MPI Performance and Scaling Study

Weixiao Zhan

June 6, 2024

Aliev-Panfilov cardiac simulation

Aliev-Panfilov cardiac equations describe the signal propagation in cardiac tissue.

$$\begin{aligned}
\frac{\partial e}{\partial t} &= \delta \nabla^2 e - ke(e-a)(e-1) - er \qquad \text{on } \Omega_T \\
\frac{\partial r}{\partial t} &= -\left[\varepsilon + \frac{\mu_1 r}{\mu_2 + e}\right][r + ke(e-b-1)] \quad \text{on } \Omega_T, \\
\vec{n} \cdot \delta \nabla e &= 0 \quad \text{on } \partial\Omega \\
(e, r)_{t=0} &= (e(\cdot, 0), r(\cdot, 0))
\end{aligned}$$

Listing 1 shows the pseudo-code for numerical computation for above differential equations, in which, E, E_p, R are three $[(m+2) \times (n+2)]$ size matrices.

```
For a specified number of iterations
for (j=1; j < m+1; j++) {
for (i=1; i < n+1; i++) {
    // NOT a PDE solver, but calculating the Laplace operator
    E[j,i] = E_p[j,i] + alpha*(   E_p[j,i+1] + E_p[j,i-1]
                                + E_p[j+1,i] + E_p[j-1,i]
                                - 4 * E_p[j,i]);
    // ODE SOLVER
    E[j,i] += -dt * (kk * E_p[j,i] * (E_p[j,i]-a)
                  * (E_p[j,i]-1) + E_p[j,i] * R[j,i]);
    R[j,i] +=  dt * (eps+M1* R[j,i]/(E_p[j,i]+M2))
                  * (-R[j,i]-kk*E_p[j,i]*(E_p[j,i]-b-1));
}}
swap E_p and E
End repeat
```

Listing 1: Pseudocode for numerical simulation

1 Development Flow

The Aliev-Panfilov cardiac simulation program has simple data dependency. thus can be scaled to multi-processes using MPI library.

1.1 Program Description

There are three stages in simulation program, called apf, execution: init, solve, and reduce. The over Memory layout of one matrix is shown in Figure 1.

