MPI Implementation of 2D Poisson Solver

Carlos Vazquez Gomez

March 12, 2019

1 Presentation of the Algorithm

This algorithm uses non-blocking `MPI_Isend` and `MPI_Irecv` to initiate data inter-process communication of block edge buffers. After posting sends and recvs on all four edges (eight total calls), a process begins computation. If a point update requires a neighboring process's data, it consults the *previously* received buffers from neighbors.

After this computation update step, a process blocks (if necessary) in an orderly manner to wait for all the processes that it sent and/or received edges from. To avoid deadlocks, processes with an even x-coordinate talk first to southern neighbors, while those with an odd x-coordinate talk to northern neighbors. Similarly (and after North/South communication), processes with an even y-coordinate talk first to eastern neighbors, while processes with an odd y-coordinate talk first to western neighbors.

Every 1000 cycles, a process checks its infinity norm convergence error ($||T^{k+1} - T^k||$) and determines if it should stop computing. If it's time to stop, on the next (last) communication with neighbors, it sends a 1 on the last entry of each edge buffer, thus signaling that the neighbor should remember this last buffer and never wait for a new one from it again.

2 Verification

We verify the second order accuracy and the ability to solve heterogeneous grids (grids with different numbers of x and y points) of our 2D Poisson solver. For a simple visual inspection see Figure 1, where the output of 100 processes solving a 1x2 grid with source terms $x \cdot e^y$ with error threshold $1e$-12 are displayed in ParaView.

2.1 Second Order Accuracy

We test the grid convergence behavior of our algorithm by solving 20x20, 40x40, 60x60, ..., 380x380, 400x400 grids with four processes on a 1x2 sized rectangular

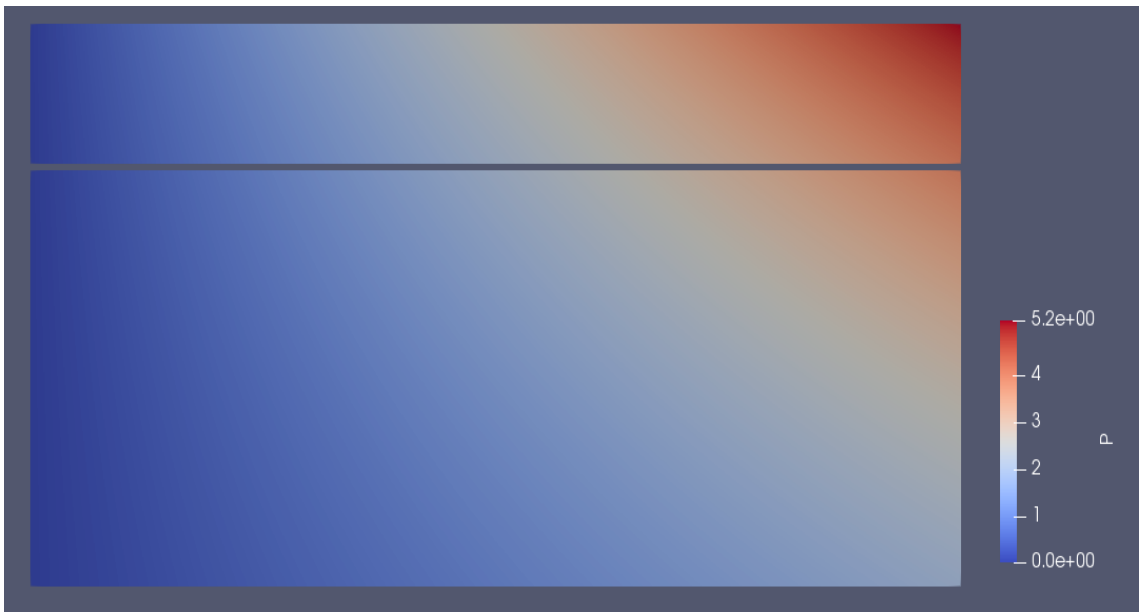

Figure 1: Solution to Y-stripped, 1 by 2 region with 40,000 points and 100 processes, and with convergence error threshold 1e-12. Process 74's output is hidden to reveal partitions

