An Optimized Finite Difference Formulation of the 2D Lid-Driven Cavity Flow Problem

Declan Murphy
MSc. in High Performance Computing
Trinity Centre for High Performance Computing

September 2014

Acknowledgements

First and foremost I would like to thank my supervisor at TCHPC, Dermot Frost, without whom I would still be trying in vain to solve the MultiGrid on Lonsdale.

I would like to thank the Trinity Centre for High Performance Computing for everything throughout the last 12 months of this MSc. program.

All calculations were performed on the Lonsdale and Kelvin clusters maintained by the Trinity Centre for High Performance Computing. The Lonsdale cluster was funded through grants from Science Foundation Ireland. The Kelvin cluster was funded through grants from the Higher Education Authority, through its PRTLI program.

My mother, Bríd, for making me dinners while I lived in front of the laptop like a goblin.

My dog and best mate, Shadow, for consistently giving educated feedback in our many conversations about parallelisation techniques. And for being so resilient.

Contents

List of Figures

1 Introduction & Aims

Computer parallelism has been employed for many years, mainly in high performance computing, but interest in it has been growing at an accelerated rate in recent times. This growth in interest is partially due to the physical constraints preventing further frequency scaling, including the increased power consumption that comes with increasing clock speeds. As power consumption (and consequently heat generation) by processors has become a concern, parallel computing has become the dominant paradigm in computer architecture - mainly in the form of multi-core processors. In the ideal scenario, solving a problem with n parallel cores will take one nth the time as solving in serial. However, scaling is never ideal. Parallelisation brings with it additional temporal overheads including inter-node communications, intra-node thread initialisation and more. High Performance Computing is the realm of minimising these negative overheads and approaching the ideal scenario.

The field of Computational Fluid Dynamics (CFD) is an increasingly relevant and popular field for computer graphics and CGI, realistically simulating liquids, smoke, explosions and other related phenomena. It is also a computationally expensive field and offers an ideal scenario to employ High Performance Computing tools and resources. Multiple opportunities arise not only for optimization through parallelization methods, but optimization through choice of algorithm. The aim of this project will be to implement parallelization techniques and to select algorithms to optimize fluid flow problems.

The benchmark problem for all Computational Fluid Dynamics software packages is the Lid-Driven Cavity Flow problem. In two dimensions, this problem consists of a rectangular problem area with 3 rigid walls and one free wall which moves with constant velocity, causing the fluid inside to move in a circular flow. For high Reynolds number flows, where motion is governed by inertial forces, fluid flow can become turbulent and chaotic so significant mesh resolution will be required in parts of the problem domain to deal with any discontinuities that arise. A naive solver might apply an ultra-fine resolution over the entire domain but this would result in vastly increased computation times, not to mention the inefficiencies that arise from the fact that most of the domain may not require such resolution. In scenarios such as these, it is near to impossible to predict in advance where finer resolution will be needed. This type of problem presents the ideal opportunity to use much more elegant and computationally efficient solutions such as MultiGrid methods or non-uniform (and in the case of an unsteady solution, adaptive) mesh refinement.

In this project, the aim is to create a numerically accurate, efficient, stable, and scalable solver for the unsteady 2D Lid-Driven Cavity Flow problem with the ultimate aim of creating a high performance software suitable for a wider range of 2-dimensional CFD problems. The problem domain is meshed and the relevant equations discretized under a Finite Difference formulation. A high performance result takes second priority only to a physically meaningful result. The latter will briefly be shown to have been achieved and the former will then be the focus of this work.

2 Methodology

The overwhelming problem with the Navier-Stokes Equations is that the momentum equations for each dimension have pressure gradient terms, but there does not exist a governing equation to solve for the pressure. A trick to getting around this is to manipulate the N-S equations to remove the pressure terms. Such an approach is called the Streamfunction-Vorticity method.

2.1 Streamfunction-Vorticity

The 2-dimensional Navier-Stokes equations consist of the x- and y-momentum equations and the continuity equation

$$\rho\left[\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}\right]=-\frac{\partial p}{\partial x}+\mu\left[\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right] \tag{1}$$

$$\rho\left[\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}\right]=-\frac{\partial p}{\partial y}+\mu\left[\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right] \tag{2}$$

$$\frac{\partial \rho}{\partial t}+\frac{\partial(\rho u)}{\partial x}+\frac{\partial(\rho v)}{\partial y}=0 \tag{3}$$

where ρ is the fluid density, p the pressure, μ the viscosity and u and v are the x- and y-velocities, respectively. Assuming an incompressible fluid, the density ρ is constant, so (3) can be written as

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0$$

Now, taking the y-derivative of (1) and the x-derivative of (2),

$$\begin{aligned}&\rho\left[\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial y}\right)+\frac{\partial u}{\partial y}\frac{\partial u}{\partial x}+u\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x}\right)+\frac{\partial v}{\partial y}\frac{\partial u}{\partial y}+v\frac{\partial^2 u}{\partial y^2}\right]\\&=-\frac{\partial^2 p}{\partial x\partial y}+\mu\frac{\partial^2}{\partial x^2}\frac{\partial u}{\partial y}+\mu\frac{\partial^2}{\partial y^2}\frac{\partial u}{\partial y}\end{aligned} \tag{4}$$

$$\begin{aligned}&\rho\left[\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x}\right)+\frac{\partial v}{\partial x}\frac{\partial v}{\partial y}+v\frac{\partial}{\partial x}\left(\frac{\partial v}{\partial y}\right)+\frac{\partial u}{\partial x}\frac{\partial v}{\partial x}+u\frac{\partial^2 v}{\partial x^2}\right]\\&=-\frac{\partial^2 p}{\partial x\partial y}+\mu\frac{\partial^2}{\partial x^2}\frac{\partial v}{\partial x}+\mu\frac{\partial^2}{\partial y^2}\frac{\partial v}{\partial x}\end{aligned} \tag{5}$$

and subtracting (5) from (4) to remove the pressure gradient terms,

$$\begin{aligned}&\rho\left[\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)+\frac{\partial u}{\partial x}\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)+u\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)\right.\\&\left.+\frac{\partial v}{\partial y}\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)+v\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)\right]=\mu\nabla^2\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right)\end{aligned} \tag{6}$$

Define the vorticity $\theta = \nabla \times \vec{\mathbf{v}}$ as the curl of the velocity field. In 2 dimensions, this expression for vorticity reduces to

$$\theta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \tag{7}$$

Using this and the continuity equation above, (6) can be written as a convection-diffusion type equation

$$\boxed{\rho\left[\frac{\partial \theta}{\partial t} + u\frac{\partial \theta}{\partial x} + v\frac{\partial \theta}{\partial y}\right] = \mu\nabla^2\theta} \tag{8}$$

Now, define another parameter ψ as

$$u = \frac{\partial \psi}{\partial y}$$
$$v = -\frac{\partial \psi}{\partial x}$$

then, from the definition of θ, it is easy to see that

$$\begin{aligned}\frac{\partial u}{\partial y} &= \frac{\partial^2 \psi}{\partial y^2}\\ \frac{\partial v}{\partial x} &= -\frac{\partial^2 \psi}{\partial x^2}\end{aligned} \tag{9}$$

$$\boxed{\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\theta} \tag{10}$$

Now (8) can be iteratively solved as a Convection-Diffusion type equation for the vorticity, θ, and (10) can be solved as a Poisson equation with source term $-\theta$. Once ψ - which shall be called the streamfunction - is known, u and v can be directly calculated via (9).

In many fluid-flow problems, it is important to have an understanding of the pressure field in the fluid. The Streamfunction-Vorticity approach attmpts to remove the pressure gradient terms from the formulation, however, with a little trick, it is simple to calculate. Taking the x-derivate of (1) and the y-derivative of (2)

$$\begin{aligned}&\rho\left[\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial x}\right) + u\frac{\partial^2 u}{\partial x^2} + \left(\frac{\partial u}{\partial x}\right)^2 + \frac{\partial v}{\partial x}\frac{\partial u}{\partial y} + v\frac{\partial^2 u}{\partial x \partial y}\right]\\ &\quad = -\frac{\partial^2 p}{\partial x^2} + \mu\nabla^2\left(\frac{\partial u}{\partial x}\right)\end{aligned} \tag{11}$$

$$\begin{aligned}\rho\Bigg[\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial y}\right)+v\frac{\partial^2 v}{\partial y^2}+\left(\frac{\partial v}{\partial y}\right)^2+\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}+u\frac{\partial^2 v}{\partial x\partial y}\Bigg]\\ =-\frac{\partial^2 p}{\partial y^2}+\mu\nabla^2\left(\frac{\partial v}{\partial y}\right)\end{aligned}\tag{12}$$

Adding (11) and (12) and using the continuity equation,

$$\begin{aligned}\rho\Bigg[\left(\frac{\partial v}{\partial y}\right)^2+\left(\frac{\partial u}{\partial x}\right)^2+2\frac{\partial u}{\partial x}\frac{\partial v}{\partial y}-2\frac{\partial u}{\partial x}\frac{\partial v}{\partial y}+2\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\\ =-\left(\frac{\partial^2 p}{\partial y^2}+\frac{\partial^2 p}{\partial x^2}\right)\end{aligned}$$

$$\boxed{\nabla^2 p=2\rho\left[\frac{\partial u}{\partial x}\frac{\partial v}{\partial y}-\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\right]}\tag{13}$$

which is also a Poisson equation with source term depending on u and v, which will be known. It can be seen that the streamfunction ψ and the vorticity θ are independent of the pressure p. Therefore, it is only necessary to solve the pressure Poisson equation for time-steps where the pressure field is required, and not for every intermediate step.

It will be seen later that the Poisson equations require a large amount of computation time relative to the Convection-Diffusion equations. As such, a proportionately large amount of time and effort will be applied to optimising the Poisson equation solvers.

2.2 Boundary Conditions

At the left and right walls, the x-velocity is given by $u(0)=u(L)=\frac{\partial\psi}{\partial y}\Big|_0=\frac{\partial\psi}{\partial y}\Big|_L=0$ so $\psi=const$. Similarly for the top and bottom walls, the y-velocity is given by $-\frac{\partial\psi}{\partial x}\Big|_0=-\frac{\partial\psi}{\partial x}\Big|_L=0$ so $\psi=const$ also. By continuity, one can say that ψ has the same value on all boundaries and set $\psi=0$, creating a Dirichlet boundary problem for the streamfunction.