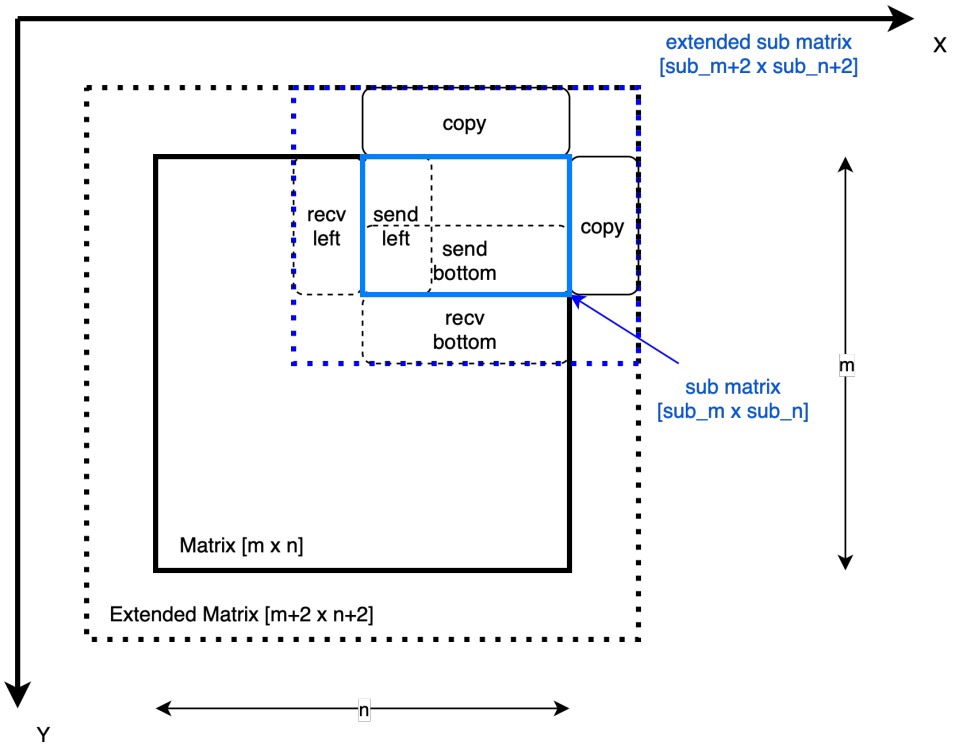

Figure 1: layout

1.1.1 init

The main process initialize extended matrix of E_p and R with starting conditions; then uses MPI to distribute extended sub matrices to their corresponding working processes.

1.1.2 solve

For every iteration, each process first exchange ghost sells of E_p with their neighbor processes. Specifically, the process need to pack the left and right cells its needs to send and unpack the received left and right ghost cells. Top and bottom ones can be sent and received in place as the matrices are stored in row major order. The exchange of ghost cells are using non-blocking MPI to reduce waiting. If the process is on the edge, the ghost sells are copy updated by the row or column that are 2 cells away.

After ghost cell exchange, each process perform the numerical computation shown in Listing 1 to solve the PED. The numerical computation along with some of the packing and unpacking routine are vectorized using compiler hints.

1.1.3 reduce

After or during the simulation, the program might need to gather results to one process. The program supports reducing two types of results.

One is gathering all sub matrices of E to reconstruct the full matrix of E, which can be helpful for plotting and downstream computation that requires the full simulation results. This is achieved by regular MPI send and receive routine.

The other is just gathering L_∞ and L_2 norm of E, which can be helpful for correctness check and save bandwidth. This is achieved by `MPI_Reduce` with max and sum reduce operator respectively.

1.2 Development Process

1. Implement initial distribution routine;
2. Implement ghost cell exchange routine;
3. Change numerical simulation code to tile loop (hoping to improved cache hit rate);
4. Implement reduce routine;
5. Change numerical simulation code to use SIMD;
6. Change ghost cell exchange routine to use SIMD;
7. Batch jobscript for N0, N1, N2.

1.3 Performance Improvements

Along the development process, the perform are measured at $m = n = 800$ with 4×4 process grid on a local machine. The performance improvements are shown in Figure 2. The multi-processing and SIMD improved performance.

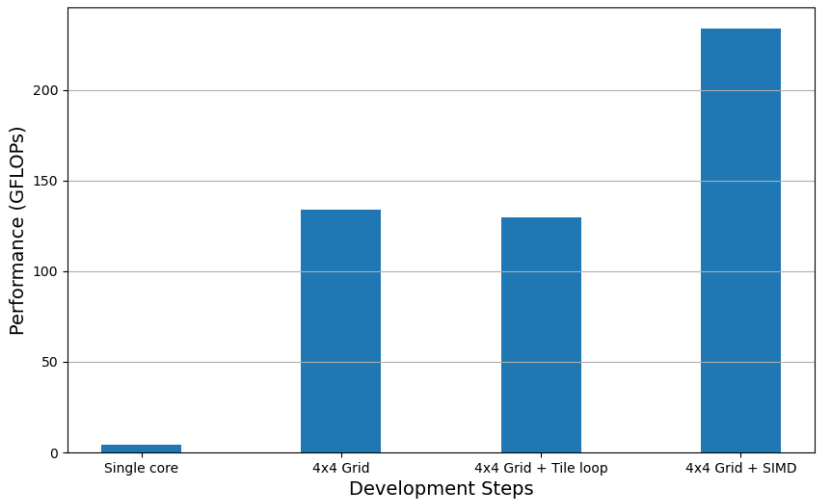

Figure 2: Performance Implements

2 Results

2.1 Single Processor Performance

Figure 3 shows the peak performance reached at N0 problem size.

MPI overhead caused $-0.3\%, 4.4\%, 3.7\%, 8.5\%, 13\%, 20\%$ of performance loss on $1, 2, 4, 8, 12, 16$ processes respectively. At 16 processes, 6% out of 20% performance drop comes from computation. Reminder 14% comes from other non-communication overhead.