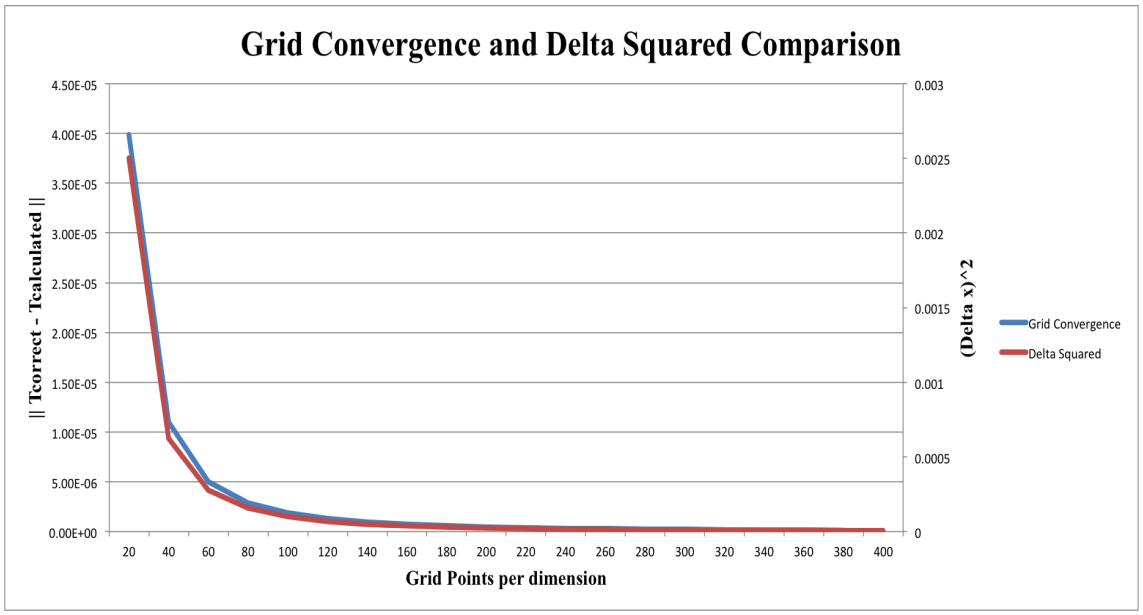

Figure 2: The decay of absolute error follows the decay of Δx^2 with a constant factor of 0.0159 as grid points increase

region with source values $x \cdot e^y$ and convergence error threshold $1e$-12. The infinity-normed error of analytic to computed solution is shown to be linearly proportional to Δx^2 via the constant 0.0159. Figure 2 reveals the decay of the absolute error as grid points increase. The linear proportionality of the absolute error and the square of the step size confirms that the algorithm is second-order accurate.

2.2 Solving Heterogeneous Grids

Now we verify that our solver works for cases where the x and y dimensions have different number of grid points by running our program on the same 1x2 sized rectangular region on various heterogeneous grids and checking the absolute error. From Table 1, we can see that the computation errors from solving heterogeneous grids in the vicinity of the homogeneous grid 60x60 are comparable to that of 60x60. This proves that our solver can handle heterogeneous grids.

