View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

A Linear Multigrid Preconditioner for the solution of the Navier-Stokes Equations using a Discontinuous Galerkin Discretization

by

Laslo Tibor Diosady

B.A.Sc., University of Toronto (2005)

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author . . . .
School of Engineering
May 25, 2007

Certified by . . . .
David Darmofal
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . .
Jaime Peraire
Professor of Aeronautics and Astronautics
Co-Director, Program in Computation for Design and Optimization

A Linear Multigrid Preconditioner for the solution of the Navier-Stokes Equations using a Discontinuous Galerkin Discretization

by

Laslo Tibor Diosady

Submitted to the School of Engineering
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computation for Design and Optimization

Abstract

A Newton-Krylov method is developed for the solution of the steady compressible Navier-Stokes equations using a Discontinuous Galerkin (DG) discretization on unstructured meshes. An element Line-Jacobi preconditioner is presented which solves a block tridiagonal system along lines of maximum coupling in the flow. An incomplete block-LU factorization (Block-ILU(0)) is also presented as a preconditioner, where the factorization is performed using a reordering of elements based upon the lines of maximum coupling used for the element Line-Jacobi preconditioner. This reordering is shown to be far superior to standard reordering techniques (Nested Dissection, One-way Dissection, Quotient Minimum Degree, Reverse Cuthill-Mckee) especially for viscous test cases. The Block-ILU(0) factorization is performed in-place and a novel algorithm is presented for the application of the linearization which reduces both the memory and CPU time over the traditional dual matrix storage format. A linear p-multigrid algorithm using element Line-Jacobi, and Block-ILU(0) smoothing is presented as a preconditioner to GMRES. The coarse level Jacobians are obtained using a simple Galerkin projection which is shown to closely approximate the linearization of the restricted problem except for perturbations due to artificial dissipation terms introduced for shock capturing. The linear multigrid preconditioner is shown to significantly improve convergence in terms of the number of linear iterations as well as to reduce the total CPU time required to obtain a converged solution. A parallel implementation of the linear multigrid preconditioner is presented and a grid repartitioning strategy is developed to ensure scalable parallel performance.

Thesis Supervisor: David Darmofal
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank all those who helped make this work possible. First, I would like to thank my advisor, Prof. David Darmofal, for giving me the opportunity to work with him. I am grateful for his guidance, inspiration and encouragement, and I look forward to continuing my studies under his supervision. In addition, I would like to thank the Project X team (Bob, Chris, Garrett, Todd, Mike, JM, David and Josh). This work would not have been possible without their help. Finally, I would like to thank Laura, my parents and my brother for their support and encouragement.

This work was partially supported by funding from The Boeing Company with technical monitor Dr. Mori Mani.

Contents

1 Introduction 13

2 Solution Method 17

2.1 DG Discretization 17

2.2 Linear System 20

2.3 Linear Solution Method 21

2.4 Residual Tolerance Criterion 21

3 In-Place Preconditioning 25

3.1 Stationary Iterative Methods 25

3.2 Block-Jacobi Solver 27

3.3 Line-Jacobi Solver 28

3.4 Block-ILU Solver 29

3.5 Timing Performance 34

3.6 In-place ILU Factorization of General Matrices 36

4 ILU Reordering 37

4.1 Line Reordering 38

4.2 Numerical Results 39

4.3 In-place ILU Storage Format 40

5 Linear Multigrid 43

5.1 Linear Multigrid Algorithm 43

5.2 Coarse Grid Jacobian 45

5.2.1 Stabilization Terms 47
5.3 Memory Considerations 49
5.4 Numerical Results 51
5.4.1 Inviscid Transonic flow over NACA0012, $M = 0.75$, $\alpha = 2°$ . . 51
5.4.2 Viscous Subsonic flow over NACA0012, $M = 0.5$, $\alpha = 0°$, $Re = 1000$ 53

6 Parallel Performance **57**
6.1 Parallel Implementation 57
6.2 Parallel Preconditioner Implementation 57
6.3 Grid Repartitioning 58
6.4 Numerical Results 60
6.4.1 Linear multigrid preconditioner with Line-Jacobi smoothing . 61
6.4.2 Linear multigrid preconditioner with Block-ILU smoothing . . 63

7 Conclusions **67**

List of Figures

4-1 Non-linear residual vs linear iterations using the Block-ILU(0) preconditioner with different reordering techniques for a transonic Euler solution of the flow about the NACA0012 airfoil 40
4-2 Non-linear residual vs linear iterations using the Block-ILU(0) preconditioner with different reordering techniques for a Navier-Stokes solution of the flow about the NACA0012 airfoil 41

5-1 Eigenvalue analysis of NACA0012 subsonic viscous test 47
5-2 Eigenvalue analysis of the transonic, $M = 0.75$, NACA0012 test case 48
5-3 Eigenvalue analysis of the hypersonic, $M = 11$, shock ramp test case . 49
5-4 Convergence plots of the inviscid transonic NACA0012 test case . . . 53
5-5 Convergence plots of the viscous subsonic NACA0012 test case 55

6-1 Different grid partitionings for the viscous NACA0012 test case . . . 60
6-2 Parallel performance of the linear multigrid preconditioner with Block-Jacobi smoothing for the inviscid transonic NACA0012 test case . . . 61
6-3 Parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing inviscid transonic NACA0012 test case 62
6-4 Parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing viscous subsonic NACA0012 case 63
6-5 Parallel performance of the linear multigrid preconditioner with Block-ILU smoothing NACA0012 Navier-Stokes test case 64
6-6 Parallel performance of the linear multigrid preconditioner with Block-ILU smoothing viscous subsonic NACA0012 case 65

6-7 Parallel timing results 66

List of Tables

2.1 Number of modes per element, n_m, as a function of solution order, p . 20

3.1 Block-Jacobi solver asymptotic operation count per element 27

3.2 Line-Jacobi solver asymptotic operation count per element 29

3.3 Block-ILU solver asymptotic operation count per element 34

3.4 Block-ILU solver asymptotic operation count per element 35

3.5 Linear iteration asymptotic operation count per element (in multiples of n_b^2) 36

4.1 Revised Timing Estimate for Application of N for in-place Block-ILU(0) 42

5.1 Additional memory usage for lower order Jacobians for linear multigrid 50

5.2 Additional memory usage for lower order Jacobians for linear multigrid solver in terms of solution vectors 51

5.3 Convergence results of the inviscid transonic NACA0012 test case . . 52

5.4 Convergence results of the viscous subsonic NACA0012 test case . . . 54

Chapter 1

Introduction

Discontinuous Galerkin (DG) discretizations have become increasingly popular for achieving accurate solutions of conservation laws. Specifically, DG discretizations have been widely used to solve the Euler and Navier-Stokes equations for convection dominated problems [6, 7, 8, 13, 14, 5]. DG methods are attactive since the element-wise discontinuous representation of the solution provides a natural way of achieving higher-order accuracy on arbitrary, unstructured meshes. A detailed overview of DG methods for the discretization of the Euler and Navier-Stokes equations is provided by Cockburn and Shu [14]. They, among others [19, 29], have noted that while DG discretizations have been extensively studied, development of solution methods ideally suited for solving these discretizations have lagged behind. In this work a Newton-GMRES approach using a linear multigrid preconditioner is developed as a solution method for DG discretizations of the steady Navier-Stokes equations.

