PARALLELZATION OF THE MULTIGRID METHOD USING OPENMP AND MPI

Anish Anand Pophale
CH20B012

Shasank G
CH20B097

ABSTRACT

Tackling engineering problems with a computational approach often involves solving partial differential equations numerically. Solutions to partial differential equations using finite difference methods generates a system of linear equations, which is generally solved using iterative methods such as the Gauss Seidel or Jacobi methods. For accuracy, fine meshes are required to discretize the domain leading to a larger system of equations to be solved. For such iterative methods, it is observed that the error initially reduces rapidly but the rate of convergence decreases as the error reduces. The error field over the domain can be thought of as having high and low frequency components similar to a Fourier series. The high frequency error components die down faster, but the low frequency components slow down the convergence.

The Multigrid method makes use of this property to accelerate the convergence of such iterative methods by using multiple meshes of different sizes. The main idea behind this is that a low frequency error on a fine mesh is a high frequency error on a coarser mesh which can be transferred over and reduced rapidly followed by transferring the solution back to the fine mesh. This ideology can be applied to any of the iterative methods like Gauss Seidel, Jacobi, Gradient descent, etc. which are known as smoothers in the context of multigrid method.

As a part of this project, we will parallelize the multigrid method and apply it for the solution of a 2D Poisson equation using the finite difference method. In the multigrid method, the step of transferring the solution or residuals over various meshes is inherently sequential while the iterations of the smoother at each mesh level can be parallelized which will be our primary focus. Other methods of parallelization modify the basic algorithm to eliminate the sequential nature of this process. We will explore different types of smoothers namely the Jacobi and Red Black Gauss Seidel methods and use them for a two-level V cycle multigrid. This would be implemented in both OpenMP and MPI. The performances of the serial and parallel versions of the multigrid method will be compared. These results will also be compared with the standard Jacobi and Red Black Gauss Seidel methods without using multigrid, implemented in both serial and parallel.

INTRODUCTION

For accurate solutions of partial differential equations using finite difference method, a large mesh size is required leading to a larger system of equations to be solved. Traditional iterative methods such as the Jacobi and Gauss Seidel method require a large number of iterations in such problems. Monitoring the error at each iteration leads to the observation that the rate of convergence decreases as the error decreases, making these methods slow to converge. Analyzing the error field calculated after each iteration leads to a very interesting explanation for the above observation. If the error field is decomposed as a Fourier series, it can be shown that the high frequency components of the error are eliminated rapidly, but the low frequency components need a greater number of iterations to reduce or smoothen out. These low frequency components are responsible for the reduced rate of convergence of such iterative methods.

The multigrid method accelerates the converge of these iterative methods by exploiting this above property. The key idea in the multigrid method is that low frequency error components on a fine mesh, have a high frequency on a coarse mesh which can be eliminated quickly. Hence, in multigrid method, the errors are transferred from a fine mesh to a coarser mesh where they can be smoothened out quickly. As the final solution is required on a fine mesh for accuracy, the solution from the coarse mesh is transferred back to the fine mesh in the final step. This methodology can be applied to any of the iterative solvers which are know as smoothers in this context as they are used to smoothen out the high frequency error components of each mesh level.

Based on the number of levels of the coarse and fine meshes as well as the patterns in which the solution data is transferred across various meshes, there are 3 types of multigrid cycles which are the V cycle, W cycle and the F cycle as shown in Fig 1. In this project, we implement a 2 level V cycle multigrid, which uses only two mesh levels, a fine mesh and a coarse mesh. The detailed algorithm for the implementation of this 2 level V cycle multigrid is presented in the next section. Apart from this there is another form of classification of multigrid methods, which is the algebraic multigrid (AMG) and the geometric multigrid (GMG). The geometric multigrid method requires information about the meshes or problem geometry at each level while the algebraic multigrid method

does not need any information about the problem geometry and can be used for complicated cases such as unstructured meshes. Both these methods still use the same philosophy of transferring the errors between a hierarchy of grid levels

The multigrid method in itself improves the convergence rate of iterative methods. In this study, the main objective is to examine if the speed of this method can be further improved with parallelization. We will use a shared memory model - OpenMP as well as a distributed memory model – MPI. The performance of both the paradigms will also be compared.