Expanding ψ on a horizontal boundary using a Taylor Series expansion provides a boundary condition for θ

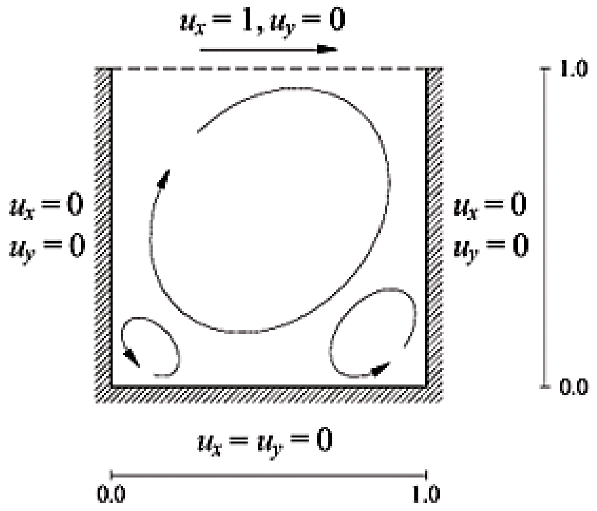


Figure 1: A schematic of the 2-dimensional 'Lid-Driven Cavity' problem. *Picture: NACAD*

$$
\begin{aligned}
\psi_1 &= \psi_0 + \left.\frac{\partial \psi}{\partial y}\right|_0 \Delta y + \left.\frac{\partial^2 \psi}{\partial y^2}\right|_0 \frac{\Delta y^2}{2} + O(\Delta y^3) \\
&= \psi_0 + u_0 \Delta y + \left.\frac{\partial u}{\partial y}\right|_0 \frac{\Delta y^2}{2} + O(\Delta y^3) \\
&= \psi_0 + u_0 \Delta y + (-\theta_0) \frac{\Delta y^2}{2} + O(\Delta y^3) \\
\theta_0 &= \frac{2}{\Delta y^2} (\psi_0 - \psi_1) + O(\Delta y)
\end{aligned}
\tag{14}
$$

A higher order approximation for boundary terms can be achieved from expanding the relevant Taylor series'. This gives second order accurate vorticity boundary conditions in contrast to the first order accurate condition above

$$
\begin{aligned}
\theta_0 &= -\left.\frac{\partial u}{\partial y}\right|_0 \\
&= \left(\frac{u_0 - 4u_1 + 3u_2}{2\Delta y}\right) + O(\Delta y^2)
\end{aligned}
\tag{15}
$$

2.3 Reynolds Number

The Reynolds number *Re* is a key parameter in assessing the behaviour of fluid flow systems; two fluids with radically different properties but with similar Reynolds number will, ceteris paribus, behave in much the same manner. A dimensionless parameter, Re is defined as the ratio of inertial forces to viscous forces. It appears in the discretization of the Convection-Diffusion equation in the form ρ/μ.

In this problem's discretization, the stability criteria will require that the mesh size be scaled with the inverse of Reynolds number and the time step be scaled with the square of the mesh size: clearly the chosen value for Re will affect the computational time of the problem, but not the complexity. For this reason, the majority of the testing will be kept confined to the low end of the Reynolds number spectrum. Ghia et al. [4], with whom the results of this work will be compared, produced velocity field data for Reynolds numbers $= \{100, 400, 1000, 3200\}$.

3 Software Design

The current formulation requires allocated memory for the streamfunction (and, by extension, the x- and y-velocities), the vorticity, and the pressure at every point in the grid. A global data structure called a 'Cell' is declared. Each Cell has the 5 member variables just listed as `double` precision floating point numbers, with an extra variable of the same type for storing the vorticity at the previous timestep - this is necessary for the soon-to-be-chosen vorticity solver. While certain algorithms such as the Fast Fourier Transform (FFT) and MultiGrid will allocate more memory (in the form of arrays of `doubles` for transforms, `MPI_Windows` for RMA operations, etc.), for a typical problem size (say, `N=128`), 128^2 grid Cells are allocated, each 48 bytes in size, for a total of less than 1MB of memory.

3.1 Convection-Diffusion Equation

For simplicity, define a parameter a as the inverse of the Reynolds number. Now, (8) can be written

$$\frac{\partial \theta}{\partial t} = a\nabla^2\theta - u\frac{\partial \theta}{\partial x} - v\frac{\partial \theta}{\partial y} \tag{16}$$

Discretizing the time derivative using forward difference at time n and discretizing all spatial derivatives using a central difference scheme, this is equal to

$$\begin{aligned}\frac{\theta_{i,j}^{n+1} - \theta_{i,j}^{n}}{\Delta t} &= a\left(\frac{\theta_{i-1,j}^{n} + \theta_{i+1,j}^{n} + \theta_{i,j-1}^{n} + \theta_{i,j+1}^{n} - 4\theta_{i,j}^{n}}{h^2}\right) \\ &- u\left(\frac{\theta_{i+1,j}^{n} - \theta_{i-1,j}^{n}}{2h}\right) - v\left(\frac{\theta_{i,j+1}^{n} - \theta_{i,j-1}^{n}}{2h}\right) \\ &+ O(\Delta t) + O(h^2)\end{aligned} \tag{17}$$

where h is the mesh size, (i, j) is the grid point located at (ih, jh), and subscript n denotes the value of the variable at time $n\Delta t$. Rearranging and substituting $r = \Delta t/2h^2$ produces an implicit expression for $\theta_{i,j}^{n+1}$

$$\begin{aligned}\theta_{i,j}^{n+1} &= (1 - 8ra)\,\theta_{i,j}^{n} \\ &+ (2ra + rhu)\,\theta_{i-1,j}^{n} + (2ra - rhu)\,\theta_{i+1,j}^{n} \\ &+ (2ra + rhv)\,\theta_{i,j-1}^{n} + (2ra - rhv)\,\theta_{i,j+1}^{n}\end{aligned} \tag{18}$$

All of the values of θ at time $n+1$ depend on the neighbouring values at time n - i.e. the initial conditions. Alternatively, the Convection-Diffusion equation could be discretized using backward difference at time $n+1$ and central difference space. This would also yield an $O(\Delta t)$ approximation but with the advantage of being unconditionally stable. The disadvantage is that the implicit nature of the scheme requires an iterative algorithm to converge to a solution.

3.1.1 Crank-Nicolson

A mixture of implicit and explicit schemes yields the Crank-Nicolson scheme - a second order time approach with less restrictive stability criteria compared to the explicit scheme above. If the equation to be discretized reads

$$\frac{\partial \theta}{\partial t} = F(t, x, \frac{\partial \theta}{\partial x}, \frac{\partial^2 \theta}{\partial x^2}) \tag{19}$$

then Crank-Nicolson, itself an implicit scheme, describes a mix between the explicit and implicit schemes above

$$\frac{\partial \theta}{\partial t} = \frac{1}{2}\left[F^{n+1}\left(t, x, \frac{\partial \theta}{\partial x}, \frac{\partial^2 \theta}{\partial x^2}\right) + F^{n}\left(t, x, \frac{\partial \theta}{\partial x}, \frac{\partial^2 \theta}{\partial x^2}\right)\right] \tag{20}$$

When the Convection-Diffusion equation is discretized according to this scheme, $\theta_{i,j}^{n+1}$ is given by

$$\begin{aligned}\theta_{i,j}^{n+1} =& \frac{1}{1+4ra}\left[(1-4ra)\,\theta_{i,j}^{n}\right.\\ &+\left(ra+\frac{rhu}{2}\right)\left(\theta_{i-1,j}^{n+1}+\theta_{i-1,j}^{n}\right)+\left(ra-\frac{rhu}{2}\right)\left(\theta_{i+1,j}^{n+1}+\theta_{i+1,j}^{n}\right)\\ &\left.+\left(ra+\frac{rhv}{2}\right)\left(\theta_{i,j-1}^{n+1}+\theta_{i,j-1}^{n}\right)+\left(ra-\frac{rhv}{2}\right)\left(\theta_{i,j+1}^{n+1}+\theta_{i,j+1}^{n}\right)\right]\end{aligned} \tag{21}$$

where $r = \Delta t/2h^2$ as above. This yields a second order accurate in space *and* second order accurate in time iterative algorithm.

3.2 Poisson Equation

Demmel [5] provides a comprehensive list of 2D Poisson solving algorithms which are presented in the table below along with serial run-times, PRAM run-times (optimal parallel implementation with no communication cost) and memory footprint. While noting that it is not always clear which algorithm is preferable, those listed below are roughly in order of performance.

Algorithm	Serial	PRAM	Memory
Jacobi	N^2	N	N
Conj. grad.	$N^{3/2}$	$N^{1/2} log N$	N
RB GS	$N^{3/2}$	$N^{1/2}$	N
FFT	$N log N$	$log N$	N
MultiGrid	N	$log^2 N$	N
Lower bound	N	$log N$	N

All algorithms have optimal memory footprint. The MultiGrid algorithm performs optimally in serial, while the FFT performs the best in parallel. The performance of these two shall be investigated and, for completeness, the Red-Black Gauss-Seidel shall also be investigated. The Gauss-Seidel, which is a simple solver to implement, may also be used as a reference for the physical results of other implementations.

3.2.1 Gauss-Seidel

The discretization of the Poisson Equation for the streamfunction is simply converting two second order spatial derivatives into two second order central differences. Therefore (10) is approximated by the following 5-point stencil

$$\frac{\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1} - 4\psi_{i,j}}{h^2} = -\theta + O(h^2) \tag{22}$$

$$\psi_{i,j} = \frac{\theta h^2 + \psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}}{4} \tag{23}$$

The Poisson Equation for pressure is discretized similarly.

Equation (23) is the well known Gauss-Seidel iterative solver - a reliable, stable, and convergent solver for the Poisson equation. It is also easily parallelisable when presented in the 'red-black' or 'chessboard' configuration. Instead of the usual iteration through consecutive points, the equation is first solved on all 'red' points, using the values at the 'black' neighbours, and then at the 'black' points, using the values at the 'red' neighbours. In this way, the result is independent of the order in which the points are solved.

