HPC - 1 Class Project - Solution to 2D Laplace Equation

Siddhant S. Aphale,
Department of Mechanical and Aerospace Engineering,
University at Buffalo,
Person # - 50164327

December 20, 2016

Problem Definition

The aim of this project was to solve the Laplace equation in two-dimensions on a unit square grid. Dirichlet boundary conditions were implemented. The Laplace equation in two-dimensions is given as:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \qquad 0 < (x, y) < 1$$

The Dirichlet Boundary conditions are defined as follows:

$$u(x, 0) = sin(\pi x)$$
$$u(x, 1) = sin(\pi x)e^{-\pi}$$
$$u(0, y) = u(1, y) = 0$$

The analytical solution to this problem was given as: $u(x, y) = sin(\pi x)e^{-\pi y}$.
The aim was to develop a solver such that the solution to the PDE can be performed using distributed memory parallelism and scal well to at least hundred processors. Finite differences were used to discretize the PDE. The solver was developed in various stages and the conformance of the numerical solution with the analytical solution was verified.

Solution

Initially, a serial code was developed that uses the finite differences scheme as described below

$$\Delta \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} \tag{1}$$

where $u_{i,j} = u(x_i, y_j), i = 1, \cdots, N; j = 1, \cdots, M$. On implementation of this discretization, a linear system of equations was formed that gives the solution at each of the grid points, i.e. $u(i, j)$. The grid was created on the unit square domain with 98 grid points. This created 9604 linear equations. The system of linear equations was solved using the `dgesv` function available in the LAPACK library by Netlib. The solution obtained by this serial code was compared with the analytical solution using the contour plots for both the solutions. The contour plot is shown in the Figure 1 below

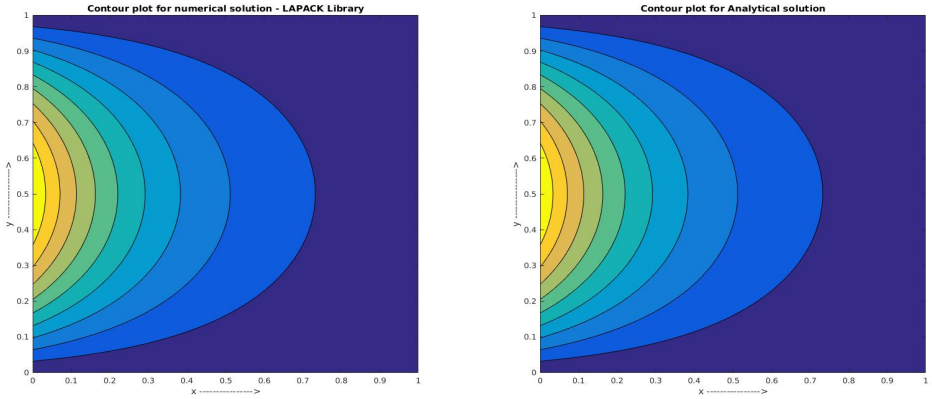

Figure 1: Verifying the Numerical Solution using LAPACK Library with the Analytical solution

The left plot represents the Numerical solution using the LAPACK library and the plot on the right represents the analytical solution for same number of grid points as above.
Once the serial code was developed, the next stage was developing the Conjugate-Gradient (C-G) Krylov solver that uses Jacobi preconditioning to solve the linear system of equations from the serial part [1]. The Jacobi preconditioning matrix M comprises of the diagonal elements of the matrix A generated above [5]. As above, grid was generated with 98 grid points on a unit square domain. The iterations were performed to meet a convergence criteria. It was observed that the solution was converged in just 133 iterations. However, the computation was slower. To verify the results obtained, contour plot was compared for numerical and analytical solution as shown in the Figure 2 below

