1 | 11.10 | 11.30 |

From the table, we could say that the overhead of mpi itself can be ignored.
If we just use 1 core in mpi, the overhead of mpi is init and finalize. The GFLOPS is very close, and this overhead could be omitted if we increase the number of cores. So we won't take it into consideration in next experiments.

| Cores | MPI Running Time(s) | MPI(without comm)Running Time(s) | overhead |
|---|---|---|---|
| 1 | 3.27 | 3.25 | 0.8% |
| 2 | 1.76 | 1.56 | 2.2% |
| 3 | 2.76 | 2.72 | 1.5% |
| 4 | 1.94 | 1.87 | 3.3% |
| 5 | 1.51 | 1.47 | 3.2% |
| 6 | 2.58 | 2.41 | 7.2% |
| 7 | 2.24 | 2.09 | 7.1% |
| 8 | 2.02 | 1.86 | 8.5% |
| 9 | 1.77 | 1.64 | 7.5% |
| 10 | 3.34 | 3.04 | 9.7% |
| 11 | 2.99 | 2.74 | 9.3% |
| 12 | 2.75 | 2.49 | 10.8% |
| 13 | 2.65 | 2.40 | 10.4% |
| 14 | 2.54 | 2.28 | 11.5% |
| 15 | 2.36 | 2.08 | 13.3% |
| 16 | 2.26 | 2.01 | 12.2% |

We measure the overhead of mpi communication from 1 to 16 cores. We could say that the absolute communication time is very small but the proportion is still large and it will increase with the number of cores. Maybe the variation is larger with the increase of processor cores.

And we get two points from the table.

1. The geometry should be considered carefully. In this part, we increase the number of cores, but the running time may not decrease. So we need to explore new geometry to get better performance.
2. The proportion of communication is still large. So to improve the performance, we could do part of computation when waiting for ghost cells.

## 2.2 Q2.b. Scaling Study on 1 to 16 Cores with N0

| Cores | MPI Running Time(s) | GFLOPS |
|---|---|---|
| 1 | 10.60 | 13.52 |
| 2 | 5.42 | 26.45 |
| 4 | 2.66 | 53.86 |
| 8 | 1.34 | 106.5 |
| 16 | 0.69 | 207.4 |

| Cores | 1->2 | 2->4 | 4->8 | 8->16 |
|---|---|---|---|---|
| **Scaling** | 97.8% | 101.8% | 98.9% | 97.4% |

From the table, we could say that our implementation has a strong scaling from 1 to 16 cores. The scaling is nearly 100% so the performance scales absolutely linearly for up to 16 cores. And in this experiment, we don't need to explore the geometry so they are all 1D.

## 2.3 Q2.c. Scaling Study on 16 to 128 Cores with N1

| Cores | Geometry | GFlops |
|---|---|---|
| 16 | px=2, py=8 | 212.5 |
| 32 | px=2, py=16 | 415.4 |
| 64 | px=2, py=32 | 767.7 |
| 128 | px=8, py=16 | 1194 |

| Cores | MPI Running Time(s) | MPI(without comm)Running Time(s) | overhead |
|---|---|---|---|
| 16 | 44.2 | 42.8 | 3.2% |
| 32 | 22.6 | 21.3 | 5.8% |
| 64 | 12.3 | 11.0 | 10.6% |
| 128 | 7.5 | 6.6 | 12% |

| Cores | 16->32 | 32->64 | 64->128 |
|---|---|---|---|
| **Scaling** | 97.7% | 92.4% | 77.7% |

From the table above we can see that from 16 cores to 32 cores we can get the scaling at 97.7%, and get more than 200 GFlops at 16 cores. However, we can also see that from 32 cores to 64 cores, we achieve the scaling at 92.4%, and from 64 to 128 cores we get scaling ~80%.

To investigate the reasons behind this, we measure the communication overhead at each core numbers. We use the -k option to enable and disable communication during running and get the running time to calculate the overhead. We find that when we change from 32 to 64 cores, the communication jumps to 10.6% of the running time, therefore we cannot get the good scaling from 64 to 128 cores. Besides, we can see that we achieve more than 700 GFlops at 64 cores, and more than 1100 GFlops at 128 cores, which is a very good improvement over 16 cores result.

