MTH 5050 - Intro to Parallel Processing

Employing a Parallel Linear Solver from PETSc to solve Poisson's Equation in 2D

Bindi Nagda

Introduction

Partial Differetial Equations (PDEs) describe how a quantity evolves in space and time and they are prevalent in many science and engineering applications. A common equation that arises in many scientific applications is the Poisson's equation which is an elliptical type of PDE, and a generalized version of Laplace's equation. Physically, Poisson's equation describes how a function diffuses in space. For example, it can be used to describe the pressure field in an incompressible fluid flow, or to describe the electric potential field for a given charge distribution, or to describe the gravitational field caused by a massive object. Therefore, this PDE has huge applications when solving problems in fluid dynamics, electrostatics and gravity. Poisson's equation can be written for 1D, 2D or 3D problems prescribed in Euclidean space. In this paper, I will be focusing on a 2D problem defined on a subset of the cartesian plane. I will be using a numerical approach in order to obtain a solution to the PDE. Numerical methods for solving PDEs are common because most real world PDEs cannot be solved analytically and thus a numerical result is the next best answer we can obtain. There are many iterative solvers that have been and continue to be developed in order to tackle this problem. A very well know set of solvers caller Krylov Subspace (KSP) Solvers are common in the field of numerical PDEs. In particular, the Generalized Minimal Residual (GMRES) Scheme is a robust iterative solver that can enjoy fast convergence rates when used with an appropriate preconditioner. Multigrid preconditioning is being used increasingly in order to attain numerical results with high accuracy and efficiency. GMRES will be applied to the system we are trying to solve, and the preconditioner choice will be Geometric agglomerated algebraic multigrid (GAMG). The problem we want to solve is described in detail in the next section.

Problem Description

We will solve a 2D Poisson problem imposed on a square bounded region Ω = [0,1] x [0,1] with Dirichlet boundary conditions on the boundary $\delta\Omega$. The following elliptical PDE will be solved:

$$-\nabla^2 u = 20\pi^2 \sin(2\pi x)\sin(4\pi y)$$

$$u(x,y)|_{\delta\Omega} = 0$$

In the above equation $u(x,y)$ denotes the potential function that we are trying to solve numerically.

We will first discretize the square region into a fine grid that has (n+2) grid points along the x-direction and (n+2) grid points along the y-direction. Here, n is the number of interior grid points along each direction. For example, if n = 32, then the grid on which the problem is defined is illustrated in figure 1 below. This problem will be solved computationally on different interior grid sizes, going from n = 32 to n = 1024. The grid will be divided up in such a way that the spacing in both x- and y-directions will be equal, i.e., $\Delta x = \Delta y = h$. The grid spacing is calculated by $h = 1/(n+1)$. Please see an example grid illustrated below:

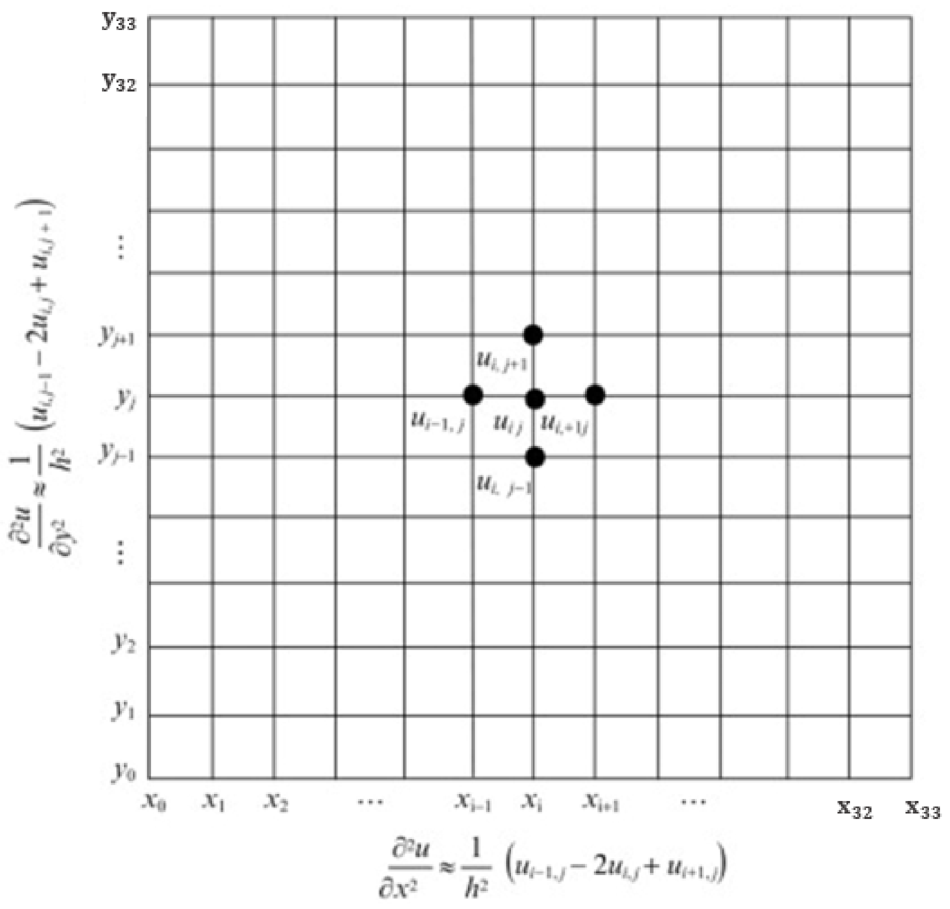

y_{33}
y_{32}
y_{j+1}
y_j
y_{j-1}
y_2
y_1
y_0
x_0 x_1 x_2 x_{i-1} x_i x_{i+1} x_{32} x_{33}
$u_{i,j+1}$
$u_{i-1,j}$
$u_{i,j}$
$u_{i+1,j}$
$u_{i,j-1}$
$\frac{\partial^2 u}{\partial y^2} \approx \frac{1}{h^2}\left(u_{i,j-1} - 2u_{i,j} + u_{i,j+1}\right)$
$\frac{\partial^2 u}{\partial x^2} \approx \frac{1}{h^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right)$

Figure 1: Discretization of region on which solution is to be found. *Source: Ref. [3]*

A second-order central differencing scheme will be employed in order to obtain the discretization of $\nabla^2 U$ as shown below:

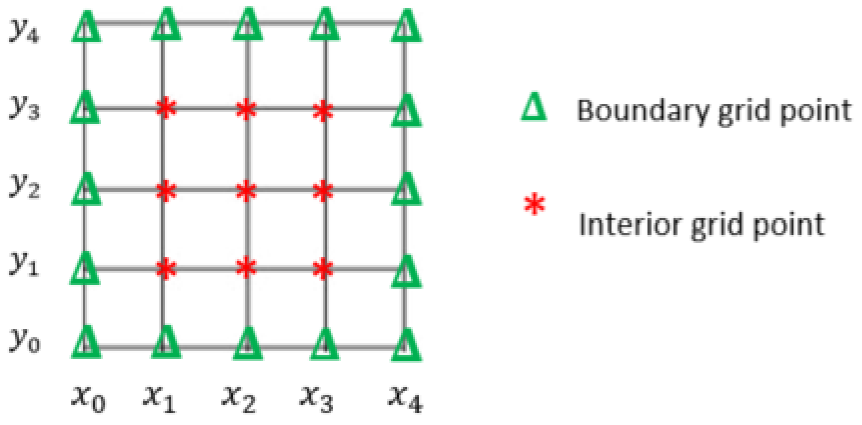
$$\nabla^2 u = u_{xx} + u_{yy} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}.$$

Now we have a linear system of equations which can be written in a compact form as $A\mathbf{u} = \mathbf{f}$.