The use of multigrid to solve DG discretizations of compressible flows was first presented by Fidkowski [17] and Fidkowski et al. [19]. Fidkowski et al. used a p-multigrid scheme with an element-line smoother to solve the non-linear system of equations. The Newton-GMRES approach has been widely used for finite volume discretizations of the Euler and Navier-Stokes equations [1, 12, 11, 36, 25, 22, 30]. In the context of DG discretizations, GMRES was first used to solve the steady 2D compressible Navier-Stokes equations by Bassi and Rebay [8, 9]. GMRES has also been used for the solution of the linear system arising at each iteration of an implicit time stepping scheme for the DG discretization of the time dependent Euler or Navier-Stokes equations [37, 16, 34]. Persson and Peraire [34] developed a two level scheme as a preconditioner to GMRES to solve the linear system at each step of an implicit time stepping scheme. They used an ILU(0) smoother for the desired p and solved a coarse grid problem ($p = 0$ or $p = 1$) exactly. Recently, several other authors have used p-multigrid methods to solve DG discretizations of the Euler or Navier-Stokes equations [20, 29, 15, 23]. Natase and Mavriplis [29, 15] used both p-multigrid (where coarse solutions are formed by taking lower order approximations within each element), and hp-multigrid, where an h-multigrid scheme was used to provide a solution update for the $p = 0$ approximation. Natase and Mavriplis used this hp-multigrid scheme with an element Block-Jacobi smoother to solve the non-linear system as well as to solve the linear system arising from a Newton scheme.

This work presents a linear p-multigrid scheme as a preconditioner to GMRES for the solution of the steady state Euler and Navier-Stokes equations using a DG discretization. While results presented here are used to solve steady state problems, the methods are also suitable for solving time dependent problems. An overview of the DG discretization and the Newton-Krylov approach for solving systems of non-linear conservation laws is presented in Chapter 2. Chapter 3 presents the Block-Jacobi, Line-Jacobi and Block-ILU(0) stationary iterative methods that are used as single-level preconditioners or as smoothers on each level of the linear multigrid preconditioner. By considering the Block-ILU preconditioner as a stationary iterative method, a memory efficient implementation is developed which requires no additional storage for the incomplete factorization, while reducing the total time required per linear iteration compared to the traditional dual matrix storage format. Chapter 4 presents a new matrix reordering algorithm for the Block-ILU factorization based upon lines of maximum coupling between elements in the flow. This line reordering algorithm is shown to significantly improve the convergence behaviour, especially for viscous problems. Chapter 5 presents a linear multigrid algorithm where the coarse level Jacobians are formed using a simple Galerkin projection. The Galerkin projection is shown to be nearly equivalent to the linearization of a restricted discretization, except in the case of strong shocks where artificial dissipation terms introduced for shock capturing add an h/p dependence to the governing equations. Chapter 5 also presents numerical results which show that the linear multigrid algorithm reduces both the number of linear iterations and the time required to obtain a converged solution. Finally, Chapter 6 presents the parallel implementation of the linear multigrid preconditioner and discusses additional considerations required for developing a scalable parallel preconditioning algorithm.

Chapter 2

Solution Method

2.1 DG Discretization

The time dependent, compressible Navier-Stokes equations using index notation are given by:

$$\partial_t u_k + \partial_i F_{ki}(\mathbf{u}) - \partial_i F^v_{ki}(\mathbf{u}) \quad = \quad 0, \qquad k \in [1, n_s] \tag{2.1}$$

where u_k is the k^{th} component of the conservative state vector $\mathbf{u} = [\rho, \rho v_i, \rho E]$, ρ is the density, v_i are the component of the velocity, and E is the total energy. For two- and three- dimensional flows, $n_s = 4$ and 5, respectively (assuming turbulence modeling or other equations are not included). $F_{ki}(\mathbf{u})$ and $F^v_{ki}(\mathbf{u})$ are inviscid and viscous flux components, respectively, such that Equation (2.1) is a compact notation for the conservation of mass, momentum, and energy.

The DG discretization of the Navier-Stokes equations is obtained by choosing a triangulation T_h of the computational domain Ω composed of triangular elements κ, and obtaining a solution in $\mathcal{V}^p_h$, the space of piecewise polynomials of order p, which satisfies the weak form of the equation. We define $\mathbf{u}_h$ to be the approximate solution in $(\mathcal{V}^p_h)^{n_s}$, while $\mathbf{v}_h \in (\mathcal{V}^p_h)^{n_s}$ is an arbitrary test function. The weak form is obtained by multiplying Equation (2.1) by the test functions and integrating over all elements.

The weak form is given by

$$\sum_{\kappa \in T_h} \int_k v_k \partial_t u_k dx + \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_h) = 0, \tag{2.2}$$

where,

$$\begin{aligned} \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_h) &= \sum_{\kappa \in T_h} \left[\mathbb{E}_\kappa(\mathbf{u}_h, \mathbf{v}_h) + \mathbb{V}_\kappa(\mathbf{u}_h, \mathbf{v}_h) \right] && (2.3) \\ \mathbb{E}_\kappa(\mathbf{u}_h, \mathbf{v}_h) &= -\int_\kappa \partial_i v_k F_{ki} dx + \int_{\partial\kappa} v_k^+ \hat{F}_{ki}(\mathbf{u}_h^+, \mathbf{u}_h^-) \hat{n}_i ds && (2.4) \end{aligned}$$

and $\mathbb{V}_\kappa(\mathbf{u}_h, \mathbf{v}_h)$ is the discretization of the viscous terms. In Equation (2.4), $()^+$ and $()^-$ denote values taken from the inside and outside faces of an element, while $\hat{n}$ is the outward-pointing unit normal. $\hat{F}_{ki}(\mathbf{u}_h^+, \mathbf{u}_h^-)$ is the Roe numerical flux function approximating F_{ki} on the element boundary faces [38]. The viscous terms, $\mathbb{V}_\kappa(\mathbf{u}_h, \mathbf{v}_h)$ are discretized using the BR2 scheme of Bassi and Rebay [8]. The BR2 scheme is used because it achieves optimal order of accuracy while maintaining a compact stencil with only nearest neighbour coupling. Further details of the discretization of the viscous terms may be found in Fidkowski et al [19].

The discrete form of the equations is obtained by choosing a basis for the space $\mathcal{V}_h^p$. The solution vector $\mathbf{u}_h(x,t)$ may then be expressed as a linear combination of basis functions $\mathbf{v}_{h_i}(x)$ where the coefficients of expansion are given by the discrete solution vector $\mathbf{U}_h(t)$, such that:

$$\mathbf{u}_h(x,t) = \sum_i \mathbf{U}_{h_i}(t) \mathbf{v}_{h_i}(x) \tag{2.5}$$

Two sets of basis functions are used in the context of this work: a nodal Lagrange basis and a hierarchical basis. Further details of the bases may be found in Fidkowski et al [19].

Having defined a basis for the space $\mathcal{V}_h^p$ the weak form of the Navier-Stokes equations given in Equation (2.2) can be written in semi-discrete form as:

$$\mathcal{M}_h \frac{d\mathbf{U}_h}{dt} + R_h(\mathbf{U}_h(t)) \quad = \quad 0, \tag{2.6}$$

where R_h is the discrete non-linear residual such that $R_h(\mathbf{U}_h)_i = \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{h_i})$, while $\mathcal{M}_h$ is the mass matrix given by

$$\mathcal{M}_{h_{ij}} = \int_\kappa \mathbf{v}_{h_i} \mathbf{v}_{h_j} dx. \tag{2.7}$$

Since the basis functions are piecewise polynomials which are non-zero only within a single element, the mass matrix is block-diagonal.

In order to discretize Equation (2.6) in time, we introduce a time integration scheme given by:

$$\mathbf{U}_h^{m+1} \quad = \quad \mathbf{U}_h^m - \left(\frac{1}{\Delta t} \mathcal{M}_h + \frac{\partial R_h}{\partial \mathbf{U}_h} \right) R_h(\mathbf{U}_h^m) \tag{2.8}$$

To obtain a steady state solution of the Navier-Stokes equations we seek a solution $\mathbf{U}_h$ satisfying:

$$R_h(\mathbf{U}_h) \quad = \quad 0 \tag{2.9}$$

The steady state solution is obtained by using the time integration scheme given in Equation (2.8) and increasing the time step Δt, such that $\Delta t \to \infty$. Directly setting $\Delta t = \infty$ is the equivalent of using Newton's method to solve Equation (2.9), however convergence is unlikely if the initial guess is far from the solution. On the other hand, if the solution is updated using Equation (2.8), then the intermediate solutions represent physical states in the time evolution of the flow, and convergence is more likely.