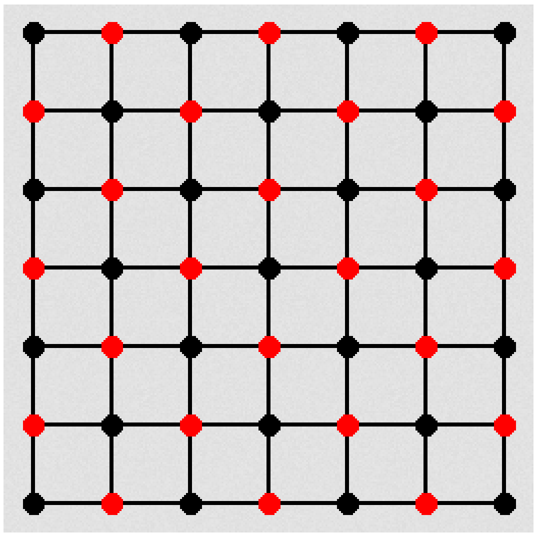


Figure 2: 'Red-black' or 'chessboard' configuration for solving the Gauss-Seidel algorithm

3.2.2 Relaxed Gauss-Seidel

A Gauss-Seidel algorithm can be modified from the familiar

$$x^{k+1} = f(x^k) \tag{24}$$

to an equation of the style

$$x^{k+1} = (1-\omega) + \omega f(x^k) \tag{25}$$

where $0 < \omega < 2$ is the relaxation parameter. From observation it is clear that $\omega = 1$ produces the regular Gauss-Seidel iterative algorithm. It is well known [1, p. 285] that the convergence of the Poisson equation with Dirichlet boundary conditions is fastest with

$$\omega = \frac{2}{1 + sin(\pi h)} \tag{26}$$

with the usual $h = 1/(N-1)$ convention. As the mesh size approaches zero, the optimal relaxation parameter approaches 2. This is useful for solving the Poisson equation in serial, however upon parallelization, local boundary conditions change. Now the optimal relaxation parameter is unknown, and an optimal parameter in the serial case may result in a non-converging solution in the parallel case.

At this point, the Poisson solver is an $O(N^{3/2})$, iterative scheme with convergence issues. An alternative scheme would be desirable.

3.2.3 FFTW Fast Poisson Solver

The Poisson solver being used up to now has been an $O(N^{3/2})$ iterative algorithm, that also has convergence and, in some cases, stability criteria. Solving the Poisson equation by means of an FFT algorithm produces an exact solution in $O(NlogN)$ time, potentially providing a speed-up of multiple orders of magnitude.

The streamfunction has Dirichlet boundary conditions so a regular Discrete Fourier Transform (DFT) will be of no use here, as that transform provides periodic boundary conditions. For Dirichlet boundary conditions, consider the Discrete Sine Transform, specifically DST-I.

Take the Poisson equation

$$\frac{U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j}}{h^2} = -S_{i,j} \tag{27}$$

with source term $S_{i,j}$. The 2-dimensional DST-I of $U_{i,j}$ is given by

$$U_{i,j} = \sum_{m=1}^{N-1} \sum_{n=1}^{N-1} \widetilde{U}_{m,n} sin\left(\frac{\pi}{N+1}(m+1)(i+1)\right) sin\left(\frac{\pi}{N+1}(n+1)(j+1)\right)$$
$$\widetilde{U}_{m,n} = \sum_{i=1}^{N-1} \sum_{j=1}^{N-1} U_{i,j} sin\left(\frac{\pi}{N+1}(i+1)(m+1)\right) sin\left(\frac{\pi}{N+1}(j+1)(n+1)\right)$$

Transforming the relevant expressions for U and S and substituting into (27) yields, after equating similar terms on both sides,

$$\widetilde{U}_{m,n} = \frac{h^2 \widetilde{S}_{m,n}}{4 - 2cos\left(\frac{\pi m}{N}\right) - 2cos\left(\frac{\pi n}{N}\right)} \tag{28}$$

Inversely transforming then yields the solution in $O(NlogN)$ time. This project will utilize the FFTW routines [2] which have built-in MPI and thread support, both of which will ultimately be utilised for a hybrid parallelization.

3.2.4 MultiGrid

An exact $O(NlogN)$ solver like the FFTW is known for being one of the fastest Poisson solvers, however the MultiGrid algorithm is also an accurate and efficient solver. The MultiGrid method solves a problem on the premise of solving for the residual error of an approximation on successively coarser grids, and then interpolating this residual error as a correction to higher resolution grids. Take the Poisson equation for the streamfunction

$$Ax = b$$

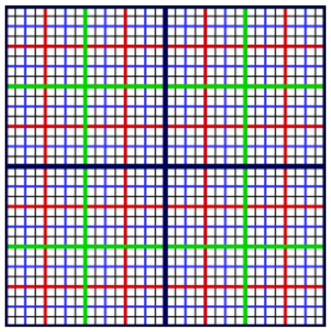
where A is the coefficient matrix for the Poisson equation. At any point in time, there exists an approximate solution $\hat{x}$ and a residual error $r = b - A\hat{x}$. If $\hat{x}$ is the exact solution, then $r = 0$. If $\hat{x}$ is an approximation, then r is non-zero and solving the non-trivial Poisson equation $Ad = r$ gives a correction to $\hat{x}$. Then the exact solution is $x = \hat{x} + d$, since

$$\begin{aligned} A(\hat{x} + d) &= A\hat{x} + Ad \\ &= b - r + r \\ &= b \end{aligned}$$

MultiGrid calculates an initial smoothed approximation to x on the finest grid (say, level `m`), and then finds the residual error to this approximation. It then restricts this residual error to the next coarsest grid (say, level `m-1`). To solve for this residual error, it gets a smoothed approximation on level `m-1` and solves for the residual. Restrict the residual to level `m-2`, and so on. This is a recursive scheme, ending on the coarsest possible grid (level `1`) with two boundary points and one internal point in each dimension. On this level, MultiGrid gets the exact solution to the 5-point Poisson discretization. It then interpolates the calculated residual errors back up to finer and finer grids, correcting each approximation and applying further smoothing until arriving back at level `m` and correcting the approximation for x. This method is referred to as a V-cycle, after the shape of the algorithm on a graph of level vs. time (Figure 3).

The purpose of the smoothing at each level is to dampen the high frequency errors. This damping is extremely effective for the first couple of iterations and less so from then on, as there are less high frequency errors to damp. For this reason, only a small number of smoothing iterations need to be executed on each V-cycle level, decimating the computation time versus the Gauss-Seidel algorithm which uses the same 5-point stencil and relaxed iteration. This project uses a $V(2,1)$ algorithm - 2 iterations in the first smoothing and 1 in the second.

The Full MultiGrid method works by obtaining the exact solution on the coarsest grid, `1`, and using this an as initial guess for solving on the second coarsest grid, `2`. Every initial guess from `2` to `m` is solved using a V-cycle.



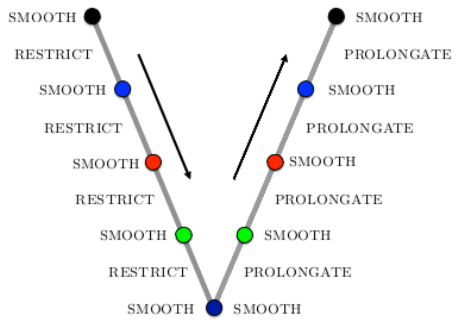


Figure 3: Grid resolutions corresponding to position in the MultiGrid V-cycle

Listing 1: MultiGrid V-Cycle

```
 Vcycle(double *b, double *x, int level)
{
        if level = 1
                x = smooth(b, x)
        else
                x = smooth(b, x)
                res = GetResidual(b, x)
                res = Restrict(res)

                Vcycle(res, d, level-1)

                d = Interpolate(d)
                x = Correct(d, x)
                x = smooth(b, x)
        endif
}
```

Listing 2: Full MultiGrid

```
FMG(double *b, double *x)
{
        x = smooth(b, x, level=1)
        for level = 2 : m
                x = Interpolate(x)
                Vcycle(b, x, level)
        endfor
}
```

3.3 Non-Uniform Adaptive Mesh Refinement

In many high-Re simulations, the physical solution is all but cosmic. Often, the mesh is too fine where it need only be coarse and often it is not fine enough. What is needed is a mesh that is non-uniform over space - coarse where the function is static, and finer where the function is in flux - and adaptive over time. This method can be used on both the Convection-Diffusion equation and the Poisson equation with separately generated meshes. In the following analysis, first and second order derivatives will be discretized on an generalized non-uniform mesh in one dimension and their accuracy will be considered.

Consider a 3-point mesh in one dimension, $M_x = \{a < b < c\}$. Denote the space steps $h_1 = b - a$ and $h_2 = c - b$, where $h_1, h_2 > 0$. Consider further a smooth function that takes the values $\{u(a), u(b), u(c)\}$ over the mesh. Any first or second order derivative of u about the point b can be expressed by the term

$$A = \alpha u(a) + \beta u(b) + \gamma u(c) \tag{29}$$

where the coefficients α, β and γ must be determined.
Taylor expanding around the points $u(a)$ and $u(c)$ gives

$$u(a) = u(b) - h_1 u'(b) + \frac{h_1^2}{2}u''(b) - \frac{h_1^3}{6}u'''(b) + \frac{h_1^4}{24}u^{iv}(b) + O(h_1^5) \tag{30}$$

$$u(c) = u(b) + h_2 u'(b) + \frac{h_2^2}{2}u''(b) + \frac{h_2^3}{6}u'''(b) + \frac{h_2^4}{24}u^{iv}(b) + O(h_2^5) \tag{31}$$

Substituting these back into (29) gives the relation

$$\begin{aligned} A =&(\alpha + \beta + \gamma)u(b) + (-\alpha h_1 + \gamma h_2)\, u'(b) + \left(\alpha\frac{h_1^2}{2} + \gamma\frac{h_2^2}{2}\right) u''(b) \\ &+ \left(-\alpha\frac{h_1^3}{6} + \gamma\frac{h_2^3}{6}\right) u'''(b) + \left(\alpha\frac{h_1^4}{24} + \gamma\frac{h_2^4}{24}\right) u^{iv}(b) + O(h_1^5, h_2^5) \\ =&\sigma_0 u(b) + \sigma_1 u'(b) + \sigma_2 u''(b) + \sigma_3 u'''(b) + \sigma_4 u^{iv}(b) + O(h_1^5, h_2^5) \end{aligned} \tag{32}$$

To calculate the values of α, β and γ, set the value of the coefficient of the desired derivative to 1 and as many of the others as possible to 0, giving priority to the highest order derivative terms. For example, to compute the first derivative, set $\sigma_1 = 1$ and all other $\sigma_i = 0$. This produces the set of linear equations

$$\begin{aligned} &\alpha + \beta + \gamma = 0 \\ &-\alpha h_1 + \gamma h_2 = 1 \\ &\alpha\frac{h_1^2}{2} + \gamma\frac{h_2^2}{2} = 0 \end{aligned} \tag{33}$$

which, when solved, gives

$$\alpha = -\frac{h_2}{h_1(h_1+h_2)}, \beta = -\alpha - \gamma, \gamma = \frac{h_1}{h_2(h_1+h_2)}$$