## 2.4 Q2.d Performance Study on 128 to 384 Cores with N2

| Cores | Geometry | GFlops |
|---|---|---|
| 128 | px=8,py=16 | 198 |
| 256 | px=32, py=8 | 540.7 |
| 192 | px=8, py=24 | 392.8 |
| 384 | px=6,py=64 | 2118 |

| Cores | MPI Running Time(s) | MPI(without comm)Running Time(s) | overhead |
|---|---|---|---|
| 128 | 71.7 | 70.8 | 1.2% |
| 256 | 24.2 | 10.6 | 56.2% |
| 192 | 35.4 | 34.8 | 1.7% |
| 384 | 8.2 | 5.7 | 30.5% |

| Cores | 128->192 | 192->256 | 128->256 | 256->384 |
|---|---|---|---|---|
| **Scaling** | 132.3% | 103.5% | 136.5% | 261.1% |

We can see that with N2, we can get 136.5% scaling from 128->256, which is definitely strong scaling. Also we can see that we can get 2118 GFlops at 384 cores. We also use the -k option to enable and disable communication during running and get the running time to calculate the overhead. I think the performance and scaling is very nice because when transmitting the data, we do PDE and ODE computation for the inner block, and when the cores becomes bigger, this will definitely reduce the impact of communication overhead over the running time, also more cores means there is more memory/cache, more memory/cache bandwidth and more network bandwidth, which improve the scaling.

## 2.5 Q2.e Explain the Communication Overhead Difference between <=128 and >128 cores

As we can see that the overhead increases slightly when the number of cores increases from 128 to 192. However, we have much longer overhead for cores 256 and 384.

As number of cores increases, the size of blocks decreases, resulting in more times of communication for the same length of information. As per communication, we do less computation. Therefore, the proportion of communication time increases. The reason why 256 cores have higher overhead than 384 core is because the geometry we used on 256 cores is much worse than 384 cores. The number of rows is much bigger than the number of columns for the geometry of 256 cores which causes more access to non-sequential data during communication.

## 2.6 Q2.f Cost of Computation

| Cores | Computation Cost = #cores x computation time |
|---|---|
| 128 | 72.4*128=9267.2s |
| 256 | 26.5*256=6784s |
| 192 | 36.5*192=7008s |
| 384 | 6.8*384=2611.2s |

384 cores is the optimal, because each core only runs for 6.8s. Although it uses triple the resources compared to 128 cores, it runs less than one thirds of the time 128 cores run. Under the condition that we have enough resources, we should choose 384 cores.

# 3 Section (3) - Determining Geometry

## 3.1 Q3.a. Top-performing Geometries at N1 for p=128

| Top Geometries at N1 | px=2, py=64 | px=4, py=32 | px=8, py=16 | px=16, py=8 |
|---|---|---|---|---|
| Performance(Gflops) | 1150 | 1188 | **1194** | 1145 |

## 3.2 Q3.b Patterns and the Reasons

To model the running time, let $T_{comm}$ denote the communication time over $P = px * py$ cores with problem size at $N1$, $\alpha$ denote the message latency and $\beta$ denote the bandwidth, then we have:

$$T_{comm} = 2(a + 8\beta^{-1}N/px) + 2(a + 8\beta^{-1}N/py)$$

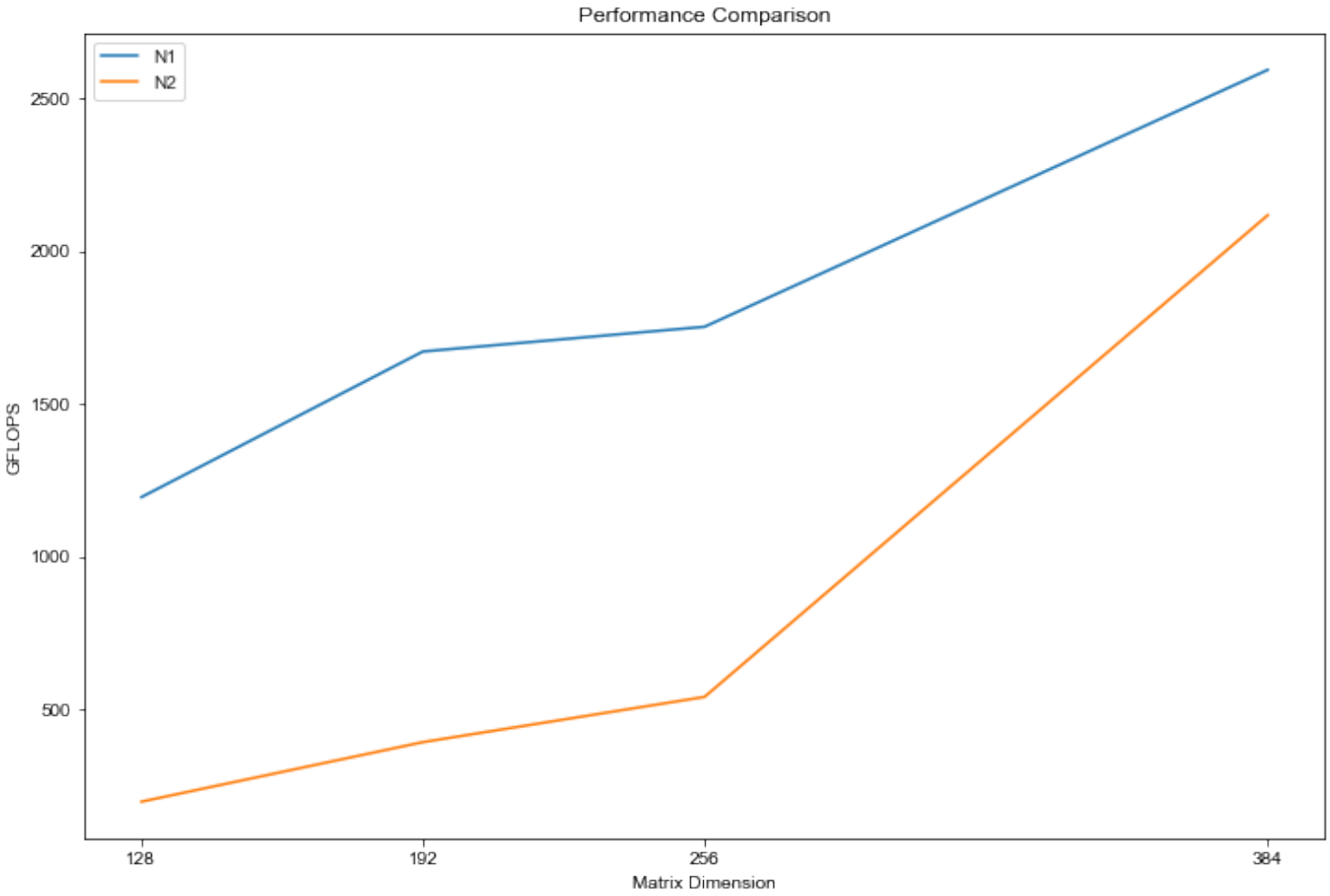
let $T(P, (N1, N1))$ denote the running on $P = px * py$ processors with problem size $N1$, let $m = N1/py$ and $n = N1/px$ then we have:

$$T(P, (N1, N1)) = T(1, (m, n)) + T_{comm}$$

From the table in Q3.a, we can see that we can get higher results when px is lower than py, however, if px is too low, the performance would be low too. From the above model, I think the reason is that when px is lower, there are more elements stored along the row in each subblock of the process, which can aid caching of the elements as rows are stored in contiguous memory locations, and this can reduce $T(1, (m, n))$. However px cannot be too low, because when the elements in the row is not a multiple of the cache line size, this will negatively impact the performance. Also if the difference between px and py is too large, $T_{comm}$ will become bigger, which will also increase the total running time. Therefore, px needs to be appropriately small, but the difference between px and py should not be too large, which explains the result in the table.

# 4 Section (4) - Strong and Weak Scaling

## 4.1 Q4.a. Compare and Graph your Results from N1 and N2 for 128, 192, 256 and 384 Cores.




| Performance(GFlops) | 128 | 192 | 256 | 384 |
|---|---|---|---|---|
| N1 | 1194 | 1671 | 1752 | 2593 |
| N2 | 198 | 393 | 541 | 2118 |

We can see that because N2's problem size is bigger than N1, at the same geometry, N1 can always get better results than N2. Also from the graph, we can see that for both problem size, we can achive strong scaling.