2.2 Linear System

The time integration scheme given by Equation (2.8) requires the solution of a large system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ at each time step, where

$$A = \frac{1}{\Delta t}\mathcal{M}_h + \frac{\partial R_h}{\partial \mathbf{U}_h} \qquad \mathbf{x} = \Delta \mathbf{U}_h^m \qquad \mathbf{b} = -R_h(\mathbf{U}_h^m). \tag{2.10}$$

The matrix A is commonly refered to as the Jacobian matrix. Since the Jacobian matrix is derived from the DG discretization, it has a special structure which may be taken advantage of when solving the linear system. The Jacobian matrix has a block-sparse structure with N_e block rows of size n_b, where N_e is the number of elements in the triangulation T_h, while n_b is the number of unknowns for each element. Here $n_b = n_s \times n_m$, where n_m is the number of modes per state. n_m is a function of the solution order p and the spatial dimension, as summarized in Table 2.1. Each block row of the Jacobian matrix has a non-zero diagonal block, corresponding to the coupling of states within each element, and n_f off-diagonal non-zero blocks corresponding to the coupling of states between neighbouring elements, where n_f is the number of faces per element (3 and 4 for 2D triangular and 3D tetrahedral elements, respectively). When the time step, Δt, is small, the Jacobian matrix is block-diagonally dominant and the linear system is relatively easy to solve iteratively. On the other hand as the time step increases the coupling between neighbouring elements becomes increasingly important and the linear system generally becomes more difficult to solve.

p	n_m, 2D	n_m, 3D
0	1	1
1	3	4
2	6	10
3	10	20
4	15	35
p	$\frac{(p+1)(p+2)}{2}$	$\frac{(p+1)(p+2)(p+3)}{6}$

Table 2.1: Number of modes per element, n_m, as a function of solution order, p

2.3 Linear Solution Method

The block-sparse structure of the Jacobian matrix and the large number of unknowns suggest the use of an iterative method, more specifically a Krylov-subspace method, to solve the linear system. Since the Jacobian matrix is non-symmetric (though structurally symmetric), the method of choice is the restarted GMRES [39, 40] algorithm which finds an approximate solution, $\tilde{\mathbf{x}}$, in the Krylov subspace, $\mathcal{K} = \{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, ...A^{k-1}\mathbf{b}\}$, that minimizes the L-2 norm of the linear residual $\mathbf{r} = \mathbf{b} - A\tilde{\mathbf{x}}$.

The convergence of the GMRES algorithm has been shown to be strongly dependent upon eigenvalues of the Jacobian matrix, A [39, 40, 42]. In order to improve the convergence properties of GMRES, a preconditioner must be used which transforms the linear system $A\mathbf{x} = \mathbf{b}$ into a related system $P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}$ with better convergence properties. Though the preconditioner, P, is presented as a matrix, any iterative method may be used as a preconditioner. A goal of this work is to develop effective preconditioning techniques for DG discretizations of convection-dominated flows which result in fast convergence of the GMRES algorithm in terms of number of iterations as well as computational effort.

2.4 Residual Tolerance Criterion

When solving the DG discretization of the steady-state Navier-Stokes equations using the time stepping scheme presented in Equation (2.8), it is often unnecessary to solve the linear system of equations exactly at each iteration. When the time step is small, or the solution estimate is far from the exact solution, the linear system only needs to be solved to a limited tolerance, which depends upon the non-linear residual. Kelley and Keyes [21] considered three phases of a time stepping scheme in order to solve the steady state Euler equations: the initial, midrange, and terminal phases. Kelley and Keyes proved super-linear convergence of the non-linear residual in the terminal phase of an inexact Newton iteration given sufficient reduction of the linear residual in each iteration. In this section an exit criterion is developed for the solution of the linear system in order to realize the super-linear convergence during the terminal phase. In order to be able to develop an exit criterion for the solution of the linear system we consider the convergence of Newton's method to solve Equation (2.9) . The solution update is given by:

$$\mathbf{U}_h^{m+1} = \mathbf{U}_h^m - \left(\frac{\partial R_h}{\partial \mathbf{U}_h}\right)^{-1} R_h(\mathbf{U}_h^m), \tag{2.11}$$

where $\mathbf{U}_h^m$ is the approximate solution at iteration m of the Newton's method. Defining $\epsilon_h^m = \mathbf{U}_h - \mathbf{U}_h^m$ to be the solution error at iteration m, quadratic convergence of the error can be proven as $\epsilon_h^m \rightarrow 0$. Namely,

$$\left|\left|\epsilon_h^{m+1}\right|\right| \quad = \quad C_1 \left|\left|\epsilon_h^m\right|\right|^2, \tag{2.12}$$

for some constant C_1[21]. Similarly quadratic convergence of the solution residual is observed,

$$\left|\left|R_h(\mathbf{U}_h^{m+1})\right|\right| \quad = \quad C_2 \left|\left|\mathcal{R}_h(\mathbf{U}_h^m)\right|\right|^2, \tag{2.13}$$

for some different constant C_2. Based on this observation, an estimate of the reduction in the solution residual may be given by:

$$\frac{\left|\left|R_h(\mathbf{U}_h^{m+1})\right|\right|}{\left|\left|R_h(\mathbf{U}_h^m)\right|\right|} \sim \left(\frac{\left|\left|R_h(\mathbf{U}_h^m)\right|\right|}{\left|\left|R_h(\mathbf{U}_h^{m-1})\right|\right|}\right)^2 = (d^m)^2, \tag{2.14}$$

where $d^m = \frac{\left|\left|R_h(\mathbf{U}_h^m)\right|\right|}{\left|\left|R_h(\mathbf{U}_h^{m-1})\right|\right|}$, is the decrease factor of the non-linear residual at iteration m. When the expected decrease of the non-linear residual is small, it may not be necessary to solve the linear system at each Newton step exactly in order to get an adequate solution update. It is proposed that the linear system given by $A_h\mathbf{x}_h = \mathbf{b}_h$ should have a reduction in linear residual proportional to the expected decrease in the non-linear residual. Defining the linear residual at linear iteration k to be $\mathbf{r}_h^k =$

$\mathbf{b}_h - A_h\mathbf{x}_h^k$, the linear system is solved to a tolerance of:

$$\frac{||\mathbf{r}_h^n||}{||\mathbf{r}_h^0||} \leq K(d^m)^2, \tag{2.15}$$

where K is a user defined constant, typically chosen in the range $k = [10^{-3}, 10^{-2}]$. Since the non-linear residual or the decrease factor may increase at some iteration m, the tolerance for the linear system presented in Equation (2.15) is modified to be:

$$\frac{||\mathbf{r}_h^n||}{||\mathbf{r}_h^0||} \leq K\left(\min\left\{1, d^m\right\}\right)^2. \tag{2.16}$$

Since the linear residual in the 2-norm is not available at each GMRES iteration and computing this linear residual can be computationally expensive, the preconditioned linear residual is used, which can be computed essentially for free at each GMRES iteration. This criterion for the reduction of the linear residual is then used to determine n, the number of GMRES iterations to perform each Newton step.