Below, I will demonstrate what matrix A would look like if we used an interior grid of size 3-by-3:

$$A_{9\times 9} = \left[\begin{array}{ccc:ccc:ccc}
4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\
\hdashline
-1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\
\hdashline
0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4
\end{array}\right]$$

The matrix above arises from the following grid:



The interior points form a grid of size 3-by-3 which is also the size of each interior block in matrix A. We are trying to solve for the potentials at these interior grid points. The potentials at the boundary points are already known due to the boundary value condition, i.e., the potentials at the boundaries are all equal to zero.

The above matrix can be generalized for any n-by-n interior grid. It has a tridiagonal block structure as shown below:

$$A_{nn\times nn} = \begin{bmatrix}
\mathbf{M} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\mathbf{I} & \mathbf{M} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\mathbf{0} & \mathbf{I} & \mathbf{M} & \mathbf{I} & \mathbf{0} & \dots & \mathbf{0} \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{I} & \mathbf{M} & \mathbf{I} & \mathbf{0} \\
\mathbf{0} & \dots & \dots & \mathbf{0} & \mathbf{I} & \mathbf{M} & \mathbf{I} \\
\mathbf{0} & \dots & \dots & \dots & \mathbf{0} & \mathbf{I} & \mathbf{M}
\end{bmatrix}$$

Where **I** is an n-by-n identity matrix and **M** is an n-by-n matrix of the form:

$$M_{n\times n} = \begin{bmatrix} 4 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 4 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 4 & -1 & 0 \\ 0 & \dots & \dots & 0 & -1 & 4 & -1 \\ 0 & \dots & \dots & \dots & 0 & -1 & 4 \end{bmatrix}$$

For an interior grid of size 32-by-32, then matrix A is a 1024-by-1024 matrix. The solution vector **u** is:

$$\mathbf{u} = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1,32} & u_{21} & u_{22} & \dots & u_{2,32} & \dots & \dots & u_{32,1} & u_{32,2} & u_{32,3} & \dots & u_{32,32} \end{bmatrix}^T$$

The right-hand side vector **f** is:

$$\mathbf{f} = \begin{bmatrix} f_{11} & f_{12} & \dots & f_{1,32} & f_{21} & f_{22} & \dots & f_{2,32} & \dots & \dots & f_{32,1} & f_{32,2} & f_{32,3} & \dots & f_{32,32} \end{bmatrix}^T$$

This linear system can be solved using an iterative scheme such as a Krylov Subspace (KSP) Method. There are many KSP methods that can be applied to this problem, and our focus will be on the Generalized Minimal Residual Scheme (GMRES). GMRES was selected because it is applicable to a wide variety of matrices and hence is prevalent in numerical PDEs, so I would like to gain a deeper understanding of the runtimes involved. The reason GMRES is so common is that it does not require the matrix to be SPD (symmetric and positive definite), even though our matrix is indeed SPD. A preconditioner will be used in order to improve the convergence of the iterative method. The preconditioner employed for this problem is Geometric agglomerated algebraic multigrid (GAMG), using v-cycle, with SOR Chebyshev as the pre/post smoother. The goal of this project is to determine the number of iterations needed for the algorithm to converge for each grid size, determine what speedups can be obtained, and find out if speedups depend on the size of the interior grid.