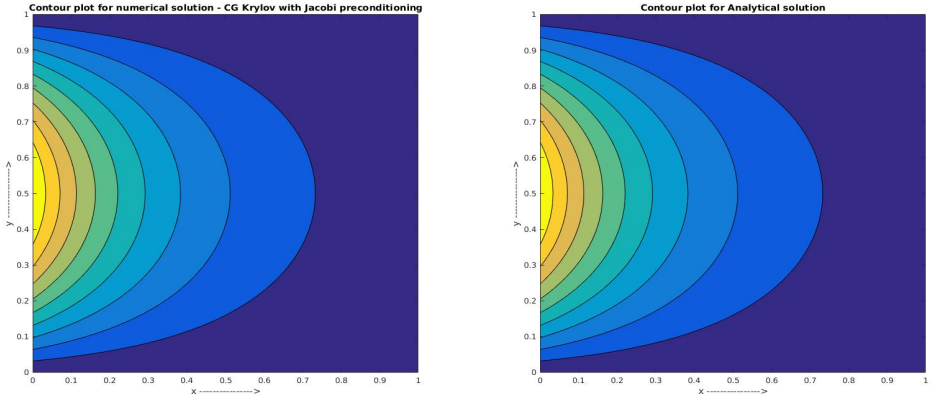

Figure 2: Verifying the Numerical Solution using C-G Krylov Solver with Jacobi Preconditioning with the Analytical solution

The left plot represents the Numerical solution obtained by Conjugate-Gradient (C-G) Krylov Solver with Jacobi Preconditioning and the plot on the right represents the analytical solution for same number of grid

points as above.
To reduce the computational time required in the C-G Krylov solver, the A matrix which comprises of a lot of zero-elements was developed in a sparse matrix format - Compressed Row Storage (CRS). This type of storage has three vectors. One vector contains all the non-zero elements of the matrix, the second vector contains all the column indices of these non-zero elements and the third vector contains the row pointers of these column indices. A computational kernel was developed further that will be used to perform the matrix-vector multiplication. This kernel was implemented in the C-G Krylov solver developed in the step above. Again the grid comprised of 98 grid points on the domain of unit square. As above, 133 iterations were required to compute the solution to the linear system. However, the computation time reduced significantly and now it took just 233.466 seconds to compute the solution. The results obtained were verified using the contour plots as earlier as in Figure 3

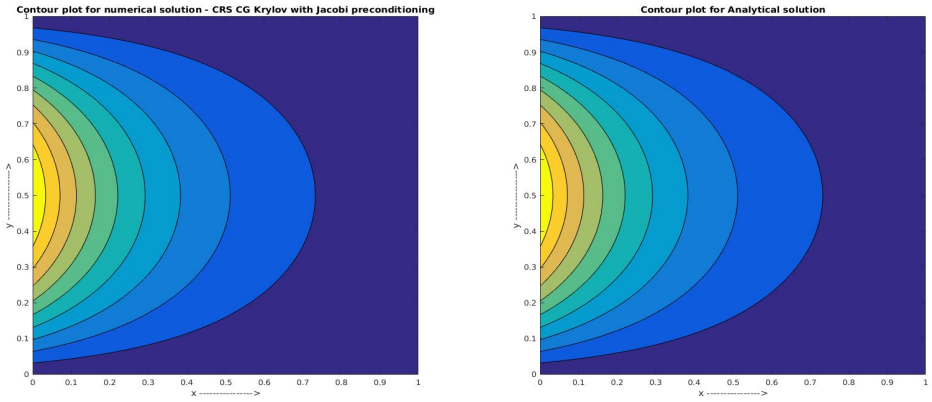

Figure 3: Verifying the Numerical Solution using CRS Matrix-Vector multiplication in C-G Krylov Solver with the Analytical solution

The next part of the problem was to parallelize the above generated solution using MPI. The first step was to deompose the domain into well-balanced chunks and assemble the locally owned grid points into the linear system. The chunk size was achieved by equally distributing the total number of linear equations on the processors. Accordingly the A matrix and the resulting B vector which was formed from the finite difference formulation was balanced and distributed on the processors. Further, the sparse matrix format developed earlier requires modification so that the processor will hold only those non-zero elements that are available in the linear system of equations. The decomposition was done only in one dimension by utilizing the row-blocking. Thus there was only need of ghost nodes transfer in one direction for outermost processors and in two direction for inner processors. The data for these ghost nodes was made available by keeping the whole data on all the processors. This will obviously increase the communication as well as memory utilization and will affect the scaling. `MPI_Allgatherv` was used for this purpose. Once the storage of matrix was decomposed, the next step was to parallelize the matrix-vector product computational kernel. The further step was to parallelize the Jacobi preconditioning and then the Conjugate-Gradient (C-G) solver itself. This was achieved by computing the residual, the values of p, z, α, β on each processor and then finally gathering the norm on root processor to check the convergence. The new matrix vector product kernel was used to iterate the C-G solver. The code thus developed was now used to solve the Laplace equation on up to 576 processors. The Infiniband communication constraint was set. All the nodes used to study the performance were constrained to 12 cores per node and the memory was set to maximum for each node i.e. 48 GB. The computation was done exclusively to achieve the best performance from each