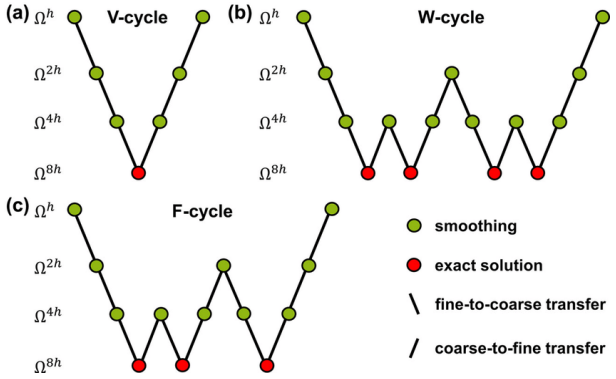

Fig 1: Schematic of different multigrid cycles

PROBLEM DEFINITION

We consider the case of a 2D Poisson equation of the following form on the domain x = [0,1], y = [0,1] with the given Dirichlet boundary conditions.

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 5e^x e^{-2y}$$

$$\phi_{exact}(x,y) = e^x e^{-2y}$$

$$\phi(0,y) = e^{-2y}, \phi(1,y) = e^{1-2y}$$

$$\phi(x,0) = e^x, \phi(x,1) = e^{x-2}$$

The discretized equation using a second order central difference using a equidistant mesh with grid size Δ in both directions is given as follows:

$$\phi(i,j) = \frac{1}{4}[\phi(i+1,j) + \phi(i-1,j) + \phi(i,j+1) + \phi(i,j-1) - \Delta^2 5e^x e^{-2y}]$$

Fig 3 shows the exact solution for the equation. The convergence criteria is defined based on the norm of the exact solution as given below,

$$\frac{||\phi_{exact} - \phi||}{||\phi_{exact}||} < 1e^{-4}$$

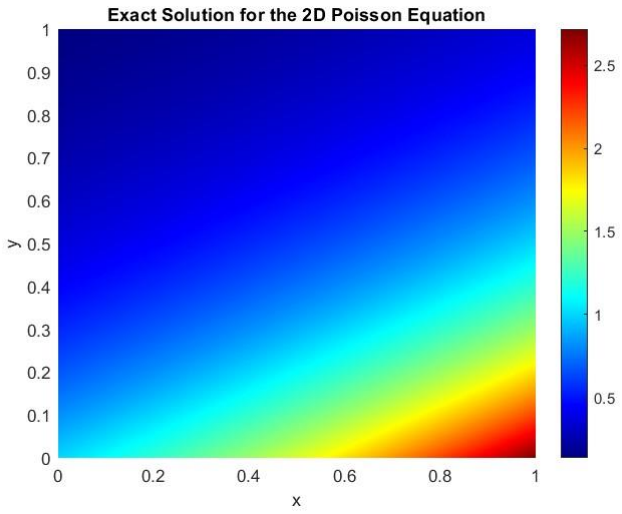

Fig 2: Exact Solution to the given 2D Poisson equation

NOMENCLATURE FOR THE GENERAL ALGORITHM

- Grid Spacing (Fine Mesh) = h
- Grid Spacing (Coarse Mesh) = 2h
- Variables on the fine mesh are denoted by A^h
- Variables on coarse mesh are denoted by A^{2h}
- System of linear equations: Au = f
- Approximate solution = v
- Grid/Mesh - Ω
- Residual = r
- Correction = e
- Interpolation/ Restriction Matrix = I

V CYCLE MULTIGRID ALGORITHM

In this section, we discuss all the steps of the geometric multigrid method using 2 levels for a V cycle multigrid, the general algorithm for the same is given in Fig 3. We further discuss each step, in detail and how is it applied to the discretized 2D Poisson equation.

Two-Grid Correction Scheme

$$\mathbf{v}^h \leftarrow MG(\mathbf{v}^h, \mathbf{f}^h).$$

- Relax ν_1 times on $A^h\mathbf{u}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.
- Compute the fine-grid residual $\mathbf{r}^h = \mathbf{f}^h - A^h\mathbf{v}^h$ and restrict it to the coarse grid by $\mathbf{r}^{2h} = I_h^{2h}\mathbf{r}^h$.
- Solve $A^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ on Ω^{2h}.
- Interpolate the coarse-grid error to the fine grid by $\mathbf{e}^h = I_{2h}^h\mathbf{e}^{2h}$ and correct the fine-grid approximation by $\mathbf{v}^h \leftarrow \mathbf{v}^h + \mathbf{e}^h$.
- Relax ν_2 times on $A^h\mathbf{u}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.