The most significant non-communication overhead is repeated ghost updates. In no communication implementation, all processes consider them as edge processes and copy update its ghost cells in every iteration. In no MPI version, the number of ghost cell is the circumference of matrices $= O(2m+2n)$. Whereas in multi processes, the number of ghost cell is number of processes * circumference of each sub matrices $= O(2(m \cdot py + n \cdot px))$ The more processes in the program, more computation spent on ghost cell updates.

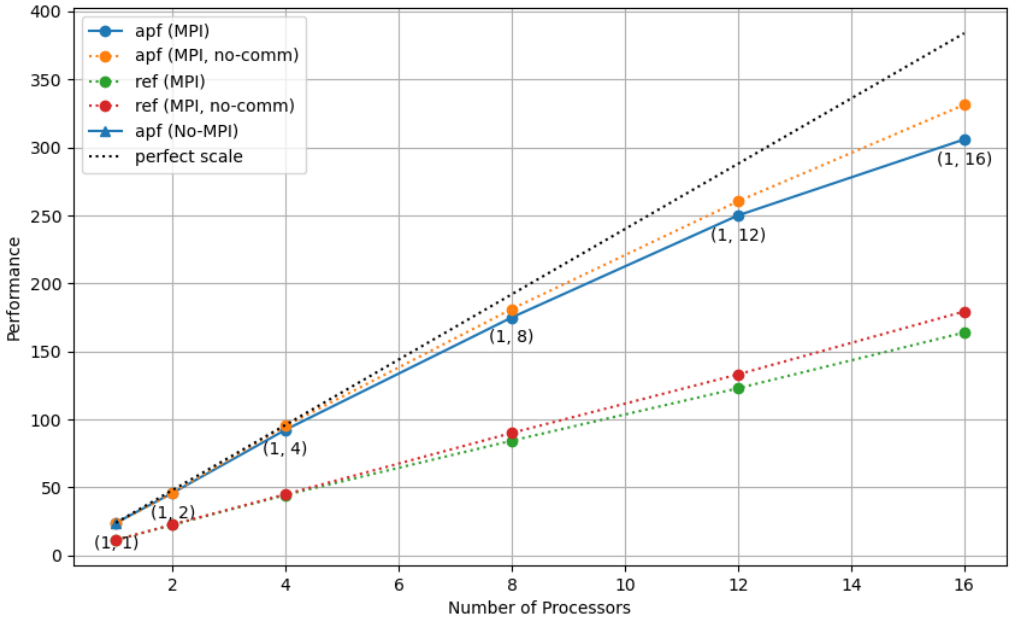

Figure 3: Singe Processor Processor

2.2 Strong Scaling Study (1-16 cores)

Figure 4 shows the speedup and efficiency of the program on N0 problem size. The figure might have suggested my apf program scale less efficiently. However, the true factor is the base lines performance. My apf program would skipped unnecessary MPI routines when running in 1 MPI process, , leading to a higher base line. Thus despite the scale efficiency is lower, my program runs significantly faster than reference program on all process numbers.