Parallelization Strategy

To solve the 2D Poisson Problem stated above, the code will be written in C and the GMRES method available from PETSc will be used. I decided to learn PETSc for this assignment as it will be highly useful towards my research in Applied Math. PETSc supports MPI so it is convenient to use, although there is still a steep learning curve for a first time user. It provides very efficient parallel linear solvers and therefore is a good fit to solve this type of PDE problem. Moreover, PETSc has good routines that enable efficient parallel creation and assembly of matrix A as well as the right-hand side vector **f**. PETSc allows the user to define their desired structured 2D grid using the *DMDACreate2d()* routine. The arguments passed to this routine include the size of the interior grid, the x and y boundary conditions, the finite differencing stencil, and expected number of non-zero elements per row of the matrix. The boundary conditions are set to none, the stencil used is the five-point stencil that arises from centered differencing (see figure 1) and the number of non-zeros per row is set to 5, as there are at most 5 non-zeros in each row of matrix A. For each processor, PETSc will automatically portion off smaller grids to give to each process. The local grid size on a particular process depents on how many processors are being used as well as how PETSc decides to split up the global grid. For example, if the grid size is 32-by-32 and we use 2 processors, then each processors will get a 16-by-32 retangular grid. However, if we use 4 processors, then each processor gets a 16-by-16 square grid. The biggest advantage to this is that the user does not have to explicitly code how many rows or columns of a matrix each MPI process is going to deal with. This provides greater flexibility in terms of using any number of processes you wish without having to worry about how to divide up the work.

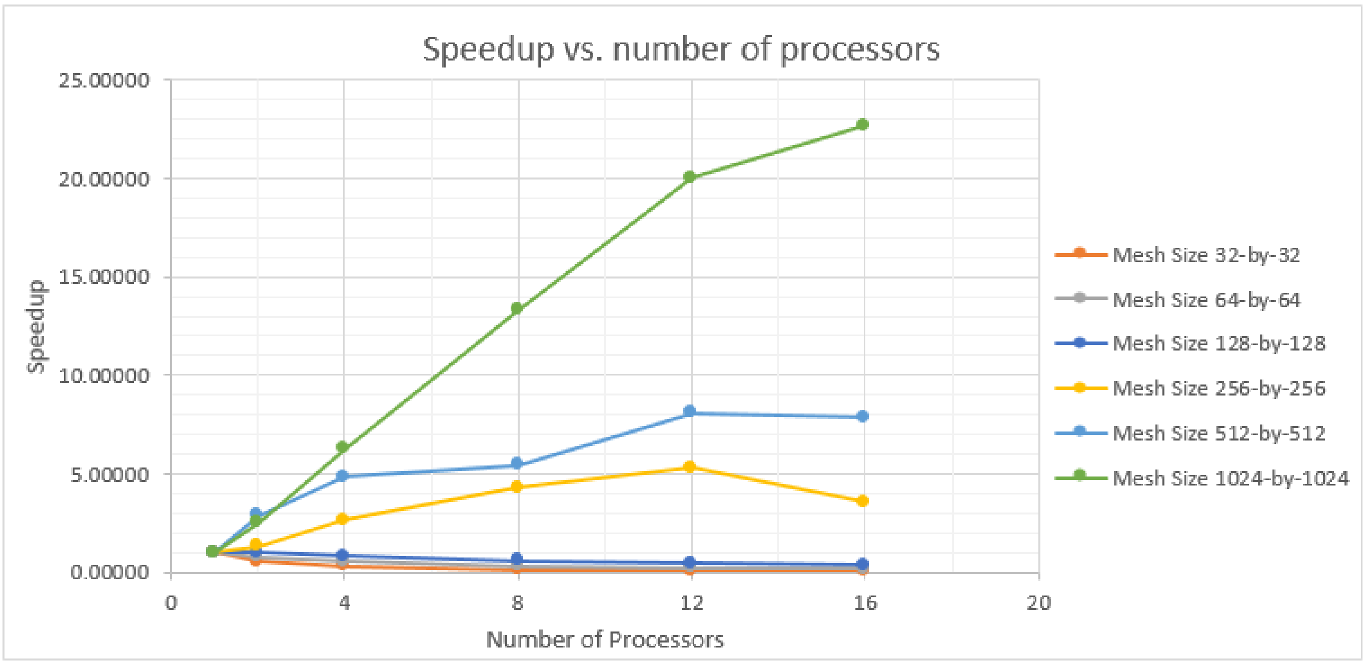
After setting up the grid, we can create matrix A and input elements into it. Since A is of tri-diagonal block structure, it is a sparse matrix and therefore can be stored in Compressed Sparse Row format. This can save a lot of memory if matrix preallocation is performed. Inputting matrix elements is accomplished by the use of *DMCreateMatrix()* and *MatSetValuesStencil()* routines. Due to having already defined the 5-point stencil on our structured 2D grid, the elements of the matrix can be easily inputted into their appropriate locations by taking advantage of the stencil. Therefore, the creation of the matrix is greatly simplified in PETSc. Setting up the right-hand side (RHS) vector is accomplished using the routine *DMCreateGlobalVector()* which sets up the vector in parallel so that each process stores a unique local portion. In order to automatically match each RHS vector element with its corresponding grid point using the indexing described in the previous section, the routine *DMDAVecGetArray()* is used. It returns a two-dimensional array whose elements can be set up using the right-hand side function. Then the array is passed back to PETSc using *DMDAVecRestoreArray()*so that it can be re-organized using the indexing scheme that PETSc uses. Hence, we don't have to worry about understanding PETSc indexing as these two routines do the work for us.