processor. To check the performance and strong scaling, the load was kept constant and the processors were increased gradually. Care was taken that the chunk remained well balanced. To check this the grid size was set to 290 grid points. Thus the total number of linear equations that formed from the finite diference formulation was 84100. Thus the A matrix which is developed by eliminating the boundaries will consist 6879707136 elements and the resulting B vector will have 82944 elements. This is a sufficiently big problem and usage of 576 processors will develop a chunk of 144 elements in B vector per processor. Thus the use of the processors is justified. All the elements in A and B are of `double` data type. Thus the memory required just to handle the matrix and vector is very large. To study the weak scaling, the load was increased as the processors were gradually increased maintaining the balance of the chunks. The solution computed using the solver was verified with the analytical solution by comparing the contour plots for both the solutions. The comparing results are for 290 grid points and are shown in the Figure 4 below.

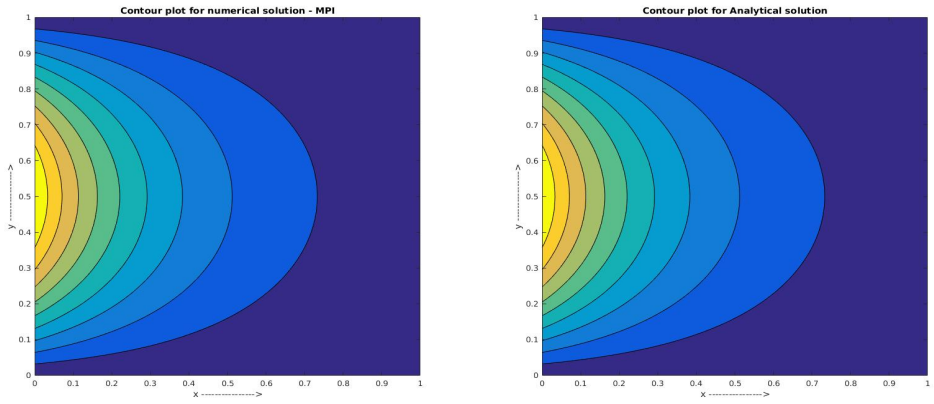

Figure 4: Verifying the Numerical Solution using MPI with the Analytical solution

It can be seen that the plots for both the analytical solution and numerical solution with the parallelized code are exactly same. Once the solution was verified, the scaling was performed. We study both strong scaling and weak scaling. Strong scaling is the study how the solution time varies with the number of processors for a fixed total load. Weak scaling on the other hand is the study of how the solution time varies with the number of processors for load fixed per processor. The time was noted for both the cases and speed up was computed. The speed up factor [2] is computed as

$$S_p = \frac{T_s}{T_p}$$

where, T_s is the execution time of the sequential algorithm, T_p is the execution time of the parallel algorithm with p processors. The parallel efficiency is calculated as follows

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

where p is the number of processors. The weak scaling was performed on up to 288 processors. Speed up factor and efficiency was computed as discussed above. Apart from this, Karp-Flatt Serial fraction was also computed to check the performance. The Karp-Flatt serial fraction [4] is given by

$$f = \frac{1/s - 1/p}{1 - /p}$$

where f is the Karp-Flatt serial fraction.

Strong Scaling

Strong scaling was studied on up to 576 processors as discussed earlier. Speed up factor was computed and plotted against the total number of processors as seen in following Figure 5

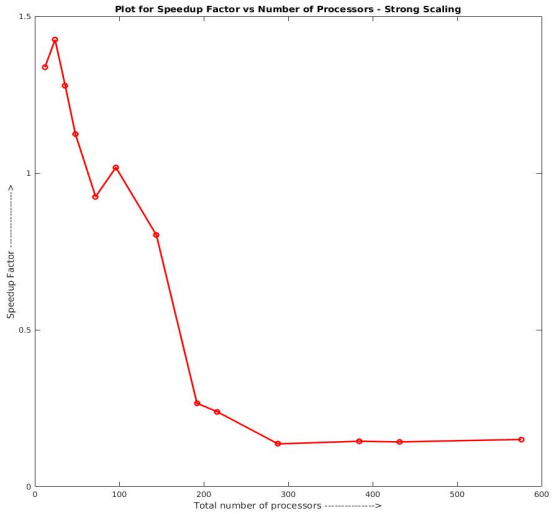

Figure 5: Speed up Factor for Strong Scaling

It can be observed from the plot that the speed up factor is decreasing as we increase the number of processors. Clearly, we are not able to achieve the scaling as expected in Amdahl's law. The plots for efficiency and Karp-Flatt serial fraction are as in Figure 6 and Figure 7