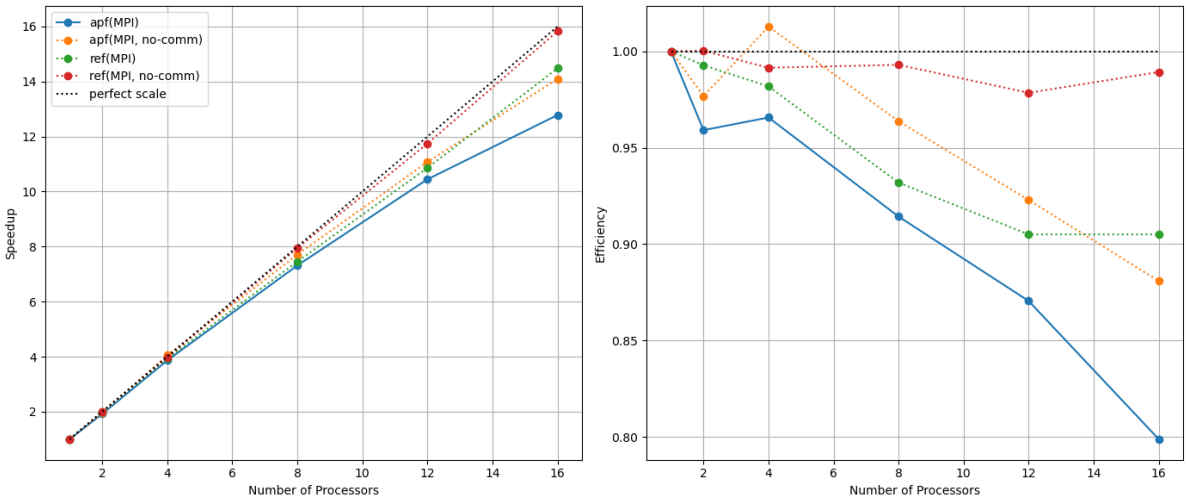

Figure 4: N0 scale study

2.3 Strong Scaling Study (16-128 cores)

Table 1, Figure 5 and Figure 6 shows the peak performance and scale efficiency at N1 problem size.

From 16 to 64 processes, the program gradually use up all cores on one AMD EPYC Processor. At 128 processes, the program takes up one entire node with two CPU socket and the MPI is communicating between Non-Uniform Memory (NUM). Despite the memory been different, MPI communication caused 14.7% of performance decrease at 64 processes, and 15.2% at 128 processes, which are fairly close, suggesting socket bus is not limiting the scalability of the program.

Figure 6 shows the scale efficiency of the program on N1 sized problem. Similar to N0 scale result, my apf has better performance than reference program at all processes sizes. The scale efficiency is lower due to better baseline result. Suggesting some optimization employed by my apf program become less effective as the number of processes increases.

Processes	Geometry	GFlops
16	8×2	381
32	8×4	705
64	8×8	1250
128	8×16	1916

Table 1: N1 geometries

2.4 Large Core Count Performance Study

Table 2, Figure 7, and Figure 8 shows the peak performance and scale efficiency at N2 problem size. Doe to reference program is too slow, skip running them on N2.

Based on experience, the processes geometry should be around square and has slightly larger py. Such geometries will results in near square and slightly wide sub matrices for each process. Because the square sub matrices has the bast circumference to area ratio, they have the least ghost cell to exchange. The wider sub matrices are also preferred, as top and bottom ghost cells can exchanged in place and no need to pack and unpack.