Fig 3: Two level V cycle multigrid algorithm

- **Step 1: Smoothing / Relaxation of the equations on the fine mesh**:

In this step, a few iterations of the smoother are applied on the fine mesh. After this step, we get v^h, which is an approximate solution to the set of equations obtained after a few iterations of the smoother. For the Poisson equation, we do not use an explicit A matrix as shown in the algorithm but we can directly use the discretized equation as it involves only 4 neighbors. The update equation using the Jacobi iterative method is given below where n is the iteration number. The discretized equation changes with the mesh size as the grid spacing Δ appears in the equation.

$$\phi^{n+1}(i,j) = \frac{1}{4}[\phi^n(i+1,j) + \phi^n(i-1,j) + \phi^n(i,j+1) + \phi^n(i,j-1) - \Delta^2 5e^x e^{-2y}]$$

- **Step 2: Calculating the fine grid residual**:

In this step, we calculate r^h which is the fine grid residual defined as f^h - $A^h v^h$. Again, for the Poisson equation, we can calculate the residual using the discretized equation using the expression below,

$$r = 5e^x e^{-2y} - \frac{[\phi(i+1,j) + \phi(i-1,j) + \phi(i,j+1) + \phi(i,j-1) - 4\phi(i,j)]}{\Delta^2}$$

- **Step 3: Restriction of residual from fine mesh to coarse mesh**:

Here we calculate r^{2h} from r^h using a restriction matrix I which means we are transferring the residual from the fine mesh to the coarse mesh. This can also be implemented explicitly by taking a weighted average of all the neighbors.

$$\begin{aligned} r^{2h} =& \frac{1}{4}r^h(i,j) + \frac{1}{8}[r^h(i+1,j) + r^h(i-1,j) \\ &+ r^h(i,j-1) + r^h(i,j+1)] + \frac{1}{16}[r^h(i+1,j+1) \\ &+ r^h(i-1,j+1) + r^h(i+1,j-1) + r^h(i-1,j-1)] \end{aligned}$$

- **Step 4: Solving the correction equation on the coarse mesh**:

Consider the equation $r^{2h} = f^{2h} – A^{2h}v^{2h}$, if we assume that the solution obtained in the next iteration v' is the exact solution, we get $0 = f^{2h} – A^{2h}v^{2h}$'. Subtracting the equations, we get $A^{2h}e^{2h} = r^{2h}$ where e^{2h} is the correction for v calculated on the coarse mesh. This equation can be solved iteratively for a few iterations to obtain the correction e, as we know the A matrix for coarse mesh and r^{2h} is calculated in the previous step.

For the discrete 2D Poisson equation, again we do not use the A matrix explicitly and depending on the iterative method used, an appropriate update equation can be derived as done in first step. Using the Jacobi method, the explicit update equation for e is given as follows, where Δ is the coarse mesh size and n is the iteration number here.

$$e^{n+1}(i,j) = \frac{1}{4}[e^n(i+1,j) + e^n(i-1,j) + e^n(i,j+1) + e^n(i,j-1) - \Delta^2 r(i,j)]$$

- **Step 5: Interpolating the correction to the fine mesh**:

Now, we transfer the correction e^{2h} to the fine mesh so that it becomes e^h. For this the same matrix I can again be used as given in the algorithm or an explicit formula given below can be used. Once e^h is calculated, it is added to v^h to get the improved value of the solution.

$$e^h(2i,2j) = e^{2h}(i,j)$$

Horizontal Sweep:

$$e^h(2i,2j+1) = \frac{1}{2}[e^{2h}(i,j+1) + e^{2h}(i,j)]$$

Vertical Sweep:

$$e^h(2i+1,2j) = \frac{1}{2}[e^{2h}(i+1,j) + e^{2h}(i,j)]$$

4 point interpolation:

$$e^h(2i+1,2j+1) = \frac{1}{4}[e^{2h}(i,j) + e^{2h}(i,j+1) + e^{2h}(i+1,j) + e^{2h}(i+1,j+1)]$$

- **Step 6: Go back to step 1 and iterate till convergence**

Now as we have obtained an improved value of the solution, we check for convergence. If the convergence criteria is not satisfied, we go back to step 1 and iterate till convergence is achieved. Steps 1 to 5 correspond to one full iteration of the multigrid method. So, each iteration of the multigrid method involves a fixed number of smoothing iterations on the fine and the coarse meshes.

CODE IMPLEMENTATION AND PARALLELIZATION STRATEGY

In this section, we present the details of the implementation of the multigrid method and the parallelization strategy using both OpenMP and MPI.