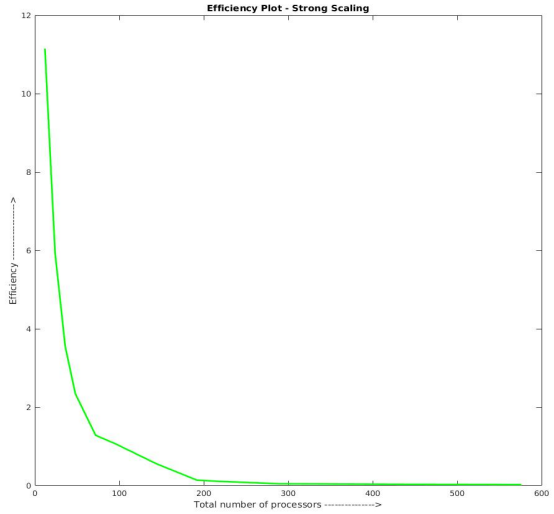

Figure 6: Efficiency plot for Strong Scaling

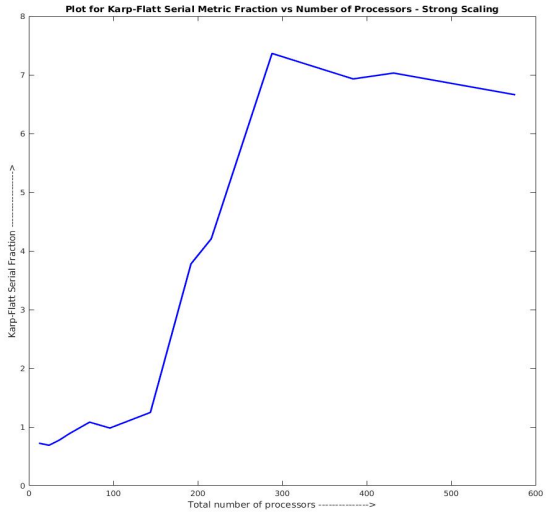

Figure 7: Karp-Flatt Serial Fraction-Strong Scaling

It can be observed that the efficiency is also very poor and is decreasing as we increasing the number of processors. Ideally, the serial fraction shall reamin constant as per Karp-Flatt. However, we see an increase in the serial fraction as the number of processors increase. This proves that the problem is not well scaled.

Weak Scaling

Weak scaling was studied by keeping the load on each processor same as we increase the number of processors. Weak scaling was studied for up to 288 processors. As in Strong scaling speedup factor, efficiency and Karp-Flatt serial fraction was computed and plotted against the total number of processors. The plots are shown in Figures 8, 9 and 10 below