Writing equation (29) in terms of h_1 and h_2, the first derivative approximation is

$$\begin{aligned} A = &- \frac{h_2}{h_1(h_1+h_2)}u(a) + \frac{h_1}{h_2(h_1+h_2)}u(c) \\ &+ \left(\frac{h_2}{h_1(h_1+h_2)} - \frac{h_1}{h_2(h_1+h_2)}\right)u(b) \\ &\approx u'(b) + \frac{h_1h_2}{6}u'''(b) \end{aligned}$$

A similar analysis to find an expression for the second derivative yields the approximation

$$\begin{aligned} A = &- \frac{2}{h_1(h_1+h_2)}u(a) + \frac{2}{h_2(h_1+h_2)}u(c) \\ &+ \left(-\frac{2}{h_1(h_1+h_2)} - \frac{2}{h_2(h_1+h_2)}\right)u(b) \\ &\approx u''(b) + \frac{h_2-h_1}{3}u'''(b) + \frac{h_1^2 - h_1h_2 + h_2^2}{12}u^{iv}(b) \end{aligned}$$

In both cases, it is clear from observation that if a uniform mesh $h_1 = h_2 = h$ was used rather than a non-uniform, the expressions for the first and second derivatives would reduce to the usual second-order accurate finite difference operators

$$\begin{aligned} u'(b) &= -\frac{1}{h^2}u(a) + \frac{1}{h^2}u(c) + O(h^2) \\ u''(b) &= \frac{1}{h^2}u(a) - \frac{2}{h^2}u(b) + \frac{1}{h^2}u(c) + O(h^2) \end{aligned}$$

The lack of symmetry in the non-uniform mesh introduces higher order errors, turning the second derivative into a first-order accurate discretization. Indeed, Sfakianakis [3] shows that a second derivative approximation on a non-uniform mesh of this kind may be *at most* first order accurate. This will play an important role later when looking at this non-uniform mesh method for solving for the streamfunction and pressure - both of which are solutions to Poisson equations and depend heavily on the accuracy of the second derivative approximations.

However, ignoring this numerical inaccuracy for now, this non-uniform mesh implementation involves the principal of equi-distributed grid points along the solution curve. To ensure that the points are equally spaced along the curve, some information must be gained from the function itself which in turn determines the mesh spacings. Sfakianakis derived the curvature of a function $u\,(x)$ to be

$$K_u(x) = \frac{|u''(x)|}{\left(1 + (u'(x))^2\right)^{3/2}} \tag{34}$$

and another function called the monitor function as

$$M_i^{dscr} = \int_0^{x_i^n} K_{U^n}(x)\,dx \tag{35}$$

The curvature and monitor functions are discretized and, from this discretization and a set of simultaneous equations, the mesh spacing is given by

$$x_i^{new} = x_k^n + \left(x_{k+1}^n - x_k^n\right) \frac{\frac{i}{N} M_N^{dscr} - M_{U^n}\left(x_k^n\right)}{M_{U^n}\left(x_{k+1}^n\right) - M_{U^n}\left(x_k^n\right)} \tag{36}$$

A code is written and tested on a sample problem later in this report.

3.4 Stability

The Hurwitz stability criteria states that stability is implied when all coefficients are of the same sign (all positive or all negative). For the explicit Convection-Diffusion equation, requiring all coefficients to be negative invokes some peculiar velocity criteria (x- and y- components may never move slower than $2a/h$). Assert, then, all coefficients must be positive. Now,

$$\begin{aligned} 1 - 8ra &> 0 \\ 2h^2 &> 8a\Delta t \\ \Delta t &< \frac{h^2}{4a} \end{aligned} \tag{37}$$

where the parameter a corresponds to the inverse of the Reynolds number Re. Mesh spacing stability criteria is in turn given by

$$\begin{aligned} 2ra - rh\mathbf{u} &> 0 \\ 2a &> h\mathbf{u} \\ h &< \frac{2a}{\mathbf{u}} \end{aligned} \tag{38}$$

for $\mathbf{u} = \{u, v\}$.

The implicit discretization of the Convection-Diffusion equation is unconditionally stable.

The Crank-Nicolson scheme stability criteria are calculated in the same fashion as the explicit method. A stability analysis returns $\Delta t < h^2/2a$ and $h < 2a/\mathbf{u}$ for $\mathbf{u} = \{u, v\}$ - slightly less strict than the explicit method's criteria.

3.5 Parallelization

In the quest for optimization, multiple compute cores are collectively tasked with solving the relevant equations. In the absolutely ideal case, solving a problem on n cores should only take one nth the time as solving in serial. This ideal scenario cannot be met as there exists overheads with creating and destroying threads, initialising and finalising the MPI interface and time spent communicating relevant data between non-shared memory cores. While some of these overheads are concrete, some can be minimised by the user and some can be avoided altogether using hybrid programming.

3.5.1 MPI

The Message Passing Interface (MPI) is a high performing, scalable, and portable communications protocol used to program on distributed-memory computers. It is the dominant model used in High-Performance Computing today and as such will be the dominant model used in the parallelization of this code.

The initial method of sub-dividing the problem domain to accomodate multiple processors was a "tic-tac-toe" arrangement. If the problem is to be solved on a 2-dimensional square domain on 16 processors, for example, the domain will be split in 4 equal-sized sub-domains in the x-direction and 4 equal-sized sub-domains in the y-direction creating a total of 16 square sub-domains. Some simple geometry shows that this method minimizes the border area per processor, minimising the volume of data each processor needs to communicate to it's neighbours. This method also reduces the memory footprint of the problem as less "ghost cells"[1] need to be duplicated.

What the "tic-tac-toe" arrangement provides in memory management, it suffers in code complexity, number of communications per process and, in the case of the FFT, efficiency. The FFT solver performs a transform on all the rows of data at once, and then transforms all the columns of data at once. After some operation is performed on the transformed data, the data is then transformed back again, rows first, columns second. In a tic-tac-toe arrangement, there would need to be MPI communications from every process to every other in the same dimension for every transform in that dimension. It is more efficient, then, to sub-divide the problem domain in *only one* dimension, as in Figure 4. Then MPI communication is only necessary for one dimensions transforms and not both. Not only does this simplify the code somewhat, but now there are no necessary communications in the x-direction and the more efficient FFT solvers also earn a small performance boost.

[1]All of the discretizations involved in solving for the streamfunction and the vorticity are 5-point discretizations. They require knowledge of the data at the local point as well as the nearest neighbours in both the x- and y-directions. Never does a discretization require data belonging to a point more than one nearest neighbour away.

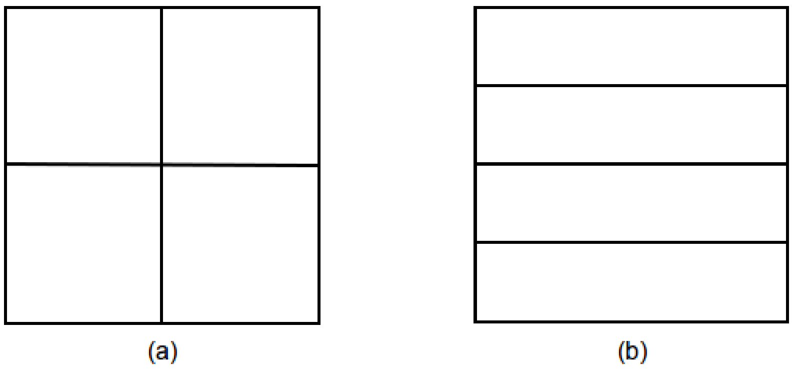


Figure 4: The "tic-tac-toe" memory distribution in (a) keeps inter-node boundaries to a minimum as the number of nodes increase, but the striped distribution in (b) reduces the complexity of the code, the number of communications each process must perform and is also an optimal configuration for the FFT Poisson solver.

3.5.2 OpenMP Threads

The MPI protocol is, by definition, not a shared-memory protocol. Data must be duplicated and communicated between distinct MPI nodes, creating a significant performance overhead. Most processors today have the characteristic of hyper-threading - having the ability to run multiple CPU cores on a single processor. These "threads" share a common memory base so suffer from no communication woes and benefit from a substantially smaller memory footprint - but at an increased risk of data corruption and race hazards, as well as a non-trivial overhead for creating and destroying threads when necessary.

3.5.3 Hybrid MPI/OpenMP

This project experimented somewhat in mixing MPI and OpenMP threads to achieve the best performance. This would not always be the best option: the portion of work designated to each thread must be large enough to justify the creation/destruction overhead (so there can't be too many threads) and it would also be desirable to keep the number of MPI tasks to a minimum to manage the duplication and communication of data. So this makes it obvious that for smaller problem sizes, the code would benefit from a smaller number of CPU cores, and vice versa.

The Lonsdale cluster at the Trinity Centre for High Performance Computing (TCHPC) - where this research was conducted - consists of a cluster of Quad-Core AMD Opteron processors. Each shared-memory node consists of 8 CPU cores. Parallelization options range from pure MPI (8 MPI tasks per node - an $(8, 1)$ configuration) to the purest threaded implementation: MPI communication between the main core of each node, and 8 total threads spawned off each of those "main" cores (a $(1, 8)$ configuration). Other options exist in between such as 2 MPI tasks per node with 4 threads per task, etc.

3.5.4 RMA

In the beginning, the nature of the MultiGrid algorithm implied difficulty in regular halo array exchanges. As the grid levels change, so do the nearest neighbours and their displacements in memory. While possible to calculate in advance what data one will want to exchange, it is tedious and impractical. Remote Memory Access (RMA) communication seemed to provide a much more intuitive approach to parallelizing the MultiGrid algorithm, in which a process can take data from other processes only when it is required, and without the permission of the other processes.

Rebuilding the memory structure of the MultiGrid algorithm was a must to minimise the probability of memory leaks and faults. Rather than allocate memory and MPI windows as they are needed, all of the memory must be allocated before beginning the algorithm. This enables the MPI windows to also be created as the address and size of the memory to be exposed is now known. This begs the question: how much memory needs to be allocated?