For ease of implementation, we take the number of grid points on the fine mesh as $2^n + 1$ and on the coarse mesh as $2^{n-1} + 1$ which makes the restriction and interpolation steps easier to implement in the code. We set the number of smoothing iterations to 5 for both Step 1 and Step 4. The number of iterations performed on each mesh can also be optimized to minimize the overall computational expense for convergence but this is out of the scope of this study. Instead, we use only 5 iterations in each of the smoothing step. Hence, in each of the multigrid iteration we have 5 smoothing iterations of the fine mesh and 5 iterations on the coarse mesh. This corresponds to 10 smoothing iterations per outer iteration of the multigrid method. The C codes attached with the report include the following:

Jacobi Method:

- Jacobi_Serial.c - Serial Jacobi Method
- Jacobi_OMP.c - Parallel Jacobi Method using OpenMP
- Jacobi_MPI.c - Parallel Jacobi Method using MPI
- JacobiMG_Serial.c - Serial Jacobi Method with Multigrid

- JacobiMG_OMP.c - Parallel Jacobi Method with Multigrid using OpenMP
- JacobiMG_MPI.c - Parallel Jacobi Method with Multigrid using MPI

Red Black Gauss Seidel Method:

- RedBlack_Serial.c - Serial Red Black Method
- RedBlack_OMP.c - Parallel Red Black Method using OpenMP
- RedBlack_MPI.c - Parallel Red Black Method using MPI
- RedBlackMG_Serial.c - Serial Red Black Method with Multigrid
- RedBlackMG_OMP.c - Parallel Red Black Method with Multigrid using OpenMP
- RedBlackMG_MPI.c - Parallel Red Black Method with Multigrid using MPI

The algorithmic steps discussed in the previous section are also applicable for any type of multigrid cycle with any number of steps. These set of steps are generally implemented recursively if there are a greater number of mesh levels, but we do not require this as we only perform a 2-level multicycle. These steps of smoothing, restriction and interpolations are to be performed sequentially by the nature of the algorithm. For our parallel implementation this is not altered as we parallelize each of the steps independently. Some implementations of the multigrid method in parallel involve modifying the algorithm to eliminate this inherent sequential nature but we will not be focusing on this type of approach.

Parallelization using OpenMP:

The parallel implementation of all the serial codes using OpenMP is straightforward. We use the Red Black Gauss Seidel method instead of the normal Gauss Seidel method as it has very limited parallelization potential due to the inherent data dependency on the updated values of the current iterations. All the loops which can be parallelized are run in parallel using the #pragma omp parallel for directive. Specifically, parallelizing the smoothing, restriction and interpolating tasks provides the majority of the speedup. Other less calculation intensive loops such as the calculations for error, exact solution, initializing, etc. are also parallelized. The code is written in a way to minimize the number of times threads are forked and joined inside the main iteration loop so that the threads are only forked once per outer iteration of the main iterative loop.

Parallelization using MPI:

Parallelization using distributed memory programming with MPI gets a bit complicated. We use a row block decomposition of the domain for both the fine and the coarse meshes. For each of the steps of smoothing, restriction and interpolation, we need to use ghost or virtual nodes at the top and the bottom boundaries of the row decomposed domain. Due to the large number of send and receives that occur at every iteration of the outer loop, the benefit of the MPI parallelization is evident for only larger problem sizes where the communication overhead in negligible compared to the benefit of parallelizing the computations, although the amount of data sent and received also increases with the problem size.