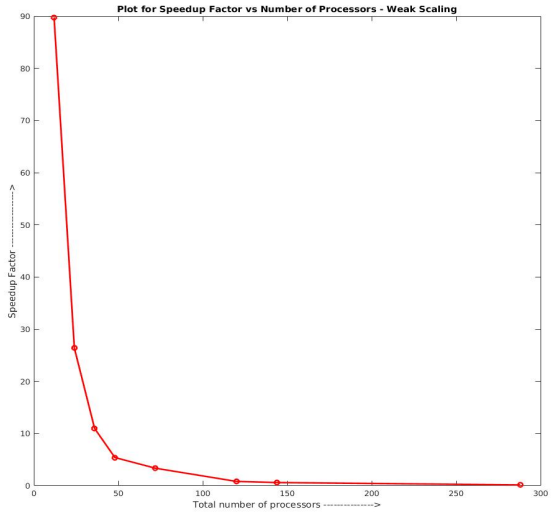

Figure 8: Speed up Factor for Weak Scaling

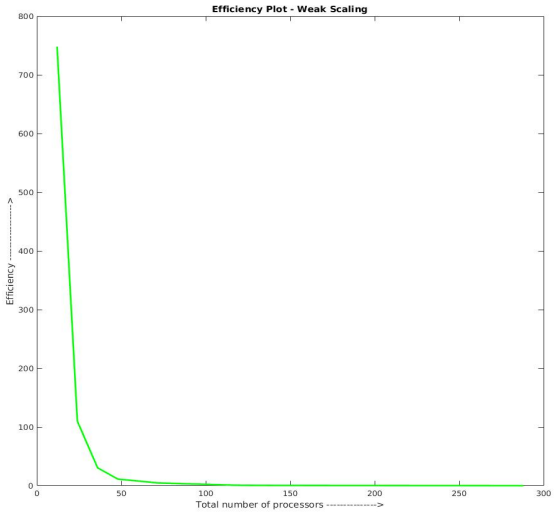

Figure 9: Efficiency plot for Weak Scaling

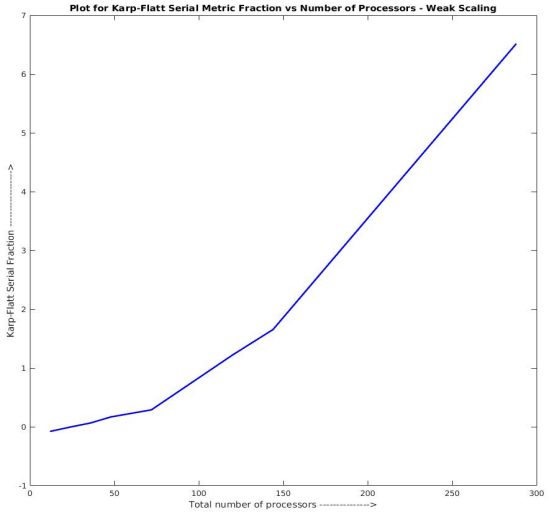

Figure 10: Karp-Flatt Serial Fraction-Weak Scaling

It can be observed that the speed up factor and efficiency gradually decrease for the weak scaling. However, the Karp-Flatt serial fraction is increasing approximately linearly. This means that the problem size is not growing fast enough to completely counter the loss of efficiency as more processors are added.
There can be a lot of possible reasons for the problem not scaling.The domain is decomposed using blocking-row instead of using 2-D decomposition. Furthermore, in the code `MPI_Allgather` is used to make the data available across al the processors. In strong-scaling this will require countless memory depending on the size of the problem and thus message passing increases. The message size is also large and it will kill the problem scalability. By considering the cost analysis, the parallel matrix-vector multiplication does not scale in this decomposition technique.

$$\lim_{p\to\infty} E_p(n) = \lim_{p\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] = 0$$

Thus, eventually the parallel efficiency becomes nearly non-existent [3] as can be seen in Figure 6
In weak scaling, we increase the problem size as we increase the number of processors. This makes the amount of memory available to store the problem scale linearly with total number of processors. The efficiency in this case is given as

$$\lim_{p\to\infty} E_p(n_{max}(p)) = \lim_{p\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}} \right] = 0$$

Thus, the parallel algorithm for matrix-vector multiplication does not scale. Also the message passing is linearly increasing as we increase the problem size and processors. Also the time for computation is limited. And efficiency cannot be maintained as the number of processors is increased and the exection time is capped according to

$$\lim_{p\to\infty} E_p = \sqrt{\frac{T\gamma}{p\beta}}$$

Summary

A parallel MPI code was developed to solve the two-dimensional Laplace equation. Conjugate-Gradient (C-G) Krylov solver with Jacobi preconditioning was used to solve the system of linear equation. The domain and the sparse matrix form was decomposed in a way to balance it across a number of processors equally. Strong and weak scaling was studied and the results were plotted in the form of speed up, efficiency and Karp-Flatt serial fraction. It was obsevred that the efficiency decreases and goes to negligible as expected by cost analysis for block-row decomposition with matrix-vector multiplication and `MPI_Allgather`. The sparse matrix storage form was useful in making the computation faster and Jacobi preconditioning ensured that the convergence was achieved faster.

References

[1] Conjugate gradient method. Available at `https://en.wikipedia.org/wiki/Conjugate_gradient_method`.

[2] Dr. Paul Bauman. Class notes and slides, Fall 2016.

[3] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. 2 edition, 2016.

[4] Alan H Karp and Horace P Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[5] C. Vuik. Iterative solution methods. 2015. Available at `http://ta.twi.tudelft.nl/users/vuik/burgers/lin_notes.pdf`.