Starting on the finest grid, `level m` with $N = 2^m + 1$ grid points, and working through to the lowest level, `level 1`, the total number of data points is given by a converging sum

$$\sum_{i=0}^{m-1} \left(2^{m-i} + 1\right)^2 \tag{39}$$

Now this much memory, and an appropriately sized window, can be allocated in the initialization period and it is not necessary to have to allocate or de-allocate further until finalising the job.

When the memory was structured as such, it became apparent that halo arrays could in fact be practical. Non-blocking sends and receives were implemented and provided quite a performance benefit over RMA operations. The two are compared for the MultiGrid parallelisation in the Performance and Results sections.

There exists a small numerical misfortune with the MultiGrid algorithm in parallel: as the problem is restricted to coarser and coarser grids, each process has less grid-points to solve. In serial, the number of points on the minimum level is 3 in each dimension. In parallel, it may be the case that some processes have no data as MultiGrid traverses coarser and coarser grids. Even with 4 processes, at least one of them will be idle on the minimum level. To combat this inefficiency, assert that the minimum level is such that each process will always have a non-zero amount of data. Denoting the minimum permissible level by m and the number of processes by np, then

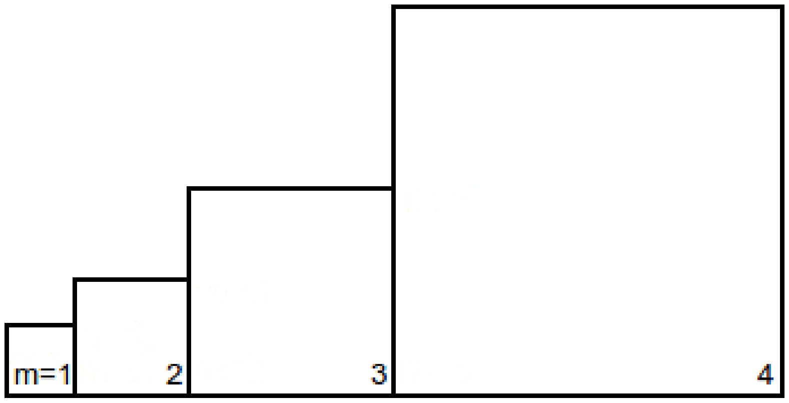


Figure 5: Total memory structure for `N=17` MultiGrid. The total area of the structure is given by the result of the converging sum above. This total amount is allocated as one contiguous block. An array of m pointers is also allocated, where `arr[i]` points to sub-block `i+1`. In parallel, each of these sections is scaled down by the number of processes.

$$\begin{aligned} N &= \frac{2^m}{np} > 1 \\ m &> log_2\left(np\right) \\ m &= 1 + log_2\left(np\right) \end{aligned} \tag{40}$$

since m is an integer. By losing out on the corrections from lower levels, the numerical accuracy will suffer somewhat.

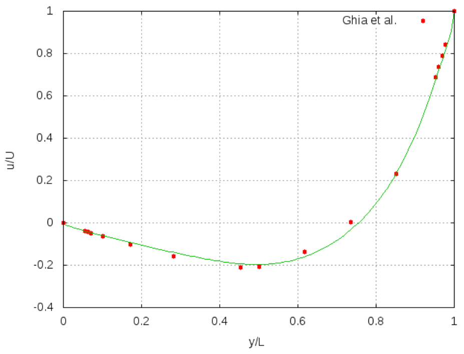
Every iteration of the MultiGrid algorithm executes a large number of MPI communication calls. Specifically, if there are m levels, every level above the minimum executes 9 communication calls, and the minimum level executes 2. This gives a total of $9m-7$ communication calls per iteration. This will be relevant when looking at the performance of the MultiGrid algorithm later.

This parallelisation led to some other peculiar behaviour. Initial tests done on Intel architecture proved convergence for small problem sizes and a small number of MPI tasks. Moving to Lonsdale's AMD architecture, it was found that for larger problem sizes and larger numbers of MPI tasks, the solution converged slower and eventually began to diverge. This required an increasing number of V-Cycles to keep the solution stable - but this destroyed any kind of efficiency of the algorithm. Converting to the Kelvin architecture at TCHPC (Intel Xeon processors), the solutions converged rapidly again for a $V(2,1)$ configuration.

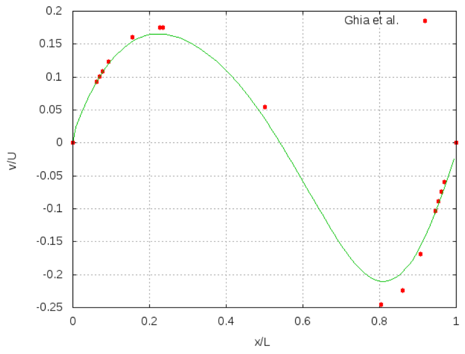
This hardware dependence of the MultiGrid algorithm was only realised in the final week of the project. As such, there was insufficient time to re-run both the FFT and Gauss-Seidel algorithms on Kelvin for consistency (the Intel Xeon processors on Kelvin have clock-speeds of 2.67 GHz compared to Lonsdale's AMD Opteron's 2.0 GHz). So while the compute times of the FFT/GS cannot be compared to those of the MultiGrid, one may still investigate how each algorithm scales with increasing parallelisation.

4 Testing

Requiring a physically meaningful result, while not the aim of this project, was absolutely necessary. Figures 6 and 7 below are comparisons between the results of this code and the results of Ghia et al. [4]. The graphs show the cross-sections of the horizontal (vertical) velocity along the vertical (horizontal) mid-way cross-section after the solution has reached equilibrium. Each algorithm gave very similar results with slightly increased (but negligible) accuracy for larger problem sizes. Shown here are the results for the FFT Poisson solver (exact) and Crank-Nicolson Convection-Diffusion solver (second order accurate in space and time) for a problem size of `N=128`.



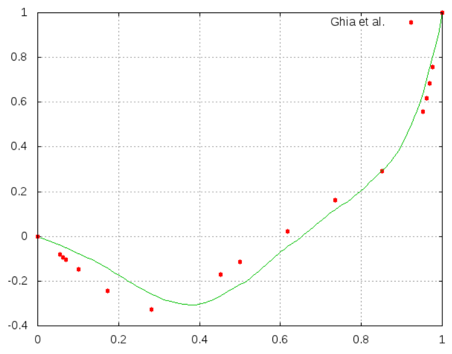
(a) x-velocity along x=0.5



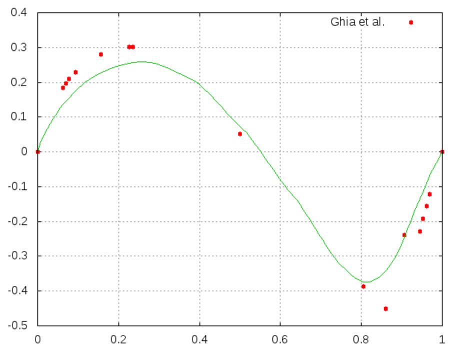
(b) y-velocity along y=0.5

Figure 6: Comparison of physical results with those of Ghia et al. [4] for Re = 100 on a 128x128 grid. For low Reynolds numbers such as this, it is not necessary to have ultra-fine mesh spacing to ensure accurate results.

The velocity field for the entire solution space at equilibrium is given in Figure 8 below. The left, right, and bottom walls are static and the top wall moves from left to right.



(a) x-velocity along x=0.5



(b) y-velocity along y=0.5

Figure 7: Comparison of physical results with those of Ghia et al. [4] for Re = 400 on a 128x128 grid. For higher Reynolds numbers, results are less accurate for the same mesh spacing as above. As the inertial forces dominate, the fluid is more likely to exhibit chaotic behaviour so a finer mesh and smaller time step are necessary.

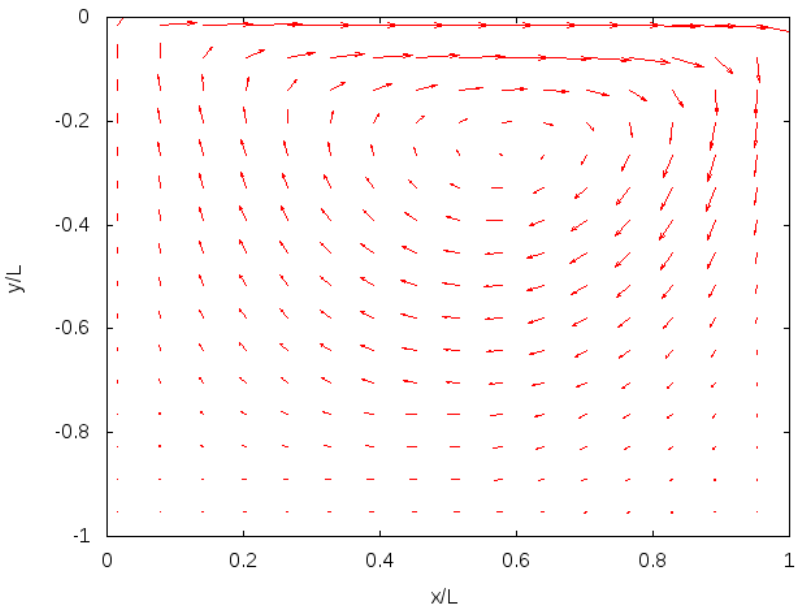


Figure 8: Equilibrium Velocity Field for Re = 100

4.1 Non-Uniform Adaptive Mesh Refinement

The method of non-uniform mesh refinement explored above provides certain numerical inaccuracies. First derivatives remain $O(h^2)$ accuracy (specifically $O(h_1 h_2)$, where h_1 and h_2 are the left and right mesh sizes, respectively). Second derivatives, unfortunately, are confined to first order accuracy (specifically $O(h_1 - h_2)$ - a numerical error that would not exist in the case of a uniform mesh). Since the Poisson equation is simply the sum of two second derivatives, this loss of accuracy plays a huge role.

$$\begin{aligned} \nabla^2 u &= -x\,(x+3)\,e^x \\ u &= -x\,(x-1)\,e^x \end{aligned} \tag{41}$$

The above one-dimensional sample Poisson problem is solved by the investigated non-uniform mesh refinement technique. Figure 9 below compares the obtained solution to the exact solution. Clearly the numerical inaccuracy of the non-uniform mesh solver leaves much to be desired, and for a Poisson equation this solver is not suitable and will not be pursued further in this project. A more suitable method will require identical mesh sizes to the left and right to redeem the second order accuracy of the second derivative. Possible approaches include recursive local refinement of square cells into 2^d smaller square cells, where d is the dimension. This approach would also require some elegant load balancing techniques between tasks. However due to time constraints on this project, this alternative approach will not be pursued and will be left as possible future work.