RESULTS AND DISCUSSION

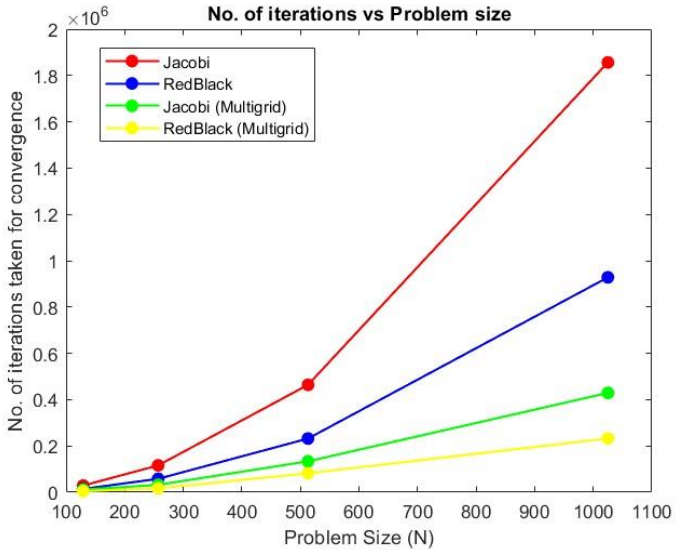

Fig 4: Comparison of Number of iterations vs Problem Size

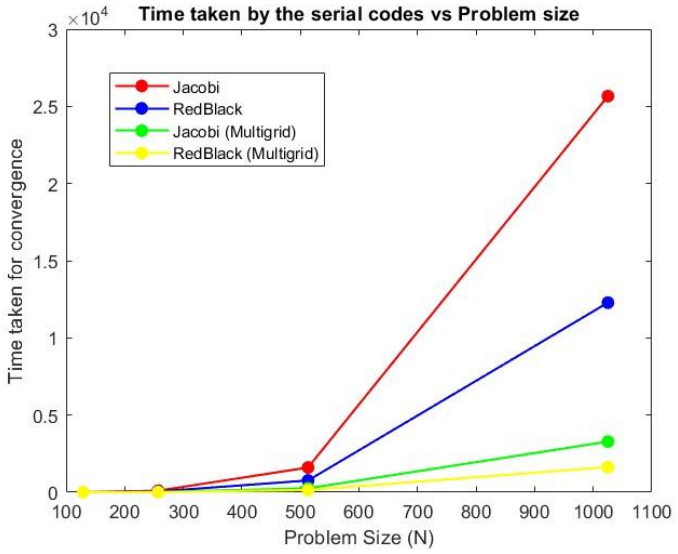

Fig 5: Comparison of Time taken in serial vs Problem Size

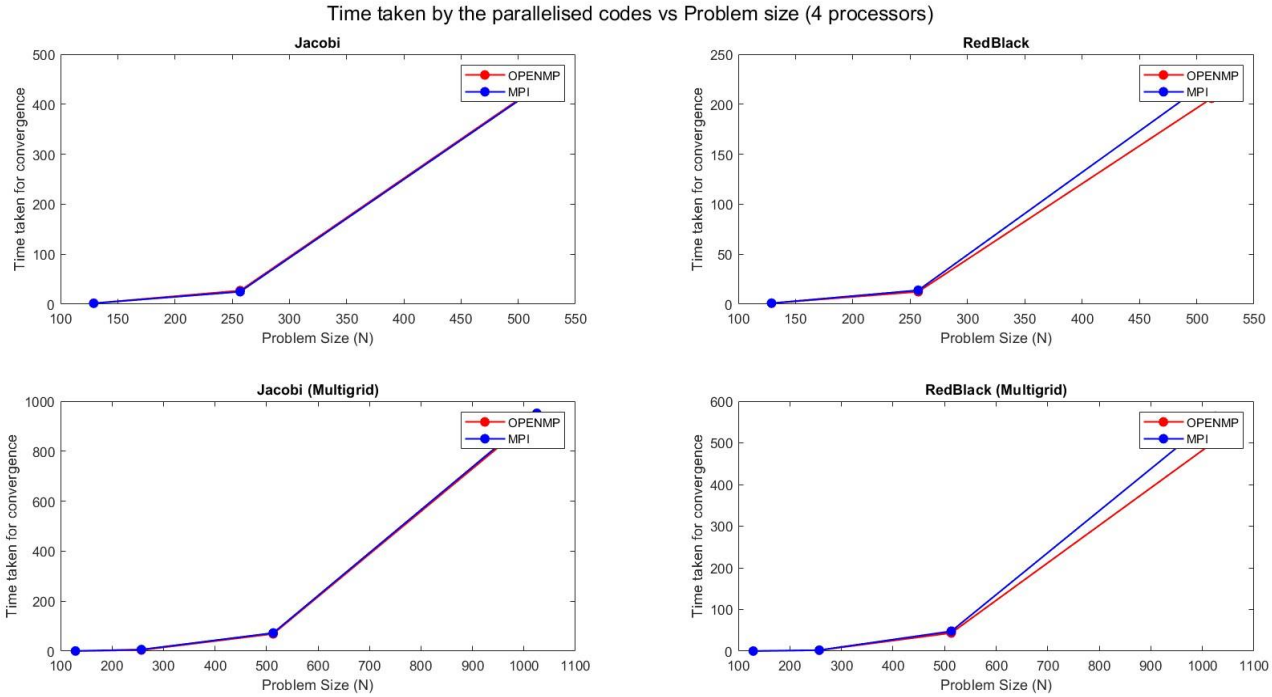

Fig 6: Comparison of Time taken (seconds) vs Problem Size for parallel codes using 4 processors