Chapter 3

In-Place Preconditioning

3.1 Stationary Iterative Methods

Stationary iterative methods used to solve the system of linear equations $A\mathbf{x} = \mathbf{b}$ involve splitting the matrix A into two parts such that $A = M + N$, where M in some sense approximates the matrix A and is relatively easy to invert. Since an iterative scheme is typically used directly as a preconditioner to GMRES, M is commonly refered to as the preconditioning matrix. The principle behind stationary iterative techniques is to rearrange the system of equation to solve for $\mathbf{x}$:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (M+N)\mathbf{x} &= \mathbf{b} \\ M\mathbf{x} &= \mathbf{b} - N\mathbf{x} \\ \mathbf{x} &= M^{-1}(\mathbf{b} - N\mathbf{x}) \qquad (3.1) \end{aligned}$$

In this form, $\mathbf{x}$ may be updated iteratively as

$$\mathbf{x}^{k+1} = M^{-1}(\mathbf{b} - N\mathbf{x}^{k}). \qquad (3.2)$$

An equivalent form of Equation (3.2) is

$$\mathbf{x}^{k+1} = \mathbf{x}^k + M^{-1}\mathbf{r}^k, \tag{3.3}$$

where $\mathbf{r}^k$ is the linear residual given by

$$\begin{aligned} \mathbf{r}^k &= \mathbf{b} - (M+N)\mathbf{x}^k \\ &= \mathbf{b} - A\mathbf{x}^k. \qquad (3.4) \end{aligned}$$

In this form it is also convenient to introduce the concept of under-relaxation, where only a fraction of the solution update is taken at each iteration:

$$\begin{aligned} \mathbf{x}^{k+1} &= \mathbf{x}^k + \omega M^{-1}\mathbf{r}^k \qquad (3.5) \\ &= (1-\omega)\mathbf{x}^k + \omega M^{-1}(\mathbf{b} - N\mathbf{x}^k). \qquad (3.6) \end{aligned}$$

In practice, stationary iterative methods involve a preprocessing stage and an iterative stage. The iterative stage involves repeated solution updates according to Equation (3.5) or Equation (3.6), where Equation (3.6) is used if the application of N is computationally less expensive than the application of A, otherwise Equation (3.5) is used. In addition, if the stationary iterative method is used as a smoother for linear multigrid, then the iterative stage will involve repeated calculation of the linear residual, $\mathbf{r}$, using Equation (3.4). In the preprocessing stage the matrix A is factorized such that the application of M^{-1}, M, N and A in Equations (3.4), (3.5), and (3.6) may be evaluated at a fraction of the computational cost of the preprocessing stage. In our implementation, the preprocessing stage is performed in place such that the original matrix A is rewritten with a factorization of M. As a result the iterative method uses only the memory required to store the original matrix A, with no additional memory storage required for M, M^{-1} or N.

3.2 Block-Jacobi Solver

The first and most basic stationary iterative method used in this work is a Block-Jacobi solver. The Block-Jacobi solver is given by choosing M to be the block-diagonal of the matrix A, where each block is associated with the coupling between the states within each element, while the coupling between elements is ignored. In the preprocessing stage each diagonal block is LU factorized and the factorization, F, is stored, where

$$F = \begin{bmatrix} LU(A_{11}) & A_{12} & A_{13} \\ A_{21} & LU(A_{22}) & A_{23} \\ A_{31} & A_{32} & LU(A_{33}) \end{bmatrix}. \tag{3.7}$$

This factorization allows for the easy application of both M and M^{-1} during the iterative stage. N is given by the off-diagonal blocks of A which are not modified in the preprocessing stage. Table 3.1 gives the asymptotic operation counts per element for forming F (given A), as well as the application of M^{-1}, M, N and A. The operation counts presented in Table 3.1 are asymptotic estimates, in that lower order terms in n_b have been ignored. The application of A is computed as the sum of the applications of M and N. Since the application of A is computationally more expensive than the application of N, the Block-Jacobi iterative step uses Equation (3.6).

Operation	Operation Count	2D	3D
Form F	$\frac{2}{3}n_b^3$	$\frac{2}{3}n_b^3$	$\frac{2}{3}n_b^3$
$x = M^{-1}x$	$2n_b^2$	$2n_b^2$	$2n_b^2$
$y = Mx$	$2n_b^2$	$2n_b^2$	$2n_b^2$
$y = Nx$	$2n_f n_b^2$	$6n_b^2$	$8n_b^2$
$y = Ax$	$2(n_f + 1)n_b^2$	$8n_b^2$	$10n_b^2$

Table 3.1: Block-Jacobi solver asymptotic operation count per element

3.3 Line-Jacobi Solver

The second stationary iterative method presented in this work is a Line-Jacobi solver. The Line-Jacobi solver is given by forming lines of maximum coupling between elements and solving a block-tridiagonal system along each line. The coupling between elements is determined by using a $p = 0$ discretization of the scalar transport equation:

$$\nabla \cdot (\rho u \phi) - \nabla \cdot (\mu \nabla \phi) = 0$$

The lines are formed by connecting neighbouring elements with maximum coupling. For purely convective flows, the lines are in the direction of streamlines in the flow. For viscous flows solved using anisotropic grids, the lines within the boundary layer often are in non-streamline directions. Further details of the line formation algorithm are presented in the theses of Fidkowski [17] and Oliver [33].

Using the notation previously presented for stationary iterative methods, M is given by the block-tridiagonal systems corresponding to the lines of maximum coupling, while N is given by the blocks associated with the coupling between elements across different lines. In the preprocessing stage M is factorized using a block-variant of the Thomas algorithm given by:

$$F = \begin{bmatrix} LU(A_{11}) & A_{12} & A_{13} \\ A_{21} & LU(A'_{22}) & A_{23} \\ A_{31} & A_{32} & LU(A'_{33}) \end{bmatrix} \tag{3.8}$$

where, $A'_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and $A'_{33} = A_{33} - A_{32}A'^{-1}_{22}A_{23}$. The corresponding LU factorization of M is given by:

$$M = \begin{bmatrix} A_{11} & A_{12} & \\ A_{21} & A_{22} & A_{23} \\ & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} I & & \\ A_{21}A_{11}^{-1} & I & \\ & A_{32}A'^{-1}_{22} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & \\ & A'_{22} & A_{23} \\ & & A'_{33} \end{bmatrix} \tag{3.9}$$

The factorization given by Equation (3.8) is stored as opposed to the LU factorization given by Equation (3.9) in order to reduce the computational cost of the preprocessing stage. The reduction in computational cost of storing the factorization given by Equation (3.8) is offset by an increase in the computational cost of applying M and M^{-1} during the iterative stage. The total computational cost for both the preprocessing and iterative stages using the factorization given by Equation (3.8) is lower than the LU factorization given by Equation (3.9), as long as the total number of linear iterations is less than the block size, n_b.

Table 3.2 gives the asymptotic operation counts per element for the preprocessing stage as well as the application of M^{-1}, M, N and A. The application of A is once again computed as a sum of the applications of M and N. As with the Block-Jacobi solver, the application of N is computationally less expensive than the application of A, so that solution update for the Line-Jacobi solver is given by Equation (3.6).

Operation	Operation Count	2D	3D
Form F	$\frac{14}{3}n_b^3$	$\frac{14}{3}n_b^3$	$\frac{14}{3}n_b^3$
$x = M^{-1}x$	$8n_b^2$	$8n_b^2$	$8n_b^2$
$y = Mx$	$8n_b^2$	$8n_b^2$	$8n_b^2$
$y = Nx$	$2(n_f - 2)n_b^2$	$2n_b^2$	$4n_b^2$
$y = Ax$	$2(n_f + 2)n_b^2$	$10n_b^2$	$12n_b^2$

Table 3.2: Line-Jacobi solver asymptotic operation count per element

3.4 Block-ILU Solver

The final iterative method presented in this work is a block incomplete-LU factorization (Block-ILU). ILU factorizations have been successfully used as preconditioners for a variety of aerodynamic problems [1, 11, 36, 25, 22, 34, 30]. Typically the LU factorization of a sparse matrix will have a sparsity pattern with significantly more non-zeros, or ”fill”, than the original matrix. The principle of an incomplete-LU factorization is to produce an approximation of the LU factorization of A, which requires significantly less fill than the exact LU factorization. The incomplete LU factorization, $\tilde{L}\tilde{U}$, is computed by performing Gaussian elimination on A but ignoring values which would result in additional fill. The fill level, k, indicates the distance in the sparsity graph of the neighbours in which coupling may be introduced in the ILU(k) factorization. In the context of this work ILU(0) is used, hence no additional fill outside the sparsity pattern of A is permitted. In order to simplify the notation, for the remainder of this work we use ILU to denote an ILU(0) factorization unless explicitly stated otherwise.