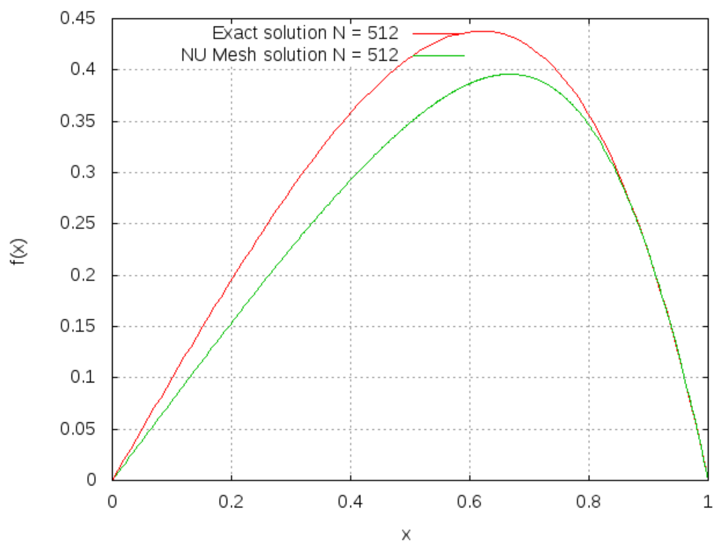


Figure 9: A sample Poisson problem is solved by an iterative non-uniform adaptive mesh refinement technique and the solution compared to the known exact solution. The first order accuracy of the second derivatives destroys the numerical accuracy of the non-uniform mesh solution.

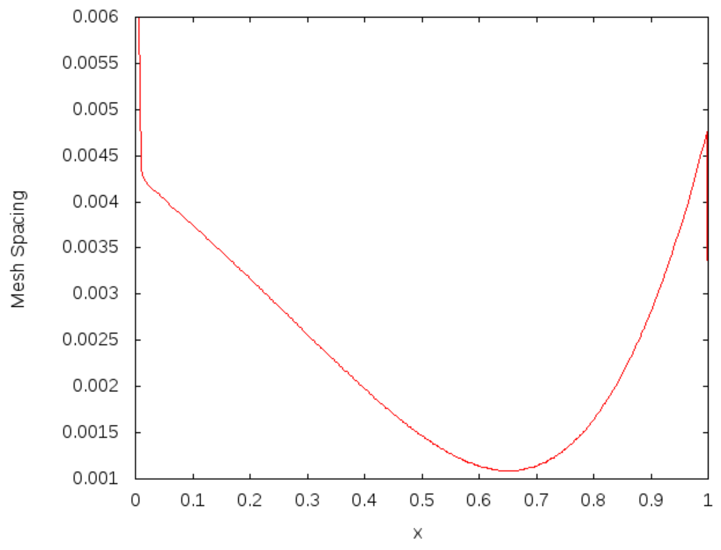


Figure 10: Mesh spacing as a function of x for the above problem. In general, the curvature function allocates finer mesh elements which scale with the second derivative of the function, so a constant or fixed slope function is satisfied with coarser elements

5 Performance

The Gauss-Seidel iterative algorithm took the solution at the previous time-step as an initial guess to solving the current time-step. While only providing a negligible performance boost in times of flux, approaching equilibrium this optimization resulted in an almost instantaneous solver. This effect was magnified at larger numbers of compute cores, as seen with the strong scaling of the Gauss-Seidel solver in Figure 21. A $(2, 4)$ hybrid solver provided improved scaling over pure MPI (Figure 24). Since the number of distributed-memory nodes is halved, the strong scaling can in general withstand twice as many processes before beginning to suffer from diminishing returns.

The FFT solver, while exact, did not benefit from the 'initial guess' approach that the Gauss-Seidel model did. However, iterations only took $O(NlogN)$ time and overall, the algorithm out-performed Gauss-Seidel for the serial LDC. The FFT scaled excellently with increasing compute cores initially, but lost this trait quite rapidly. The hybrid model performed very well even when the pure MPI implementation had begun to experience poor scaling, exhibiting the same resilience to diminishing returns as Gauss-Seidel.

5.1 MultiGrid Parallelisation

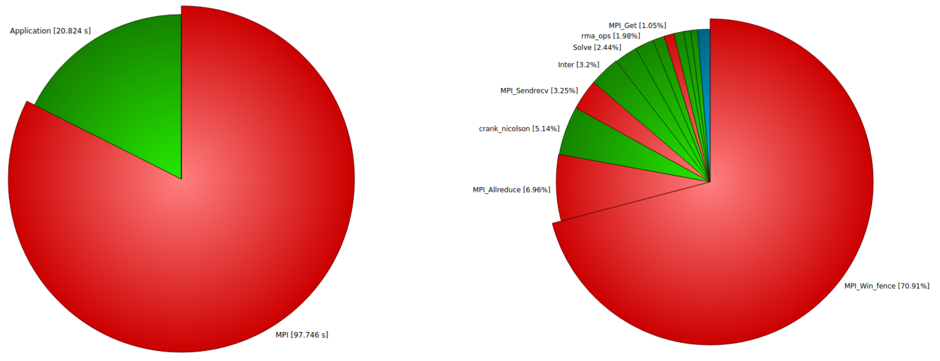


Figure 11: Vampir analysis of entire program using Crank-Nicolson and RMA MultiGrid. The left figure shows the proportion of time spent in MPI operations (red) vs. computation time (green). On the right, it is clear that the vast majority of the time spent in MPI operations is spent specifically in `MPI_Win_fence`, synchronizing the RMA windows.

An analysis of the RMA implementation of the MultiGrid algorithm provided disappointing results. While the algorithm was faster than the others overall for solving the Poisson problem, the scaling was very poor. Using the Vampir visualisation tool, one can see that a huge portion of the time spent in MultiGrid is spent on MPI operations (Figure 11). A closer look reveals that a large percentage of this time spent specifically on RMA synchronization. To this end, a MultiGrid implementation was created that would take advantage of the `MPI_Sendrecv` protocol, a dual purpose, blocking data exchange between two processes. Although the algorithm never experienced any deadlocks, analysis on Vampir showed that a cascade of blocking calls permeates through the processes on each data exchange (Figure 12). As was proven above, there are a large number of data exchanges per MultiGrid iteration and, as can be seen from Figure 12, this effect also worsens with an increasing number of compute cores. As a result of this blocking cascade, the processes were left comically out of sync for the duration of the MultiGrid solver. So one can easily deduce that the `MPI_Sendrecv` implementation will not scale well.

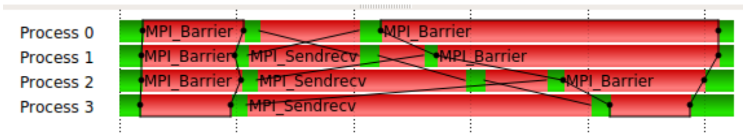


Figure 12: An example of a blocking cascade through processes using `MPI_Sendrecv`. Barriers added only for clarity.

The last bastion of hope for the parallel MultiGrid was non-blocking sends and receives. After care is taken to ensure `MPI_Wait` prevents any data corruption, the non-blocking sends and receives proved to scale very well. First, Figure 13 shows the performance and synchronization of the Crank-Nicolson iterative solver. Every iteration sees an `MPI_Allreduce` call followed by a non-blocking data exchange. Then Figures 14a and 14b show, on the same scale as Figure 13, the performance of the MultiGrid algorithm. Very dense MPI operations are observed as MultiGrid traverses the lower levels compared with increased computation time in the upper levels.

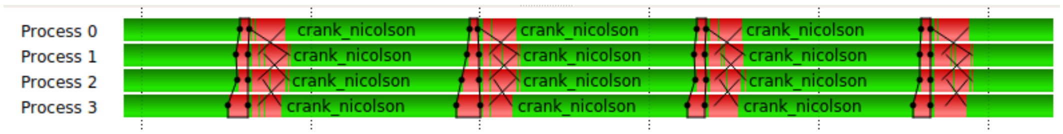
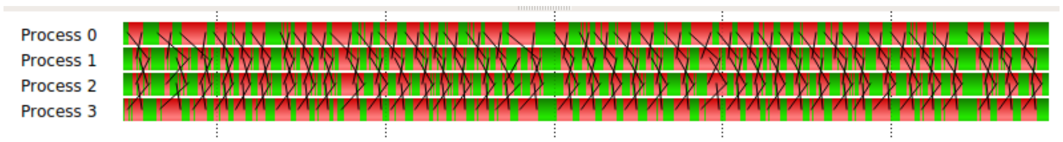
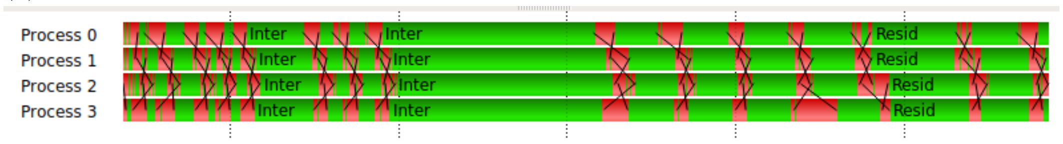


Figure 13: Crank-Nicolson solver performance under MPI.

The non-blocking implementation of the MultiGrid algorithm is not without it's MPI overheads, however the majority of the MPI communications overhead is a result of `MPI_Waitall`. The culprits here are the processes at the very top or very bottom of the grid which experience increased idle time since they only have one communication to do, compared to all other processes' two communications. As the number of cores increases (these results are from a 4-core job), this ratio should improve.

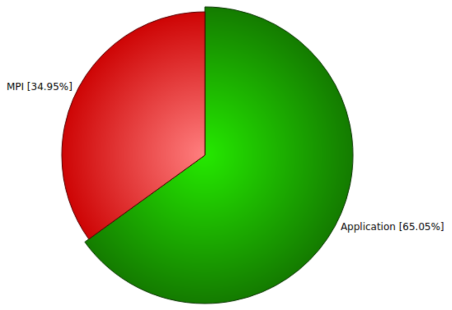


(a) MultiGrid lower levels



(b) MultiGrid upper levels

Figure 14: The performance of the non-blocking MultiGrid solver is preseted here at the same scale as the Crank-Nicolson solver analysis in Figure 13. The lower levels, while spending less time in computation, also have reduced communication time as a result of having to send less data. The higher levels approach the efficiency of the Crank-Nicolson solver. Overall, process synchronization is far improved over the blocking `MPI_Sendrecv` protocol and synchronization time has plummeted relative to the RMA implementation.