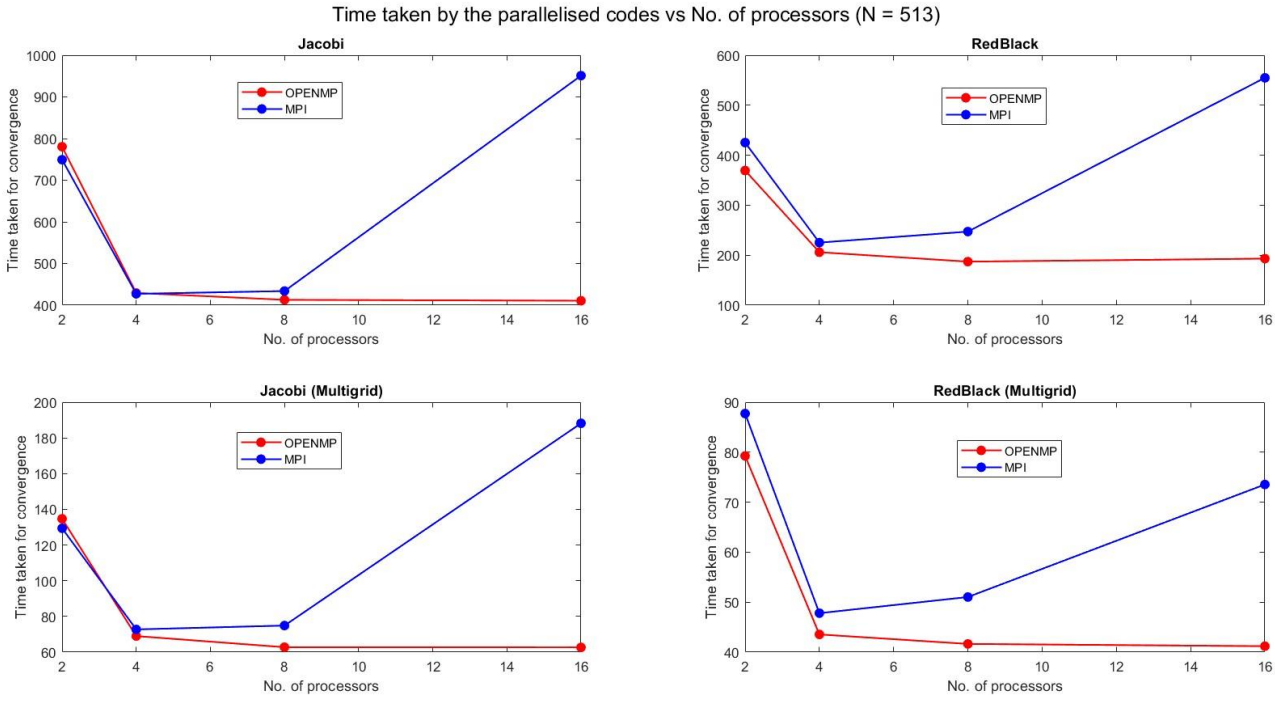

Fig 7: Comparison of Time taken (seconds) vs Number of Processors for parallel codes using a problem size of 513