3 Convergence Analysis

Two understand how the number of processes affects the convergence behavior of our algorithm, we plot the normed (maximum) difference between a grid

Grid	Absolute Error
20x100	4.26e-05
40x80	1.12e-05
60x60	5.04e-06
80x40	2.82e-06
100x20	1.74e-06

Table 1: Solving on heterogeneous grids yields similar errors to solving on a homogeneous grid of 60x60

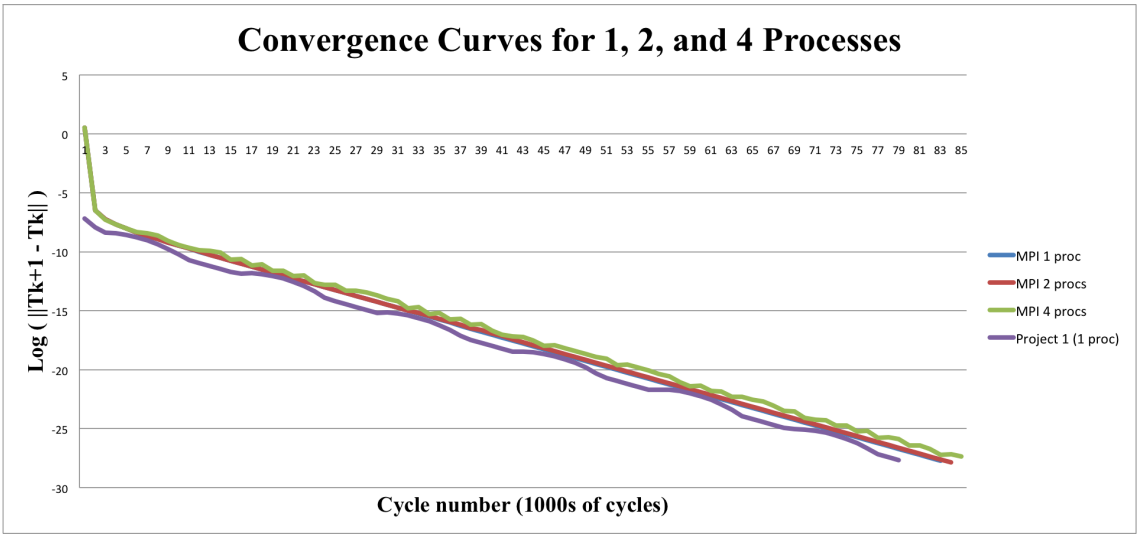

Figure 3: Convergences for different numbers of processes, and for the Project 1 implementation (purple)

and its immediate next version ($||T^{k+1} - T^k||$), and we do this every 1000 update cycles. We again use the 1x2 rectangular region with and a 200x200 grid with error threshold 1^2-12. Figure 3 shows the decay of the convergence error by plotting the natural log of $||T^{k+1} - T^k||$ against the cycle number the error was measured. The Project 1 implementation and the 4-process MPI implementation had periodic occurrences of faster convergence, but the reason for this is still not known. All MPI instances started converging very rapidly, and then slowed down before the 2000^{th} cycle. The difference between the Project 1 and the single MPI process convergence curves can be accounted for by the different point update orders, as the iterator in Project 1 travels starting from the top left, and the one in Project 2 starts at the bottom left.

4 Weak Scaling Analysis

To test the efficiency of our parallelization, we perform weak scaling analysis. The overhead of inter-process communications becomes evident when we solve a rectangle with increasing numbers of processes while keeping the points per process constant. For this analysis, we again solve the 1x2 region with error threshold $1e$-12 with 1, 4, 9, ..., 81, 100 processes while maintaining 400 points (20x20) per process. To see the difference in communication overhead between different groups of processes, we test three grid schemes: full block, x-stripped, and y-stripped. Figure 4 shows plots the efficiency (defined as $T_{serial}/T_{parallel}$) of the parallel code normalized with respect to the total number of cycles that each test requires (as more processes have more grid points which take more cycles to converge). Table 2 shows the actual times required to solve the grids for each test. There are two important things to notice. First, in Table 2, the number of cycles for the x and y stripped schemes are much larger than those of the full block scheme. This is probably due to the x/y-stripped schemes giving each process a higher surface-to-volume ratio, which requires more cycles since surfaces (process borders) are updated in a Jacobi form, which is slower than Gauss-Seidel. Also, even when we normalize the execution times with respect to the cycles-to-convergence, Full-block schemes are slower than x or y stripped schemes. This is seen in Figure 4, as the weak scaling curve of the Full-block scheme drops to about 20% efficiency, while the stripped schemes only drop to about 45% efficiency. This is explained by the fact that in stripped schemes, communication only happens in two, rather than four, directions for most processes. This means that half of the time that is spend blocking waiting for communications to complete in Full-block schemes is now unnecessary.