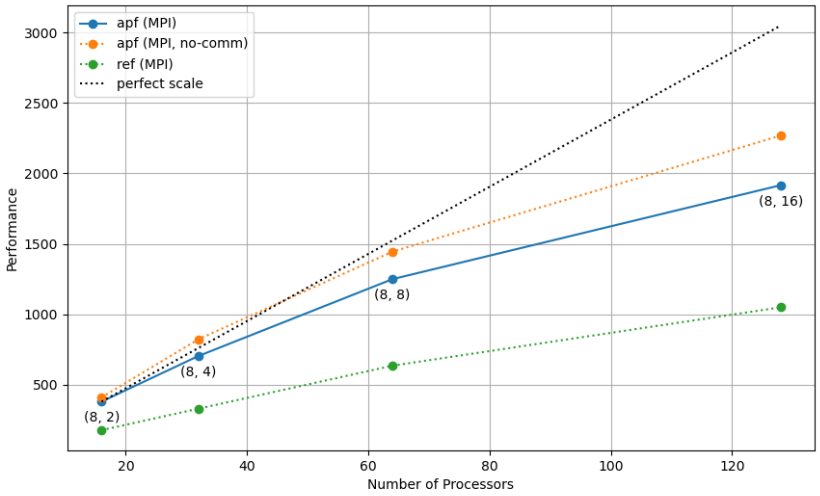

Figure 5: N1 performance

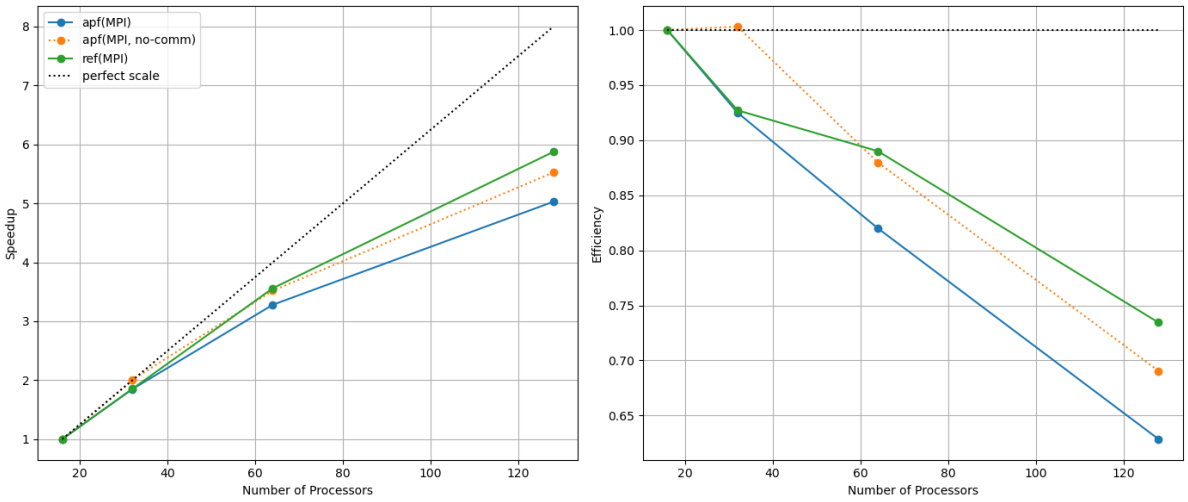

Figure 6: N1 scale study

Processes	Geometry	GFlops
128	8×16	360
192	16×12	542
256	4×64	601
384	16×24	2109

Table 2: N2 geometries

2.5 Communication Overhead Differences

For processes number less then 128, all processes nested on one node. As discussed in previous sections, the communication cost didn't surge due to in-node NUMA.

From 128 to 384 process on N2, the program spam on multiple nodes. As shown in 7, some computation cost surged but some didn't. I believe the cause is the processes are randomly distributed on different nodes, Causing different number of ghost cell ex-