Though incomplete-LU factorizations are widely used, most implementations require the duplicate storage of both the linearization A, and the incomplete factorization, $\tilde{L}\tilde{U}$. Since in most aerodynamic applications the majority of the memory is used for the storage of the linearization and its factorization, such duplicate memory storage may limit the size of the problems which may be solved on a given machine [11, 27, 34]. In this section an algorithm is developed enabling the incomplete-LU factorization to be performed in-place, such that no additional memory is required for the storage of the factorization. This in-place storage format is an enabling feature which allows for the solution of larger and more complex problems on a given machine. Assuming the majority of the memory is used for the storage of the Jacobian matrix and the Krylov vectors, the increase in the size of the problem which may be solved on a given machine is given by $\frac{2+\eta}{1+\eta}$, where η is the ratio of the memory required to store the Krylov vectors to the memory required to store the Jacobian matrix. For a typical range $\eta \in [0.1, 1.0]$, this represents an increase of 50-90% in the size of problem which may be solved.

In order to be able to develop a scheme where the memory usage is no greater than the cost of the incomplete factorization it is useful to consider the ILU factorization as a stationary iterative method. In the context of the stationary iterative methods presented previously, M is given by the product $\tilde{L}\tilde{U}$, such that $\tilde{L}\tilde{U}$ is the exact LU factorization of M. It can be easily shown that A differs from M only where fill is dropped in the incomplete LU factorization. Correspondingly, N is given by a matrix containing all fill which was ignored in the ILU factorization. To construct an in-place storage for ILU, note that both A and N may be reconstructed from $\tilde{L}\tilde{U}$ given the original sparsity pattern of A. Namely, A can be computed by taking the product $\tilde{L}\tilde{U}$ and ignoring those values not within the original sparsity pattern. Similarly N can be computed by taking the values of $-\tilde{L}\tilde{U}$ outside the sparsity pattern of A. Though recomputing A and N in this manner is possible, it is impractical since the computational cost is of the same order as the original ILU factorization and requires additional memory storage. Fortunately, only the application of A or N is required, and hence it is possible to compute these products efficiently using $\tilde{L}$ and $\tilde{U}$.

The remainder of this section describes the implementation and computational efficiency of the in-place Block-ILU. The analysis for the development of the efficient storage format for the Block-ILU solver is based on the assumption that no three elements in the computational grid all neighbour one another. While this assumption may be violated for some computational grids, such violations occur infrequently, such that the analysis based on this assumption is sufficient.

As with the Block-Jacobi and Line-Jacobi solvers, the Block-ILU solver involves a preprocessing stage and an iterative stage. In the preprocessing stage, the block incomplete-LU factorization of A is performed in-place where A is replaced by the factorization F. An example of one step of the factorization is given below:

$$\begin{bmatrix} A_{11} & & A_{13} & & A_{15} & A_{16} \\ & A_{22} & & & & \\ A_{31} & & A_{33} & & & \\ & & & A_{44} & & \\ A_{51} & & & & A_{55} & \\ A_{61} & & & & & A_{66} \end{bmatrix} \Rightarrow \begin{bmatrix} LU(A_{11}) & & A_{13} & & A_{15} & A_{16} \\ & A_{22} & & & & \\ (A_{31}A_{11}^{-1}) & & A_{33}^{'} & & & \\ & & & A_{44} & & \\ (A_{51}A_{11}^{-1}) & & & & A_{55}^{'} & \\ (A_{61}A_{11}^{-1}) & & & & & A_{66}^{'} \end{bmatrix}$$

Where $A_{33}^{'} = A_{33} - A_{31}A_{11}^{-1}A_{13}$, $A_{55}^{'} = A_{55} - A_{51}A_{11}^{-1}A_{15}$, and $A_{66}^{'} = A_{66} - A_{61}A_{11}^{-1}A_{16}$. Based on the assumption that no three elements all neighbour one another, only two of the blocks A_{ij}, A_{ik}, and A_{jk} may be non-zero for any $i \neq j \neq k$. This implies that when eliminating row i only elements A_{ji} and A_{jj}, $j \geq i$ are modified. In addition, fill is ignored at A_{jk} and A_{kj}, if elements $j, k > i$ both neighbour element i. In the general case where the assumption is violated, A_{jk} and A_{kj} are non-zero, and these terms are modified in the Block-ILU factorization such that: $A'_{jk} = A_{jk} - A_{ji}A_{ii}^{-1}A_{ik}$ and $A'_{kj} = A_{kj} - A_{ki}A_{ii}^{-1}A_{ij}$. The number of non-zero blocks in the matrix N is given by $\sum_{i=1}^{N_e} \tilde{n}_{f_i}(\tilde{n}_{f_i} - 1)$ where, $\tilde{n}_{f_i}$ is the number of larger ordered neighbours of element i. While the number of non-zero blocks is dependent upon the ordering of the elements in the ILU factorization, it is possible to obtain an estimate by assuming an ordering exists where, $\tilde{n}_{f_i} = \lceil \frac{i}{N_e} n_f \rceil$. The corresponding estimate for the number of non-zero blocks in N is $N_e(n_f^2 - 1)/3$.

In the iterative stage the application of M^{-1} can be easily performed using backward and forward substitution using $\tilde{L}$ and $\tilde{U}$. The application of A is performed by multiplying by those components of $\tilde{L}$ and $\tilde{U}$ which would not introduce fill outside the original sparsity pattern of A. Similarly the application of N may be performed by multiplying by the components of $\tilde{L}$ and $\tilde{U}$ which introduce fill outside the original sparsity pattern of A.

The application of A and N is best illustrated with a simple example. Consider the 3×3 matrix A below, and the corresponding ILU factorization, $\tilde{L}\tilde{U}$:

$$A = \begin{bmatrix} 4 & 5 & -6 \\ 8 & 3 & 0 \\ -12 & 0 & 26 \end{bmatrix} \qquad \tilde{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \qquad \tilde{U} = \begin{bmatrix} 4 & 5 & -6 \\ 0 & -7 & 0 \\ 0 & 0 & 8 \end{bmatrix}$$

The corresponding matrices M, N and F are given by:

$$M = \begin{bmatrix} 4 & 5 & -6 \\ 8 & 3 & -12 \\ -12 & -15 & 26 \end{bmatrix} \qquad N = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 12 \\ 0 & 15 & 0 \end{bmatrix} \qquad F = \begin{bmatrix} 4 & 5 & -6 \\ 2 & -7 & 0 \\ -3 & 0 & 8 \end{bmatrix}$$

The application of A to a vector x, may be performed by multiplying x by those components of $\tilde{L}$ and $\tilde{U}$ which would introduce fill outside the original sparsity pattern of A. For the sample matrix, fill was ignored in the ILU factorization at (2,3) and (3,2) when eliminating row 1. Hence, for the sample matrix the application of A may be performed as follows:

$$\begin{aligned}
y_1 &= & \tilde{U}_{11}x_1 + \tilde{U}_{22}x_2 + \tilde{U}_{33}x_3 && = 4x_1 + 5x_2 - 6x_3 \\
y_2 &= & \tilde{L}_{21}\tilde{U}_{11}x_1 + \tilde{L}_{21}\tilde{U}_{12}x_2 + \cancelto{}{\tilde{L}_{21}\tilde{U}_{13}x_3} + \tilde{U}_{22}x_2 && = 2(4x_1 + 5x_2) - 7x_2 \\
y_3 &= & \tilde{L}_{31}\tilde{U}_{11}x_1 + \cancelto{}{\tilde{L}_{31}\tilde{U}_{12}x_2} + \tilde{L}_{31}\tilde{U}_{13}x_3 + \tilde{U}_{33}x_3 && = -3(4x_1 - 6x_3) + 8x_3
\end{aligned}$$

Clearly the operation count for computing the application of A in this manner is more expensive than simply applying A in the original form. However, it is important to recognize that in the case of block matrices, each of the terms $\tilde{L}_{ij}$ and $\tilde{U}_{ij}$ are matrices and x_i's are vectors, and hence the (matrix-vector) multiplications become significantly more expensive than the (vector) additions. Hence, to leading order, the computational cost is given by the number of matrix-vector multiplications. The total number of multiplications may be reduced by recognizing that certain products $(\tilde{U}_{11}x_1, \tilde{U}_{22}x_2, \tilde{U}_{33}x_3)$ are repeated. Taking advantage of the structure of the matrix A, based on the assumption that no three elements neighbour one another, it is possible to show that the application of A using $\tilde{L}\tilde{U}$ may be performed at a computational cost of $2(\frac{3}{2}n_f + 1)n_b^2 N_e$.