Once the elements of matrix A and vector **f** are assigned as described, they will be passed as arguments to the function *KSPSolve()*. This function also requires the type of iterative solver to be passed to it. I will use GMRES as the iterative method to solve the linear system. Geometric agglomerated algebraic multigrid (GAMG) preconditioning will also be applied so as to improve convergence. Once the Krylov Subspace Solver context and Preconditioner context is set-up, the KSP solver will solve the linear system.

Numerical Experiments

The following results were obtained from the speed-up tests. (Note that the row header Grid Size really refers to interior grid size).

	Grid Size: 32-by-32			Grid Size: 64-by-64		
Processors	**No. of Iterations**	**Runtime (s)**	**Speedup**	**No. of Iterations**	**Runtime (s)**	**Speedup**
1	4	0.07125	1.00000	5	0.266820	1.000000
2	4	0.13770	0.51744	5	0.347509	0.767808
4	4	0.22241	0.32036	5	0.507407	0.525850
8	4	0.68944	0.10335	5	0.864200	0.308748
12	4	1.04850	0.06796	5	1.306177	0.204276
16	4	1.44952	0.04916	5	1.708525	0.156170
	Grid Size: 128-by-128			Grid Size: 256-by-256		
Processors	**No. of Iterations**	**Runtime (s)**	**Speedup**	**No. of Iterations**	**Runtime (s)**	**Speedup**
1	5	1.16392	1.00000	6	5.75381	1.00000
2	5	1.16173	1.00189	6	4.37948	1.31381
4	5	1.47302	0.79016	6	2.17939	2.64009
8	5	2.08490	0.55826	6	1.34699	4.27160
12	5	2.75834	0.42197	6	1.08353	5.31023
16	5	3.28142	0.35470	6	1.62331	3.54449
	Grid Size: 512-by-512			Grid Size: 1024-by-1024		
Processors	**No. of Iterations**	**Runtime (s)**	**Speedup**	**No. of Iterations**	**Runtime (s)**	**Speedup**
1	7	111.45968	1.00000	8	1644.56343	1.00000
2	7	39.09347	2.85111	8	662.95362	2.48066
4	7	22.96242	4.85400	8	262.49564	6.26511
8	7	20.60144	5.41028	8	123.84751	13.27894
12	7	13.77450	8.09174	8	82.08394	20.03514
16	7	14.17085	7.86542	8	72.55268	22.66716



Furthermore, it is interesting to see what the solution of the PDE looks like. The solution vector obtained by solving a system arising from a 64-by-64 interior grid was imported into MATLAB for visualization and the plot is shown below:

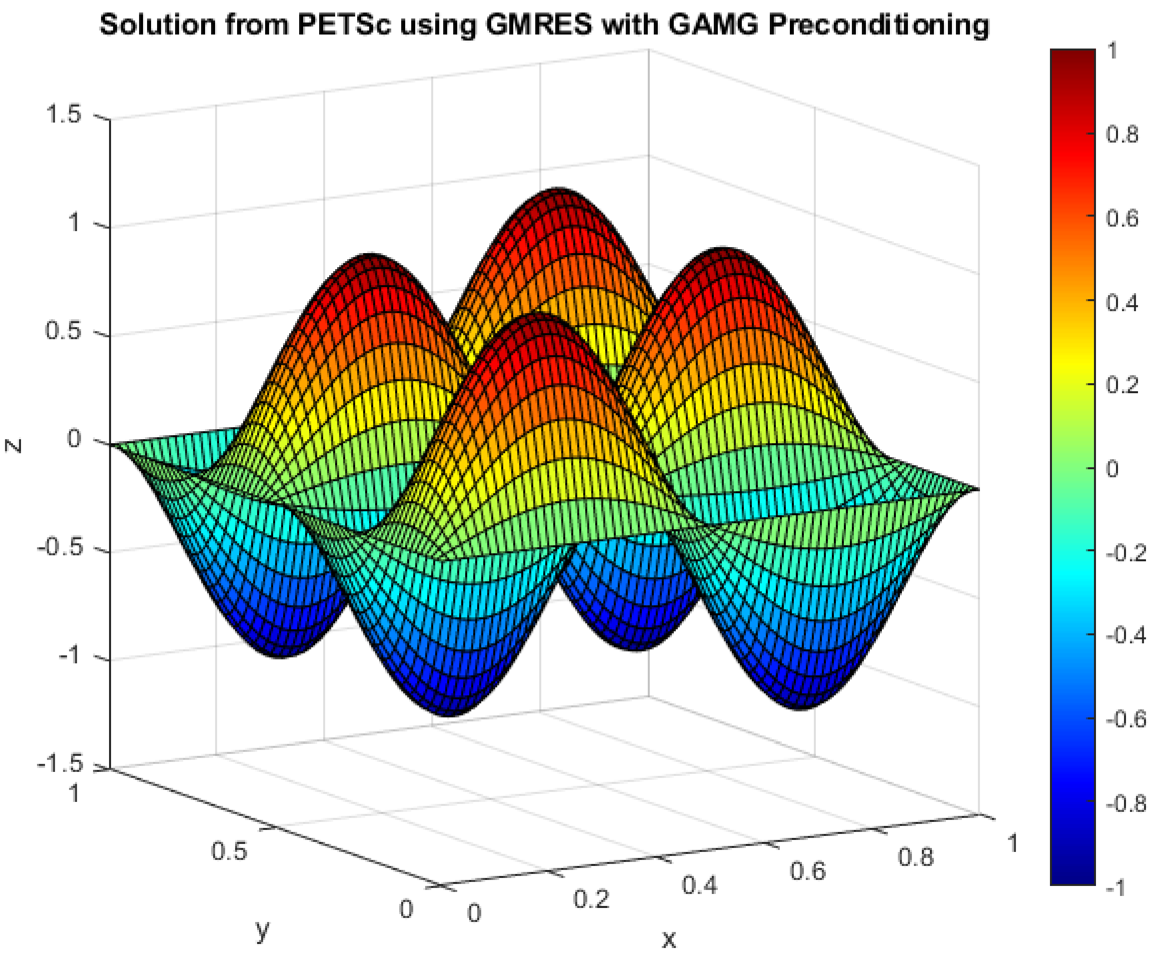


Figure 2: Visualization of the solution to the PDE using MATLAB.