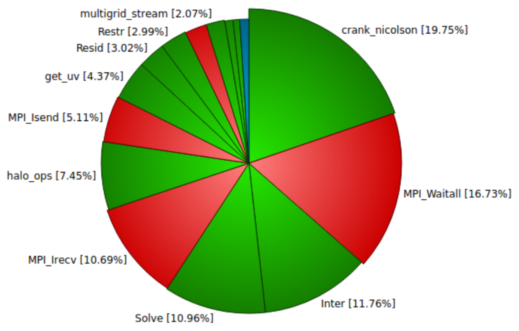


Figure 15: Performance of algorithm using the non-blocking MultiGrid solver. In comparison to Figure 11 above, scaling has drastically improved.

6 Results

6.1 Serial Codes

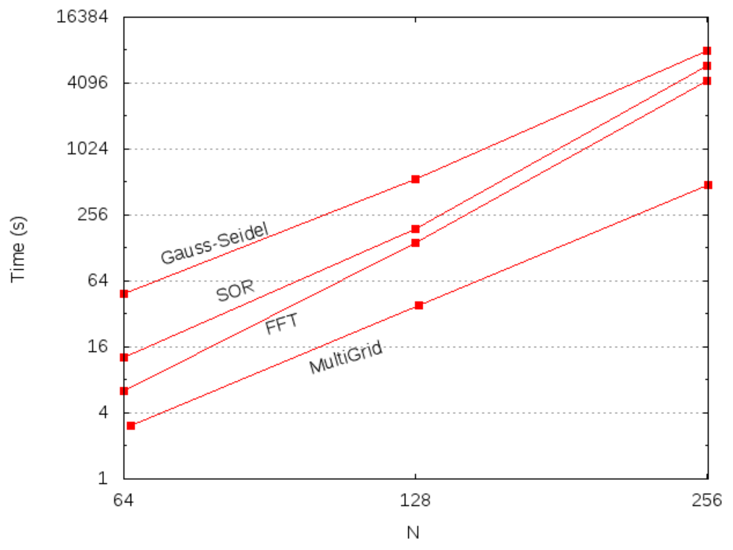


Figure 16: Comparison of different Poisson solvers in serial for various problem sizes. SOR refers to Successive Over-Relaxation - the relaxed Gauss-Seidel algorithm. It is clear that MultiGrid performs better than all other algorithms, roughly 8 times faster than the second fastest algorithm for `N=256`

6.2 Individual Algorithms

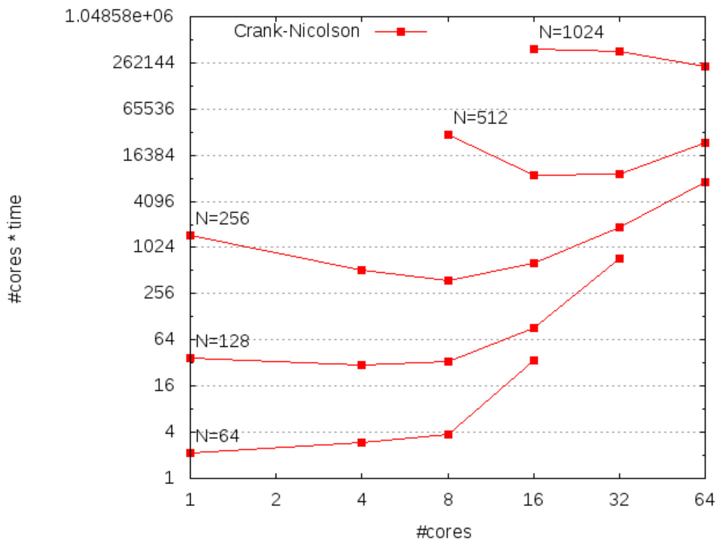


Figure 17: Scaling of the Crank-Nicolson solver for the Convection Diffusion equation using Pure MPI. Using the previous time-step's solution as an initial guess, super-optimal scaling was observed (downward slope) for cores $\ll$ `N`. However, the scaling at larger numbers of compute cores was poor and, for some large problem sizes, the vorticity solver took more compute time than the MultiGrid Poisson solver. Rather than being a necessarily bad thing for Crank-Nicolson, this is good press for MultiGrid. Our Poisson solver has caught up with the Convection-Diffusion solver.

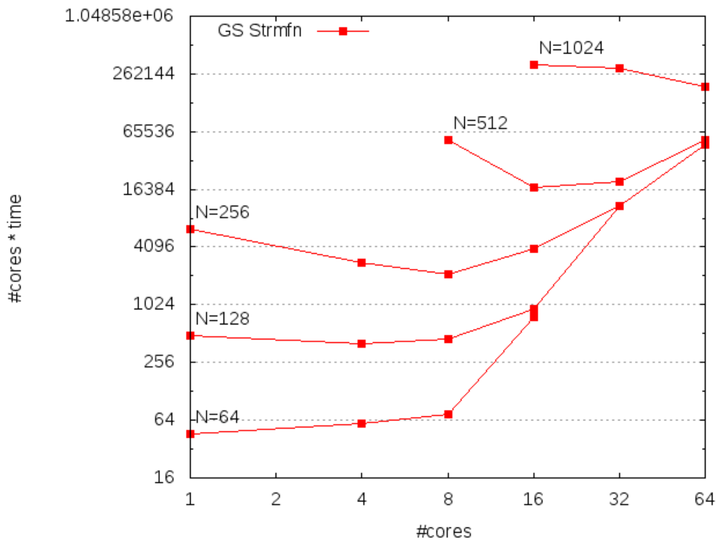


Figure 18: Scaling of the Gauss-Seidel solver for the Streamfunction Poisson equation using Pure MPI. This algorithm benefits from the same super-optimal scaling as Crank-Nicolson thanks to the initial guess optimization. Scaling suffers when the number of compute cores becomes large relative to `N`.

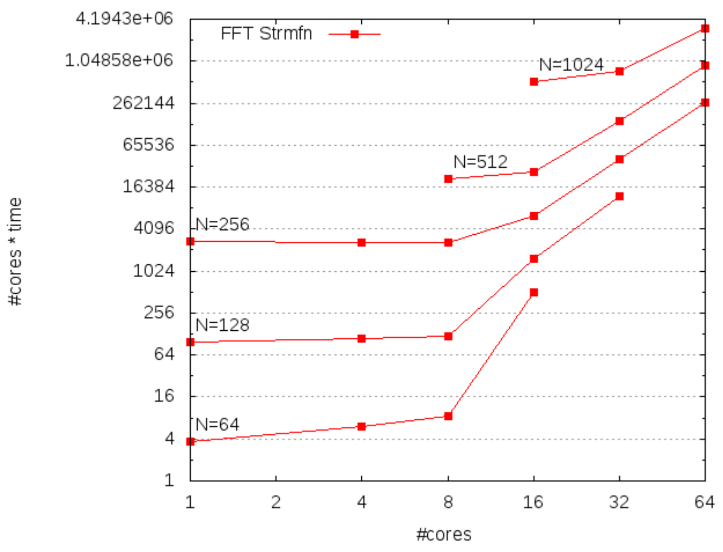


Figure 19: Scaling of the FFT solver for the Streamfunction Poisson equation using Pure MPI. Unlike the Crank-Nicolson iterative scheme, the FFT takes no initial guess so the computation time is independent of the final solution. We see optimal scaling for all problem sizes, however this decays rapidly when the number of processors becomes significant relative to `N`. At this point, the ratio of communication time to computational time becomes non-trivial.

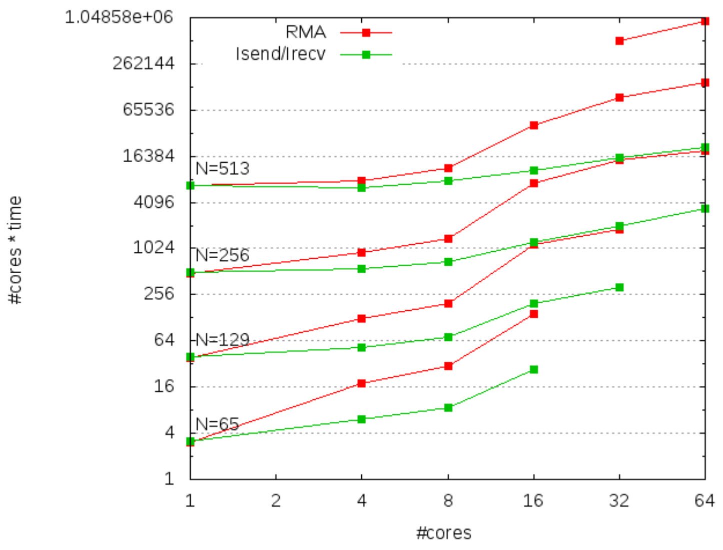


Figure 20: Scaling of the Non-blocking and RMA MultiGrid solvers for the Streamfunction Poisson equation using Pure MPI. This algorithm has no comparison to the Gauss-Seidel or the FFT. The performance literally cannot be compared as the MultiGrid tests were done on a different architecture to the others, as was explained earlier. However the scaling of the non-blocking solver with increasing compute cores is far superior, even holding it's own at large numbers of cores, unlike the FFT whose scaling is destroyed quite rapidly as the number of cores is increased. The RMA results here mirror the analysis presented in the 'Performance' section: poor scaling regardless of number of compute cores.