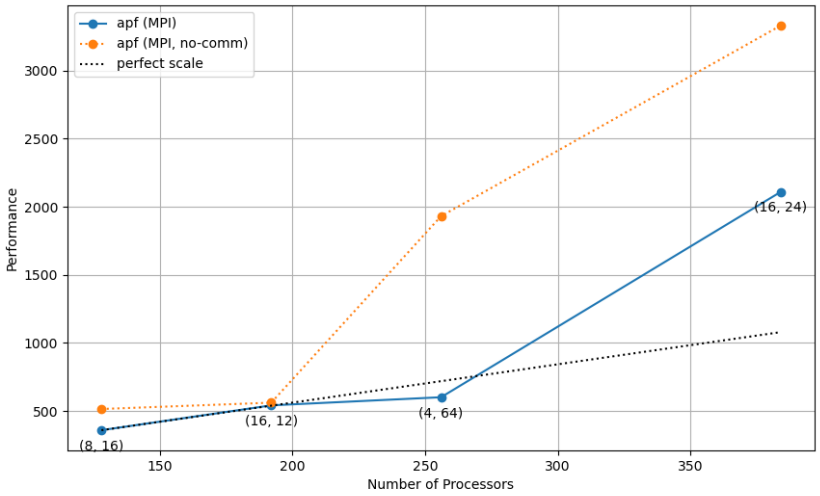

Figure 7: N2 performance

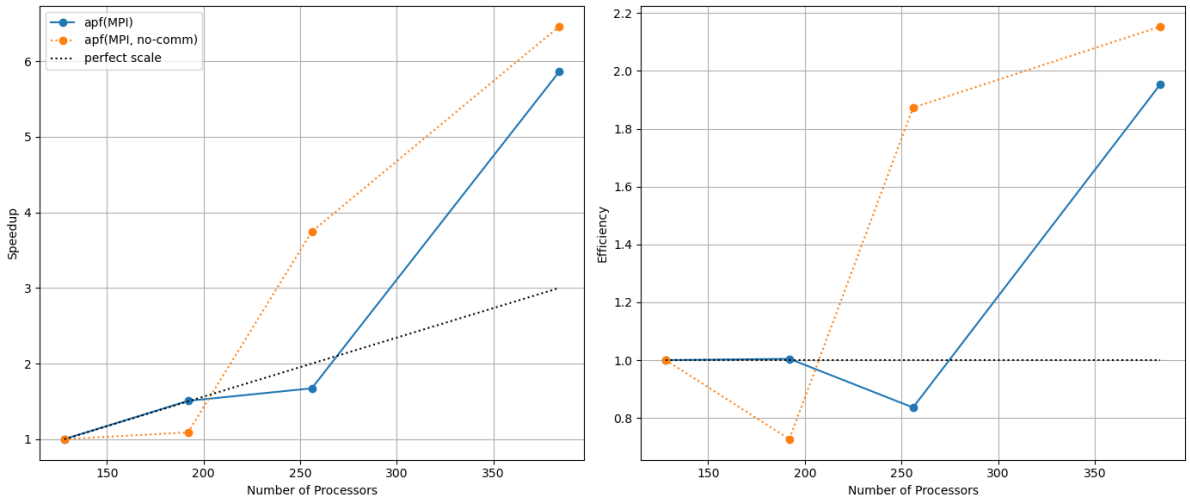

Figure 8: N2 scale study

change happened between nodes. Experiment also shows the performance on N2 vary significantly. And the variance could due to the random distribution of processes.

On a two nodes 256 processes program, if the processes are distributed following their geometry, the between-node communication can reduce by half in best case then worst case.

At 384 processes, the remote direct memory access (RDMA) is automatically disabled. MPI might have used a different routine for communication when RDMA is disabled. Which could leads to the performance surge at 384 processes.

2.6 Computation Cost

Based on 3, 384 processes yields best performance and lowest cost.

Processes	**Computation cost** (SUs)
128	1.416
256	1.410
192	1.693
384	0.725

Table 3: Computation cost

3 Determining Geometry

3.1 Top-Performing Geometries

At p=128, the top-performing geometries at N1 are 8×16 and 16×8.

3.2 Geometry Patterns

The processes geometry should be around square and has slightly larger py. Such geometries will results in near square and slightly wide sub matrices for each process. Because the square sub matrices has the bast circumference to area ratio, they have the least ghost cell to exchange. The wider sub matrices are also preferred, as top and bottom ghost cells can exchanged in place and no need to pack and unpack.

4 Strong and Weak Scaling

4.1 Strong Scaling Comparisons

The results of best N1 geometries are shown in Table 1 and Figure 5 The results of best N2 geometries are shown in Table 2 and Figure 7

4.2 Scaling Behavior

The scaling behavior are shown in Figure 6 and Figure 8 In summary, the program scale efficiency decrease as the number of process increases due to inevitable ghost cell communication and updates (discussed in previous sections). When the number of process exceeds 128, i.e. the program is running on multiple node, the scale is much less efficient.

5 Extra Credit

5.1 multi-processes plotting

Shown in Figure 9, the program supports plotting when running with multi-processes. The plotting process does not do computation.

5.2 vectorization

The computation kernel and ghost cell exchange are vectorized.