The application of N is performed by multiplying those components of $\tilde{L}$ and $\tilde{U}$ which would introduce fill outside the original sparsity pattern of A. For the sample matrix, fill was ignored at (2,3) and (3,2) when eliminating row 1. Hence, the application of N to a vector x may be performed as follows:

$$\begin{aligned}
y_1 &= & && = 0 \\
y_2 &= & -\tilde{L}_{21}\tilde{U}_{13}x_3 && = -2(-6x_3) = 12x_3 \\
y_3 &= & -\tilde{L}_{31}\tilde{U}_{12}x_2 && = 3(5x_2) = 15x_2
\end{aligned}$$

Once again, the computational cost is dominated by (matrix-vector) multiplications, and additional efficiency may be attained by recognizing that some products may be repeated. Since the number of elements in the matrix N is dependent upon the ordering of the elements used in the ILU factorization the operation count of applying

N cannot be determined exactly. However, it is possible to obtain an estimate for the operation count by using the same ordering of elements used to obtain the estimate of the number of elements in N. The corresponding estimate for the operation count for applying N is given by $2/3(n_f + 4)(n_f - 1)n_b^2 N_e$.

Operation	Operation Count	2D	3D
Form F	$2(n_f + 1)n_b^3$	$8n_b^3$	$10n_b^3$
$x = M^{-1}x$	$2(n_f + 1)n_b^2$	$8n_b^2$	$10n_b^2$
$y = Mx$	$2(n_f + 1)n_b^2$	$8n_b^2$	$10n_b^2$
$y = Nx$ (Estimate)	$\frac{2}{3}(n_f + 4)(n_f - 1)n_b^2$	$9\frac{1}{3}n_b^2$	$16n_b^2$
$y = Ax$	$2(\frac{3}{2}n_f + 1)n_b^2$	$11n_b^2$	$14n_b^2$
$y = Ax$ (Full Storage)	$2(n_f + 1)n_b^2$	$8n_b^2$	$10n_b^2$

Table 3.3: Block-ILU solver asymptotic operation count per element

Table 3.3 shows the asymptotic operation count per element for the preprocessing stage and components of the iterative stage for the Block-ILU solver using the in-place storage format. Note that if the Block-ILU factorization $\tilde{L}\tilde{U}$ is stored as a separate matrix such that the original matrix A is still available, the cost of computing $y = Ax$ is $2(n_f + 1)N_e n_b^2$. Based on the operation counts presented in Table 3.3, performing a linear iteration using triangular elements in 2D should be performed using Equation (3.6), since the application of N is computationally less expensive than the application of A. In 3D it appears as though a linear iteration should be performed using Equation (3.6) since the application of N is more expensive than the application of A. However, in practice the cost of an application of N is significantly less than the estimate given in Table 3.3. As a result the application of N is performed at a lower cost than the application of A and hence each linear iteration in 3D is also performed according to Equation (3.6).

3.5 Timing Performance

In order for the in-place factorization to be competitive, the cost of performing a linear iteration using the in-place factorization should not be significantly more expensive than that using duplicate matrix storage format. In the previous sections, timing estimates were presented in terms of the operations counts for the different components of each solver. In order to verify the validity of these estimates actual timing results were obtained using a sample 2D test grid with 2432 elements using a $p = 4$ discretization. The actual and estimated timing results are presented in Table 3.4 where the time has been normalized by the cost of a single matrix vector product of the Jacobian matrix. The timing estimates based on the operation counts provide a good estimate of the actual time taken; though the actual cost of operations involving LU factorized block-matrices are slightly more expensive than the estimates, which ignore operations of order n_b.

Operation	Block-Jacobi		Line-Jacobi		Block-ILU	
	Estimate	Actual	Estimate	Actual	Estimate	Actual
$x = M^{-1}x$	0.25	0.39	1.00	1.24	1.00	1.16
$y = Nx$	0.75	0.76	0.25	0.28	**1.17**	**0.51**
$y = Ax$	1.00	1.14	1.25	1.34	1.38	1.43

Table 3.4: Block-ILU solver asymptotic operation count per element

Table 3.4 also shows that the actual time to perform the application of N using the in-place storage format is less than half of the estimate. This justifies the assertion made in the previous section, that in practice the cost of the application of N is significantly less than the estimate.

Table 3.5 gives the asymptotic operation counts for the different solvers presented in this work. The in-place matrix storage format for the Block-ILU solver is at most 8% and 30% more expensive in 2D and 3D respectively. However, in practice a linear iteration of the Block-ILU solver may be performed faster using the in-place storage format than the traditional duplicate storage format, while achieving a 50% reduction in memory usage.

Preconditioner	2D	3D
Block Jacobi	8	10
Line Jacobi	10	12
Block-ILU In-Place	$17\frac{1}{3}$	26
Block-ILU Full Storage	16	20

Table 3.5: Linear iteration asymptotic operation count per element (in multiples of n_b^2)

3.6 In-place ILU Factorization of General Matrices

The in-place ILU algorithm developed in this chapter has been tailored for DG discretizations and may not be generally applicable to sparse matrices arising from other types of discretizations. While the application of A and N may be computed using the ILU factorization for any sparse matrix, the use of an in-place factorization may be unfeasible due to the number of operations required. The number of non-zero blocks in N and correspondingly, the computational cost for the application of N scales with the square of the number of off-diagonal blocks in the stencil of A. Similarly, if the assumption that no three elements neighbour one another is removed, the operation count for the application of A using the ILU factorization also scales with the square of the number of off-diagonal blocks in the stencil. The in-place ILU algorithm is feasible for DG discretizations since there is only nearest neighbour coupling, resulting in a stencil with few off-diagonal blocks. On the other hand, discretizations such as high-order finite volume discretizations have much wider stencils, involving 2nd and 3rd order neighbours[5, 30], making the in-place ILU factorization algorithm unfeasible.

Chapter 4

ILU Reordering

In the development of an efficient Block-ILU(0) preconditioner for DG discretizations, the choice for the reordering of the equations and unknowns in the linear system is critical. Matrix reordering techniques have been widely used to reduce fill in the LU factorization for direct methods used to solve large sparse linear systems [40]. These reordering techniques have also been used with iterative methods when incomplete factorizations are used as preconditioners to Krylov subspace methods [11, 36, 10, 30]. Benzi et al [10] performed numerical experiments comparing the effect of different reordering techniques on the convergence of three Krylov subspace methods used to solve a finite difference discretization of a linear convection-diffusion problem. They showed that reordering the system of equations can both reduce fill for the incomplete factorization, and improve the convergence properties of the iterative method [10]. Blanco and Zingg [11] compared Reverse Cuthill-Mckee, Nested Dissection and Quotient Minimum Degree reorderings for the ILU(k) factorization used as a preconditioner to GMRES to solve a finite volume discretization of the Euler Equations. They showed that the Reverse Cuthill-Mckee reordering reduced the fill and resulted in faster convergence for ILU(2). Similarly, Pueyo and Zingg [36] used Reverse Cuthill-Mckee reordering to reduce fill and achieve faster convergence for the finite volume discretization of the Navier-Stokes equations. In the context of ILU(0) factorizations, no additional fill is introduced, hence reordering the system of equations effects only the convergence properties of the iterative method. However, Benzi et al [10] showed that even for ILU(0), reordering the systems of equations can significantly reduce the number of GMRES iterations required to reach convergence.