Therefore, we conclude that although there is less communication overhead per cycle in X-stripped or Y-Stripped schemes, they usually take significantly more cycles to converge, and this outweighs the benefit of the decreased overhead. This is clearly summarized by Table 2, which shows that Full-block schemes get the same job done faster than stripped schemes.

# Procs	Points per proc	Full Block		X-stripped		Y-stripped	
		Time (s)	Cycles (thousands)	Time (s)	Cycles (thousands)	Time (s)	Cycles (thousands)
1	400	0.015	≤ 1	0.015	≤ 1	0.016	≤ 1
4	400	0.097	5	0.080	4	0.256	13
9	400	0.214	9	0.307	15	1.150	56
16	400	0.353	15	0.864	42	3.291	159
25	400	1.001	23	2.576	93	10.00	360
36	400	1.707	32	5.086	182	20.02	697
49	400	3.385	42	9.713	316	36.33	1214
64	400	4.210	53	15.85	516	60.72	1956
81	400	6.385	66	30.02	779	101.3	2967
100	400	8.015	79	38.39	1147	146.8	4301

Table 2: Weak scaling analysis for three partitioning schemes. The "Cycles" field represents the number of cycles until convergence

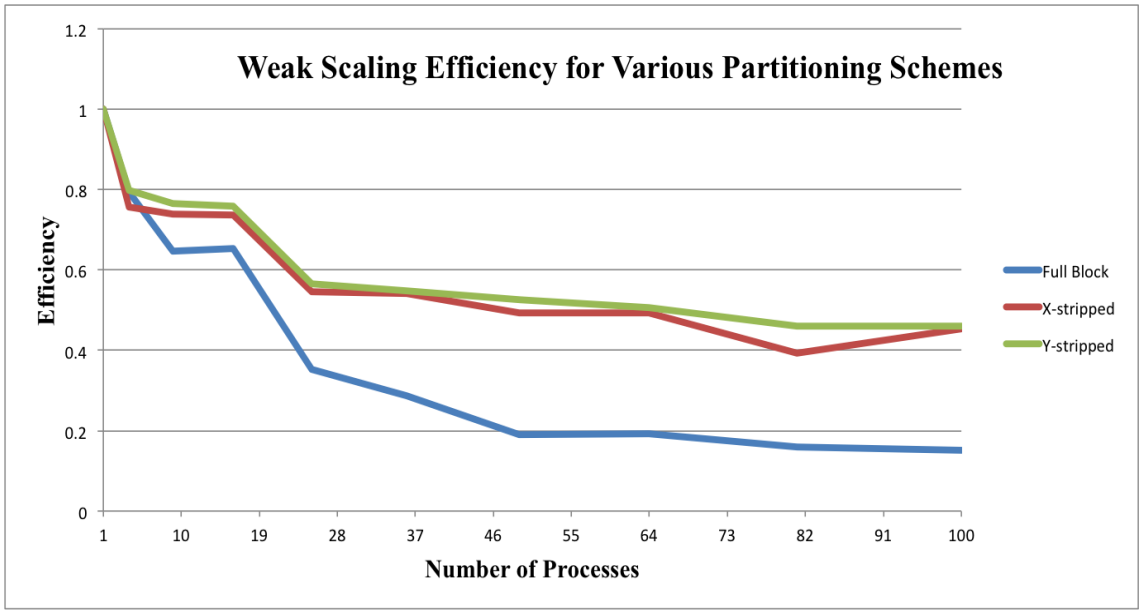

Figure 4: Efficiency $= \frac{T_{serial}}{T_{parallel}}$, where T represents the amortized time to compute a single grid update with 400 points per processor