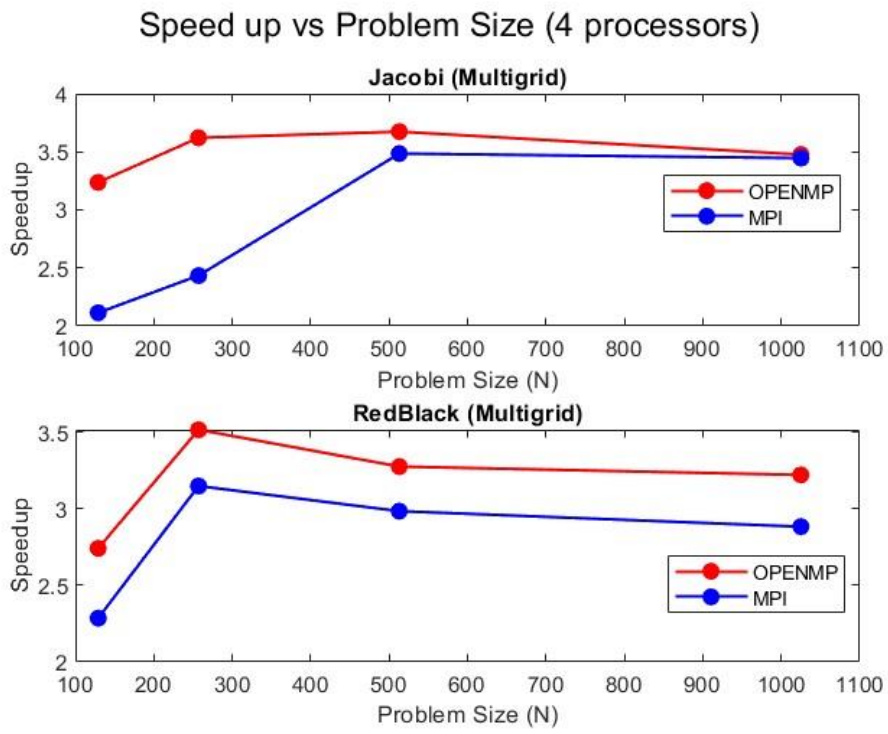

Fig 8: Speed up vs Problem Size for parallel multigrid methods using 4 processors

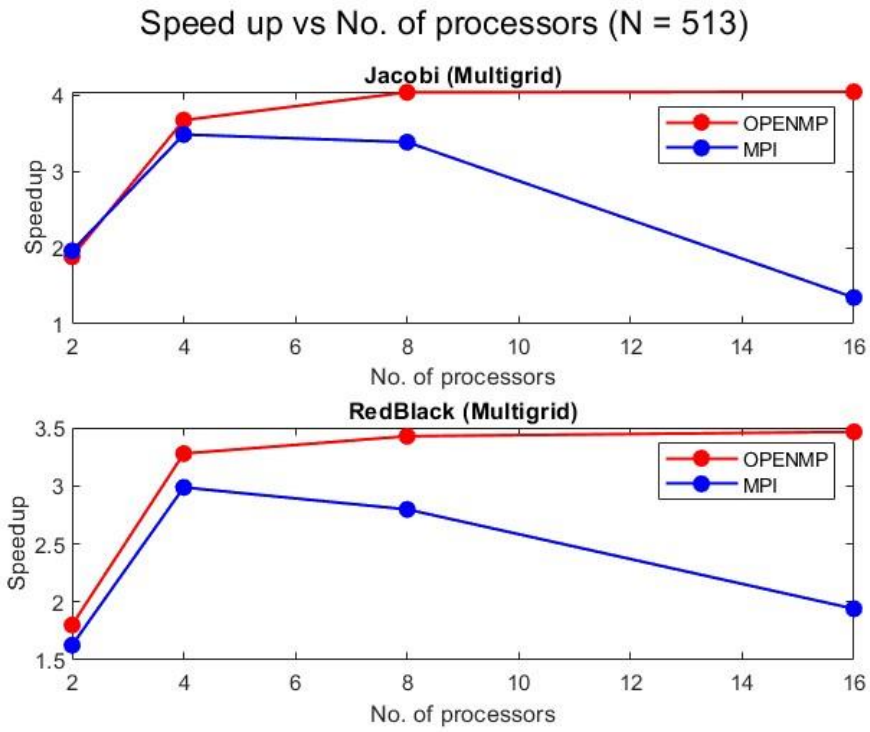

Fig 9: Speed up vs Number of processors for parallel multigrid methods using a problem size of 513

Figures 4 and 5 compare the performances of the serial Jacobi and Red Black Gauss Seidel iterative methods with and without using the multigrid method. In this section if the grid size used to discretize the domain is N x N, we call it as a problem size of N. From literature it is known that traditional iterative methods have a time complexity of $O(N^2)$ while the multigrid method is able to provide a linear trend with a time complexity of almost O(N). This can also be seen from our data in the figures, as the traditional Jacobi and Red Black method show a nearly non-linear trend while the multigrid methods show a linear trend for time taken with increasing problem size. The number of iterations as well as the computational time decreases drastically for the multigrid methods compared to the standard methods which shows the main advantage of this technique. Furthermore, the number of iterations in the multigrid method are not all on the same mesh size, half of them are on a coarser mesh which requires less computational time which makes these methods extremely fast. All the data of run time as well as speed up for all methods discussed, in serial as well as parallel in both OpenMP and MPI have been attached in the supplementary data spreadsheet along with this report.