The effect of several standard reordering techniques are examined in this chapter. The numerical results for the matrix reordering algorithms were determined using the PETSc package for numerical linear algebra [2, 4, 3]. The matrix reordering algorithms presented are those available in the PETSc package; namely Reverse Cuthill-Mckee, Nested-Dissection, One-Way Dissection and Quotient Minimum Degree. In addition, the "natural" ordering produced by the grid generation is employed. Finally, a new matrix reordering algorithm based on lines of maximum coupling within the flow is developed.

4.1 Line Reordering

The matrix reordering algorithm based on lines of maximum coupling within the flow is motivated by the success of implicit tridiagonal line solvers for both finite volume and DG discretizations [24, 26, 18, 19]. The reordering algorithm simply involves creating lines of maximum coupling in the flow as by Fidkowski et. al. [19]. The elements are then reordered in the order that the elements are traversed along each line. We note that this does not produce a unique reordering, since each line may be traversed in either the forward or backward directions. The lines themselves may also be reordered. While a systematic approach may be developed in order to choose an optimal permutation for the lines, the natural ordering produced by the line creation algorithm is used for the test cases presented. For these test cases, reordering the lines according to the standard reordering techniques (Reverse Cuthill-Mckee, Nested-Dissection, One-Way Dissection and Quotient Minimum Degree) or reversing the direction of the lines from the natural ordering did not significantly impact the convergence rate.

4.2 Numerical Results

In order to investigate the effectiveness of a reordering based upon lines, numerical results are presented for two representative test cases: an inviscid transonic flow and a subsonic viscous flow. The convergence plots are presented in terms of the number of linear iterations since the computational cost of performing the ILU(0) factorization or a single linear iteration is independent of the matrix reordering when using the traditional dual matrix storage format. The implications of matrix reordering for the in-place matrix storage format are discussed in Section 4.3.

The first test case is an Euler solution of the transonic flow over the NACA 0012 airfoil at a freestream Mach number of $M = 0.75$ and angle of attack of $\alpha = 2.0°$. The flow is solved using a $p = 4$ discretization on an unstructured mesh with 7344 elements. Figure 4-1 shows the convergence plot of the non-linear residual starting from a converged $p = 3$ solution. The fastest convergence is achieved using the reordering based on lines, which requires only 946 linear iterations for a 10 order drop in residual. One-Way Dissection and Reverse Cuthill-Mckee algorithms also perform well requiring only 1418 and 1611 iterations to converge respectively. On the other hand, Quotient Minimum Degree and Nested Dissection reorderings result in convergence rates which are worse than the "natural" ordering of the elements.

The second test case is a Navier-Stokes solution of the subsonic flow over the NACA0012 airfoil at zero angle of attack with a freestream Mack number of $M = 0.5$ and a Reynolds number of $Re = 1000$. A $p = 4$ solution is obtained on a computational mesh with 2432 elements, where the solution procedure is restarted from a converged $p = 3$ solution. Figure 4-2 presents the convergence plot of the non-linear residual versus linear iterations. The reordering based upon lines is superior to all other reorderings; requiring only 341 iterations to converge. The second best method for this test case is the natural ordering of elements which requires 1350 iterations. The natural reordering performs well for this test case since a structured mesh is used (though the structure is not taken advantage of in the solution procedure), and hence the natural ordering of the elements involves some inherent structure. Of

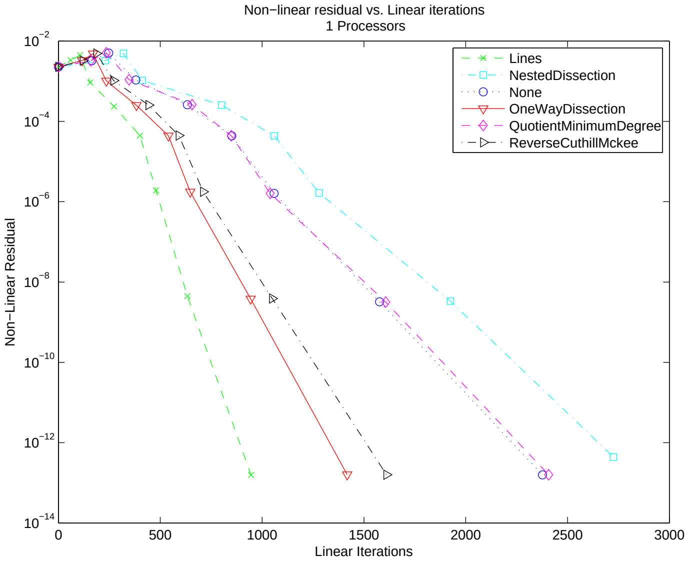


Figure 4-1: Non-linear residual vs linear iterations using the Block-ILU(0) preconditioner with different reordering techniques for a transonic Euler solution of the flow about the NACA0012 airfoil

the other reordering algorithms, Reverse Cuthill-Mckee performs best, requiring 1675 iterations, followed by One-Way Dissection, Quotient Minimum Degree and finally Nested Dissection.

Clearly, reordering the elements according to the lines of maximum coupling results in superior convergence for both inviscid and viscous test cases. The advantages of the line reordering algorithm is especially obvious in the viscous case where reordering according to lines results in a convergence rate nearly 5 times faster than the standard matrix reordering algorithms available in the PETSc package. Due to the clear success of the line reordering algorithm for these two sample problems, the line reordering method is used for the remainder of the work presented here.

4.3 In-place ILU Storage Format

Though the operation count for performing a single linear iteration is independent of the ordering of the elements when using the traditional dual matrix storage format, this is not be the case when using the in-place matrix storage format. The total

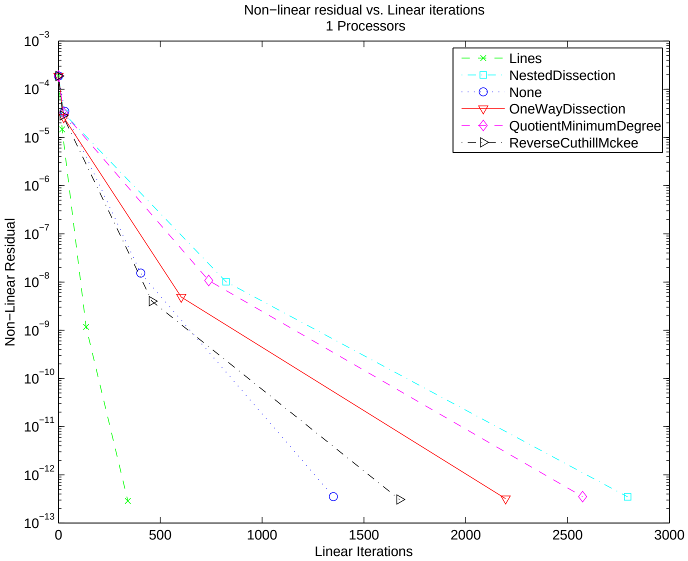


Figure 4-2: Non-linear residual vs linear iterations using the Block-ILU(0) preconditioner with different reordering techniques for a Navier-Stokes solution of the flow about the NACA0012 airfoil

fill which is ignored in the Block-ILU(0) factorization, and hence the size of N, is dependent upon the ordering of the elements, and hence the computational cost of the application of N depends upon the ordering used. As discussed in Section 3.4 the operation count for the application of N is dependent upon the number of higher numbered faces of each element. Using the ordering of the elements based upon lines effectively reduces the number of "free" faces for all but the first element in each line since at least one of the faces corresponds to a lower numbered neighbour. A revised estimate for the operation count for the application of N using the in-place storage format may then be obtained by replacing n_f by $n_f - 1$ in the initial estimate given in Table 3.3. Namely, the revised estimate for the operation count per element is given by: $\frac{2}{3}(n_f + 3)(n_f - 2)n_b^2$.