6.3 MPI Strong Scaling

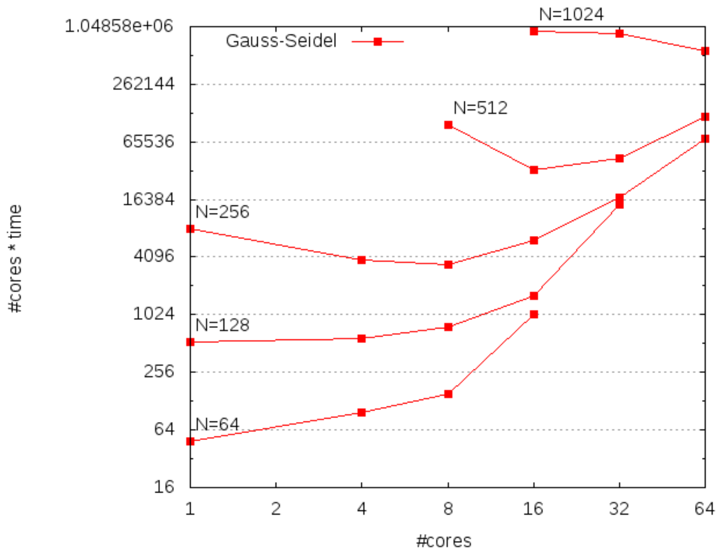


Figure 21: Pure MPI strong scaling of total solver using Gauss-Seidel/Crank-Nicolson

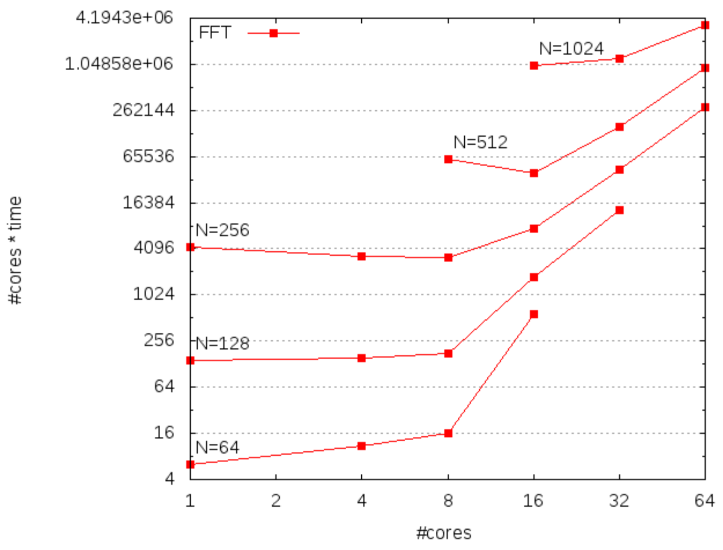


Figure 22: Pure MPI strong scaling of total solver using FFT/Crank-Nicolson

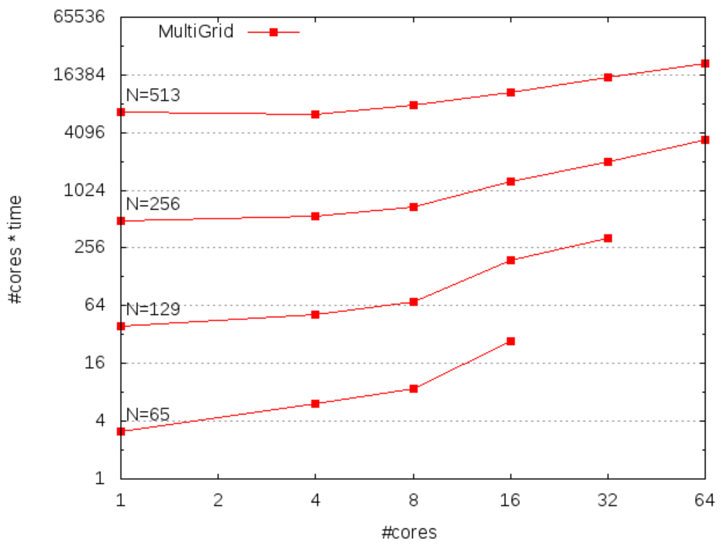


Figure 23: Pure MPI strong scaling of total solver using MultiGrid/Crank-Nicolson

6.4 Hybrid Strong Scaling

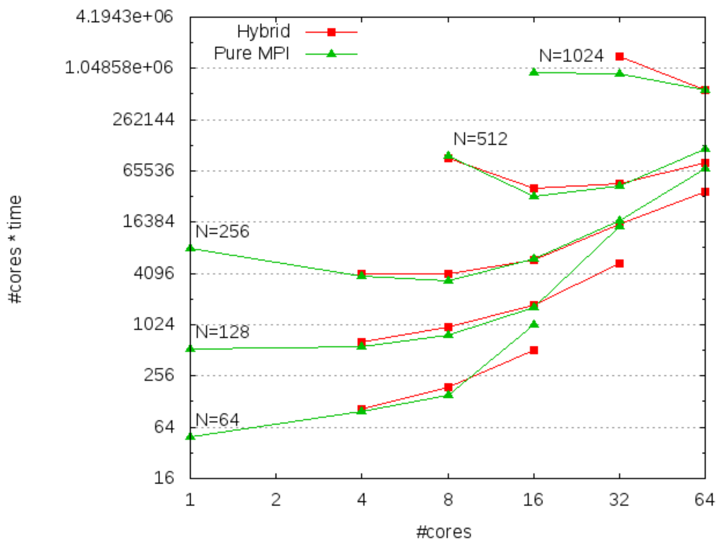


Figure 24: Strong scaling of total solver using Gauss-Seidel - Hybrid/Pure MPI comparison

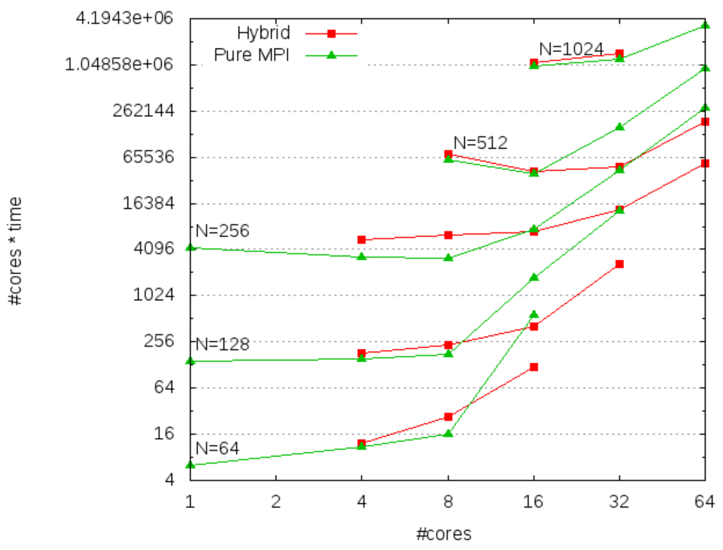


Figure 25: Strong scaling of total solver using FFT - Hybrid/Pure MPI comparison

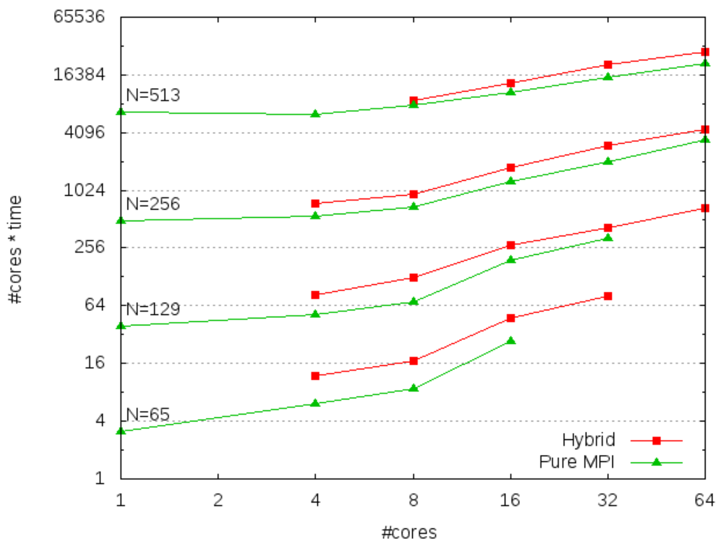


Figure 26: Strong scaling of total solver using MultiGrid - Hybrid/Pure MPI comparison

7 Conclusions & Future Work

The numerical accuracy of the developed software is excellent, with results agreeing very well with Ghia's for small Reynolds numbers (~ 100). For $Re = 400$, numerical inaccuracies appear as the coarseness of the mesh is exploited. For these problems where the inertial forces begin to dominate, the problem would likely be better tackled by a non-uniform mesh solution.

The MultiGrid algorithm likely has some potential for better scaling than was achieved here. Due to the 'AMD vs Intel' architecture issue, most of the analysis and development of the parallel MultiGrid occurred in the final days of the project. Yet it still outperformed every other algorithm for all tested configurations. The peculiar behaviour on Intel architecture also warrants further investigation. Overall the MultiGrid proved itself as an efficient, stable, numerically accurate Poisson solver.

The hybrid MPI/OpenMP configurations performed well in all scenarios. For Gauss-Seidel and the FFT, results agree with intuition: the hybrid code implementing the same number of cores doesn't experience diminished returns until after the pure MPI configuration, resulting in improved performance for a higher core count. The MultiGrid was the only algorithm in which the hybrid configurations did *not* outperform the pure MPI configuration for any test. This also agrees with intuition as threaded parallelism is only implemented for large problem sizes. The MultiGrid algorithm is, almost by definition, a collection of small problem sizes. The program as a whole will still benefit from threads during the upper levels of MultiGrid and in the chosen vorticity solver, but it will fall short of pure MPI as long as there exist un-threadable grids. As problem size and core count increase, the number of threadable grids increases and the number of un-threadable grids falls, collectively resulting in a better performing hybrid implementation.

A non-uniform mesh approach different to the one explored in this project would be an excellent and promising place for future work. An iterative algorithm such as this would not only receive the benefit of the initial guess that Gauss-Seidel does, but the rapid convergence of Gauss-Seidel when at equilibrium would become an ultra-rapid convergence as the number of mesh points drops in a near-static system - all while maintaining the numerical accuracy of a globally fine mesh. For low Reynolds numbers such as were investigated in this project, a second order accurate non-uniform mesh technique would be hard pressed to find regions of discontinuity requiring a high density of mesh points. On the other hand, for high Reynolds numbers, inertial forces dominate producing turbulent flows in which case the non-uniform mesh would converge to the solution more rapidly and more accurately (compared to, say, a Gauss-Seidel solver) as a finer mesh is applied to the chaotic, high-curvature regions of the solution space only.

It is not unthinkable either to create a non-uniform adaptive mesh software which, each timestep, is solved via MultiGrid approach.

8 Bibliography

References

[1] James W. Demmel *Applied Numerical Linear Algebra* 2009

[2] Matteo Frigo & Steven G. Johnson *The design and implementation of FFTW3, Proc. IEEE 93 (2), 216231* 2005

[3] Nikolaos Sfakianakis *Finite Difference Schemes on non-Uniform Meshes for Hyperbolic Conservation Laws* 2009

[4] U. Ghia, K. N. Ghia & C. T. Shin *High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method* 1982

[5] Yelick & Demmel *CS 267, UC Berkeley* 2011