Discussion

The results from the speed-up study show that there is pretty good speed-up when the grid size (and hence matrix size) is sufficiently large. According to the plot, it is clear that there is no speed-up for interior grid sizes 32-by-32, 64-by-64 and 128-by-128. In fact, the runtimes are increasing when more processors are being used. This can be explained by the fact that communication overhead costs due to a larger number of processors take more time than the iterations themselves. Furthermore, since we are using multigrid preconditioning, then time is taken per solver iteration to set up the coarse and fine grids as well as the restriction and interpolation operators needed. All this additional work takes some time to be completed, so when the problem is small enough, it is usually not worth it to employ such a powerful tool.

When the grid size is increased to 256-by-256, that is, when the matrix size is 65536-by-65536, then we can observe some speed-up. The runtime decreases as more processors are employed. When increasing the number of processors from 12 to 16, the runtime takes a little longer and hence there is a drop in speed-up. This is expected since using 16 processors means we are using some processors from another node so message passing between them takes a little longer. There is even better increase in speed-up when the grid size is 512-by-512. For this grid, we are solving a linear system in which the matrix is of size 262144-by-262144. Although the speed-up is not linear, there is still a considerable decrease in runtimes as more processors are used. Again, going from 12 to 16 processors results in a slight drop in speed-up because of increased communication cost between cores that are located in different compute nodes. The best speed-up observed is for a grid size of 1024-by-1024. Here the matrix size is 1,048,576-by-1,048,576. Hence we are solving for over 1 million unknowns. The plot shows that the speed-up increases in a linear fashion as number of processors is increased. Between 1 to 12 processors, the speed-up is extremely close to being perfectly linear. When going from 12 to 16 processors, the speed-up continues to increase but not at the same rate as before. In this case, the communication cost due to using 16 processors causes the program to take longer to run, but overall there is still an improvement in runtimes as compared to when 12 processors as used. Therefore, it can be concluded that the program enjoys very good speed-up for problems that are quite large, such as when an extremely fine grid is used to discretize the PDE.

The use of preconditioning ensures that the solution converges for all problem sizes. From the table, the number of iterations to reach an acceptable solution (i.e. when the residual is within a tolerance of 1e-07) is always less than 10 for all problem sizes. This shows that the GMRES solver coupled with GAMG preconditioner is quite robust. Each time the problem size is doubled, the number of iterations needed to converge to a solution only increases by 1.

For future work, it would be useful to dive deeper into understanding the details behind the GAMG method. In order to get results even faster, the algorithm could be fine tuned by setting options for the solver and the preconditioner. For example, PETSc allows the user to choose the type of GAMG (smoothed aggregate (default), geometric or classical), to define repartitioning of degrees of freedom across coarse grid as they are determined, to set maximum number of equations to aim for on coarsest grid, to define the number of smoothing steps to use with smooth aggregation, to configure the pre and post smoothers separately, to set the type (v or w cycle) and number of multigrid levels to use etc [Ref. 1]. There are so many choices to choose from that I'm sure the runtimes could be reduced with the right combination of options. However, this would require a deeper mathematical understanding of the multigrid algorithm, so I will continue to focus on this because it will directly help with my research.

The greatest challenge of this project was to learn the PETSc routines required in order to set-up the problem properly. I wrote two programs, one that did not use sophisticated objects and data structures, and one that uses sophisticated topologically structured grids that interface with distributed arrays. The latter of the two was selected since it works very well with multigrid. As a beginner, the time to code was quite significant, however the knowledge and experienced gained at the end largely offsets the setbacks at the beginning. This project taught me a lot about how parallel processing can be applicable in the real world for solving extremely large problems arising from complicated PDEs.

References

[1] PETSc Users Manual, Argonne National Laboratory, 2020, ANL-95/11 - Revision 3.13, https://www.mcs.anl.gov/petsc

[2] Leveque, Randall J. Finite Difference Methods for Ordinary and Partial Differential Equations, SIAM, 2007.

[3] Numerical Linear Algebra with Applications, William Ford, 2015.