For large problem sizes such as N = 1025, the standard iterative methods take very long compared to the multigrid method as the time taken grows non linearly. For the Jacobi method wit N = 1025, the time taken is close to 7 hours while for the Red Black method it is around 3 hours while the multigrid methods with these 2 smoothers need less than 1 hour. Hence, we do not consider the case of N = 1025 for the parallel versions for the standard methods as the time required is too high compared to the multigrid methods. In general, it is expected that the Red Black Gauss Seidel methods performs better than the Jacobi method as it uses updated values from the same iteration while Jacobi method uses the values from the previous iteration. As expected, we observe the following trend in the performance of these methods when run serially: Red Black Multigrid > Jacobi Multigrid > Red Black > Jacobi

Figures 6 and 7 compare the performances of the parallelized versions of the traditional iterative methods with the multigrid methods using both MPI and OpenMP. As justified above, we have not included the case of N = 1025 for the standard methods in these plots. In Fig 6, we compare the performance of these methods for a fixed number of processors while varying the problem size. As expected, for all the method, the time increases as we increase the problem size. The time taken using the MPI and OpenMP parallelization is almost the same for 4 processors. But this is not true in general, for larger number of processors, we see differences in the performance of these 2 parallel implementations which we will discuss in the next section. For this comparison as well, we observe the same trend in the performances as before. Another interesting observation from the data is that the serial multigrid method with both Red Black and Jacobi is still faster than the parallelized standard methods using 4 threads for the largest problem size consider which is N = 513.

In Fig 7, we compare the run time in seconds for a fixed problem size of 513 with varying number of processors used. Here as well we observe the same trend in terms of the run time as the parallel Red Black multigrid method is the fastest. In these plots we observe a difference in the performances of the OpenMP and MPI parallelization of the same code. While the OpenMP parallelization gives a better performance as the number of threads is increased, the improvement in the performances stagnates at 16 threads. But for MPI, the performance improves till 8 processors used, but beyond that, for 16 processors, the performance s worse and more time is required to run the code. This may be associated with a larger communication overhead in MPI compared to the joining and forking overhead in OpenMP which is the key difference in the distributed and shared memory paradigms. Apart from this, comparing the overall performance of these two, we see that OpenMP give better performance than MPI.

Figures 8 and 9 compare the speed up for the multigrid methods with varying problem sizes as well as number of processors. We see that the speed up obtained is better as the problem size increases which is expected as the parallelization of the computation outweighs the overheads of forking threads and communication between processors as the problem size increase. In both the figures, we can also see that the overall speedup obtained is better in the multigrid method with the Jacobi smoother than the red Black smoother. But this does not mean that the Jacobi multigrid is faster in terms of run time, the results for which is actually the opposite. This just mean that the scope or potential for parallelization is more in the Jacobi multigrid than the Red Black multigrid or that the overheads associated with the parallelization of Jacobi multigrid are lower than that for the Red Black. From Fig 9, we again the see difference between OpenMP and MPI as discussed previously where for a fixed problem size, MPI gives worse performance for more than 8 processors, but for OpenMP the performance does not get worse but remains constant.

CONCLUSIONS

As a part of this work, we looked at 4 different methods which are used to solve systems of linear equations. These methods are the standard Jacobi, Red Black Gauss Seidel, Multigrid with Jacobi smoother and Multigrid with Red black Gauss Seidel smoother. Further, the benefits of parallelizing these methods to reduce the computational time were examined. The system of equations which arise from the finite difference discretization of the 2D Poisson equation was taken as an example to demonstrate the performance of these methods. The Multigrid method with the Red Black smoother

was found to be the fastest among all the methods. Along with this the OpenMP parallelization of this method provides the best performance when compared to MPI. For OpenMP, 16 threads give the best performance while for MPI, 8 processors provide the best speedup. As we have established that the Red Black multigrid method gives the best performance, further work can include comparing different cycles in the multigrid method, adding multiple layers or even optimizing the number of smoothing iterations of each mesh level. Another interesting extension could be the implementation of this using OpenACC to understand if GPU acceleration can provide even better speed up compared to what was obtained in this study.

ACKNOWLEDGEMENTS

Thanks to Prof. Kameswararao Anupindi for his valuable guidance and support throughout the duration of the course.

SUPPLEMENTARY INFORMATION

Along with the codes submitted for each of the methods as discussed in the report, a spreadsheet containing all the data collected for the run time, speed up and number of iterations taken by each of the methods in serial as well as parallel has been attached to the report.

REFERENCES

[1] Mazumder, S. (2015). *Numerical methods for partial differential equations: Finite Difference and Finite Volume Methods*. Academic Press.

[2] Strang, G. (2006). *Multigrid methods*. https://math.mit.edu/classes/18.086/2006/am63.pdf

[3] *CS267: Notes for Lecture 17, Mar 12 1996*. (n.d.). https://people.eecs.berkeley.edu/~demmel/cs267/lecture25/lecture25.html