## 4.2 Q4.b. Explain or Hypothesize Differences in the Behavior

In this section, we model the network time is as follows:

$$Message\ passing\ time = \alpha + \beta_{\infty}^{-1} n$$

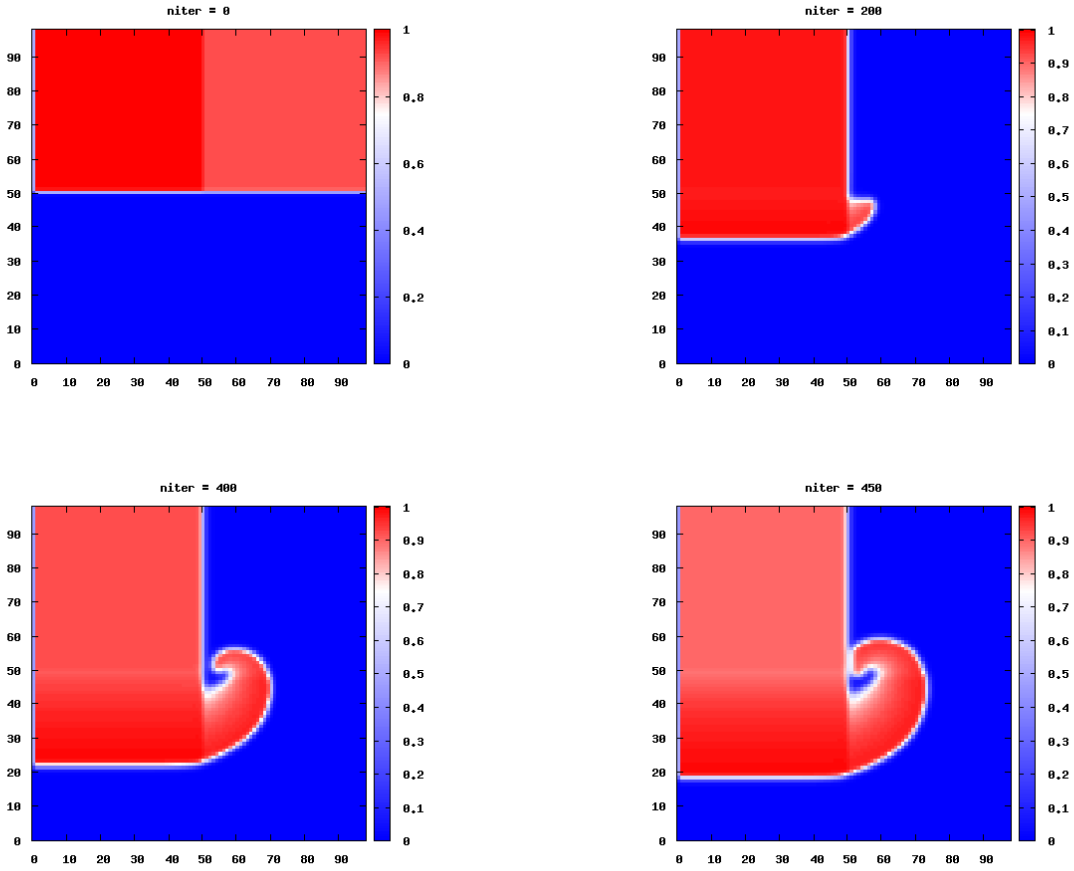
where $\alpha$ is the message latency, $\beta_{\infty}$ is the peak bandwith, and $n$ is the message length. we can see that as problem size grows, overhead for parallelism will grow. Therefore, there is weak scaling when the problem size from N1 to N2. So we can see that the performance of N2 for the same cores is lower than N1.

Also, as we can see from the table above that there is strong scaling from both experiments for N1 and N2. However, for N2, when the number of cores increases to 384, we see an abrupt burst of performance. This is due to weak scaling which happens when we increases both the problem sizes and the number of cores.

# 5 Section (5) - Extra Credit

## 5.1 EC-a

The current plotting routine only works on a single thread. So we modify the codes in `solve.cpp` and `Plotting.cpp`. By default, screens are displayed to the standard output. So in plotting, we need to change the output into file. Then in solve.cpp, we use `MPI_Gather` which is Inverted function of `MPI_Scatter` to gather E of each block. This process is very similar to scatter, but is the Inverted process. After gather, we just pass it to the `updatePlot` function and it will product a new image. And we show the intermediate result with matrix size is 500*500 and iteration is 500, which is the same as the `apf-ref`.




And in our implementation, the process 0 will scatter data at first and gather data when plotting. This will introduce much overhead. Besides, this process will do computation. So if we could create a new process to gather data and plot it, it's a better choice. And this process could be asynchronous, so our computation process doesn't need to block.

# 6 Section (6) - Potential Future Work

We tried to implement the logarithmic scattering by ourself, but get the wrong result and cannot figure it out because of the approaching deadline. Therefore, we will try to finish this part in the future. In addition, we will try to vectorize our code by hand in the future to improve the performance.

# 7 Section (7) - References


[1] https://mpitutorial.com/tutorials/

[2] https://www.sdsc.edu/support/user_guides/expanse