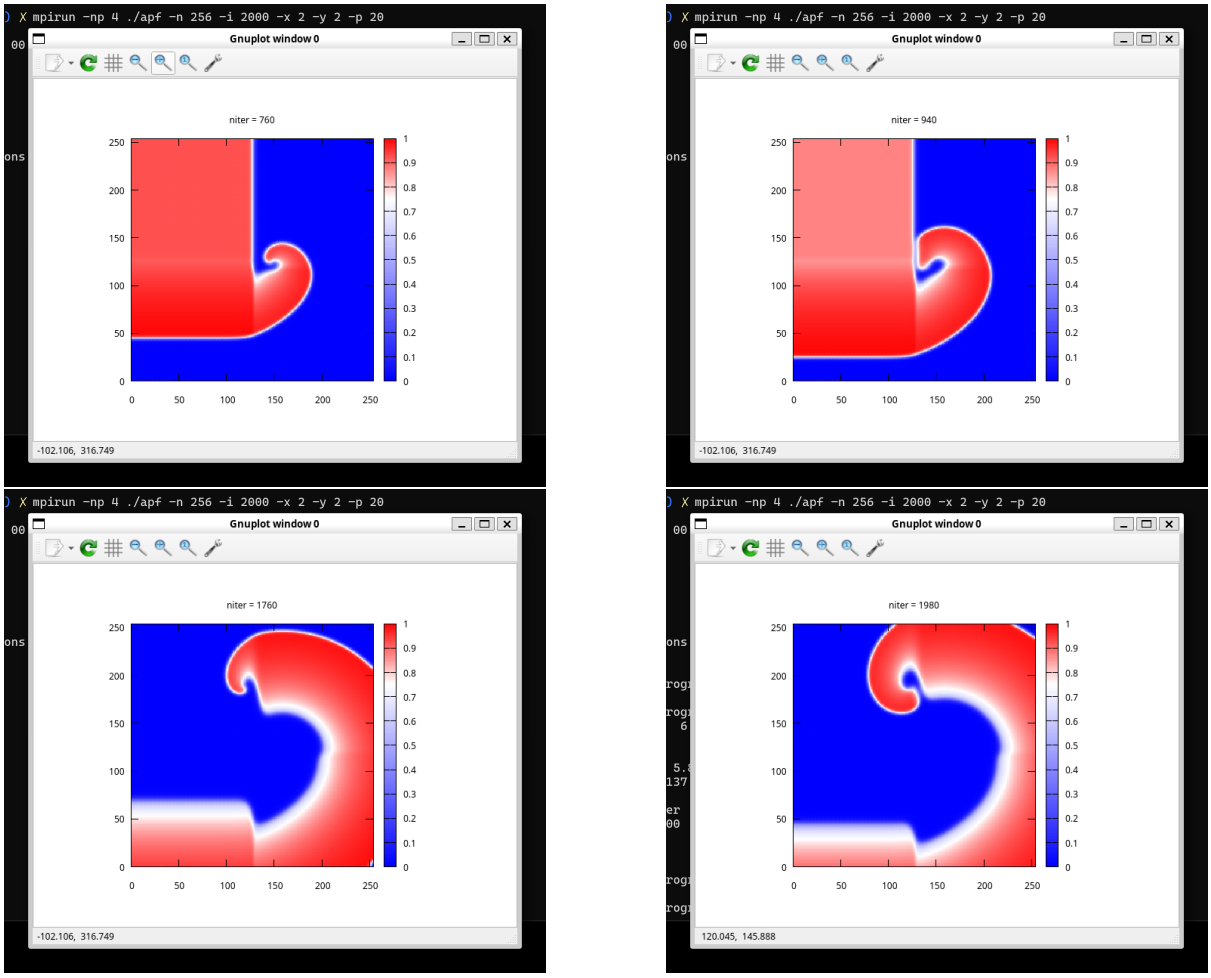

Figure 9: live plot with 2x2 processes

6 Potential Future Work

1. better distribution schedular to distribute process on multiple CPUs and/or multiple nodes and minimize the between-node communication. Essentially, break the full matrix into multiple level of sub matrices, and each computation unit (single processor, CPU die, node, server cluster, ...) takes their corresponding tasks based on this hierarchy.

2. investigate RDMA on and off performance differences.