Table 4.1 shows this revised estimate of the operation count for the application of N normalized by the operation count for the application of A using the traditional dual matrix storage format, for both 2D and 3D problems. Table 4.1 also shows the sample timing results from several sample 2D and 3D problems. For each grid, timing results are presented for $p = 1$ as well as the largest value of p for which the

Dim	Type	# Elements	p	Estimate
2D	**Estimate**			**0.50**
	Structured	2432	1	0.78
	Unstructured	7344	1	0.84
	Cut Cell	1250	1	0.69
	Structured	2432	4	0.51
	Unstructured	7344	4	0.52
	Cut Cell	1250	4	0.46
3D	**Estimate**			**0.93**
	Structured	1920	1	0.86
	Unstructured	45417	1	1.02
	Cut Cell	2883	1	0.98
	Structured	1920	3	0.77
	Cut Cell	2883	3	0.85

Table 4.1: Revised Timing Estimate for Application of N for in-place Block-ILU(0)

Jacobian matrix could fit into memory on a single machine. For the $p = 1$ cases the actual timing results are worse than the revised estimate. However, for large p the actual timing results closely match the revised estimate in 2D, and are bounded by the revised estimate in 3D. The poorer performance for the $p = 1$ cases may be attributed to the effects of lower order terms in n_b, which become significant since the block size for the $p = 1$ solution is relatively small.

A linear iteration may be performed faster using the in-place storage format if the application of N, using the in-place storage format, is faster than the application of A using the traditional dual storage. For large p, the in-place storage format for the Block-ILU(0) solver allows for a linear iteration to be performed at a computational cost which is less than that using the traditional dual matrix storage format. Even in the case of small p, where lower order terms in n_b become significant, the in-place storage format allows for the performance of a linear iteration at a computational cost no greater than the traditional dual matrix storage format.

Chapter 5

Linear Multigrid

Multigrid algorithms are used to accelerate the solution of systems of equations arising from the discretization of a PDE-based problem by applying corrections based on a coarser discretization with fewer degrees of freedom. The coarse discretization may involve a computational mesh with fewer elements (h-multigrid) or a lower order solution space (p-multigrid). The DG discretization naturally lends itself to a p-multigrid formulation as a coarser solution space may be easily created by using a lower order polynomial interpolation within each element. Multigrid algorithms may be used to directly solve a non-linear system of equations (non-linear multigrid), or to solve the system of linear equations arising at each step of Newton's method (linear multigrid). This chapter presents a linear p-multigrid algorithm which is used as a preconditioner to GMRES and makes use of the stationary iterative methods presented in Chapter 3 as linear smoothers on each multigrid level.

5.1 Linear Multigrid Algorithm

The basic two-level linear-multigrid algorithm is presented below. While only a two-level system is presented here, in general the multigrid formulation involves multiple solution levels.

- Perform pre-smoothing: $\tilde{\mathbf{x}}_h^k = (1-\omega)\mathbf{x}_h^k + \omega M_h^{-1}(\mathbf{b}_h - N_h \mathbf{x}_h^k)$

- Compute linear residual: $\tilde{\mathbf{r}}_h^{\tilde{k}} = \mathbf{b}_h - A_h \tilde{\mathbf{x}}_h^k$
- Restrict linear residual: $\mathbf{b}_H = I_H^h \tilde{\mathbf{r}}_h^k$, where I_H^h is the restriction operator
- Define coarse level correction: $\mathbf{x}_H^0 = 0$
- Perform coarse level smoothing: $\mathbf{x}_H^{j+1} = (1-\omega)\mathbf{x}_H^j + \omega M_H^{-1}(\mathbf{b}_H - N_H \mathbf{x}_H^j)$
- Prolongate coarse level correction: $\hat{\mathbf{x}}_h^k = \tilde{\mathbf{x}}_h^k + I_h^H \mathbf{x}_H$, where I_h^H is the prolongation operator
- Perform post-smoothing: $\mathbf{x}_h^{k+1} = (1-\omega)\hat{\mathbf{x}}_h^k + \omega M_h^{-1}(\mathbf{b}_h - N_h \hat{\mathbf{x}}_h^k)$

As presented in Section 2.1, the solution space for the DG discretization is given by $\mathcal{V}_h^p$, the space of piecewise polynomials of order p spanned by the basis functions $\mathbf{v}_{h_i}$. The corresponding coarse solution space is given by $\mathcal{V}_h^{p-1}$, the space of piecewise polynomials of order $p-1$ spanned by the basis functions $\mathbf{v}_{H_k}$. Since $\mathcal{V}_h^{p-1} \in \mathcal{V}_h^p$, the coarse level basis functions may be expressed as a linear combination of the fine level basis functions:

$$\mathbf{v}_{H_k} = \sum_i \alpha_{ik} \mathbf{v}_{h_i}. \tag{5.1}$$

The matrix of coefficients α_{ik} form the prolongation operator I_h^H. The coefficients of expansion may also be used to define the restriction operator by considering the restriction of a component of the residual:

$$\mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{H_k}) = \mathcal{R}_h(\mathbf{u}_h, \sum_i \alpha_{ik} \mathbf{v}_{h_i}) = \sum_i \alpha_{ik} \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{h_i}). \tag{5.2}$$

Hence the restriction operator is given by $I_H^h = \left(I_h^H\right)^T$. In our implementation of the linear multigrid algorithm, the coarse grid Jacobian A_H is given by a simple Galerkin projection of the fine grid Jacobian:

$$A_H = I_H^h A_h I_h^H. \tag{5.3}$$

5.2 Coarse Grid Jacobian

As presented in Section 5.1, the coarse grid Jacobians are formed using a simple Galerkin projection of the fine grid Jacobian. The Galerkin projection is used as opposed to the linearization about the restricted solution since evaluating the Galerkin projection is computationally much less expensive, especially in the case of hierarchical basis functions where the Galerkin projection involves a simple extraction of values. In this section we demonstrate that the Galerkin projection is nearly equivalent to the linearization of the lower order discretization of the restricted solution. The components of the fine grid Jacobian, A_h, are given by:

$$A_{h_{ij}} = \frac{\partial \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{h_i})}{\partial \mathbf{U}_{h_j}}. \tag{5.4}$$

The corresponding coarse grid Jacobian is given by $A_H = I_H^h A_h I_h^H$, where:

$$\begin{aligned} A_{H_{kl}} &= \sum_i \sum_j \alpha_{ik}\alpha_{jl} \frac{\partial \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{h_i})}{\partial \mathbf{U}_{h_j}} && (5.5) \\ &= \sum_j \alpha_{jl} \frac{\partial \mathcal{R}_h(\mathbf{u}_h, \sum_i \alpha_{ik}\mathbf{v}_{h_i})}{\partial \mathbf{U}_{h_j}} && (5.6) \\ &= \frac{\partial \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{H_k})}{\partial \mathbf{U}_{H_l}} && (5.7) \end{aligned}$$

If $\mathcal{R}_h$ is independent of the solution space, then $\mathcal{R}_H = \mathcal{R}_h$ and the expression for A_H simplifies to:

$$A_{H_{kl}} = \frac{\partial \mathcal{R}_H(\mathbf{u}_h, \mathbf{v}_{H_k})}{\partial \mathbf{U}_{H_l}}. \tag{5.8}$$

In the case of linear problems

$$\frac{\partial \mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_{h_i})}{\partial \mathbf{u}_{h_j}} = \mathcal{R}_h(\mathbf{v}_{h_j}, \mathbf{v}_{h_i}), \tag{5.9}$$

hence,

$$A_{H_{kl}} = \mathcal{R}_H(\mathbf{v}_{H_l}, \mathbf{v}_{H_k}). \tag{5.10}$$

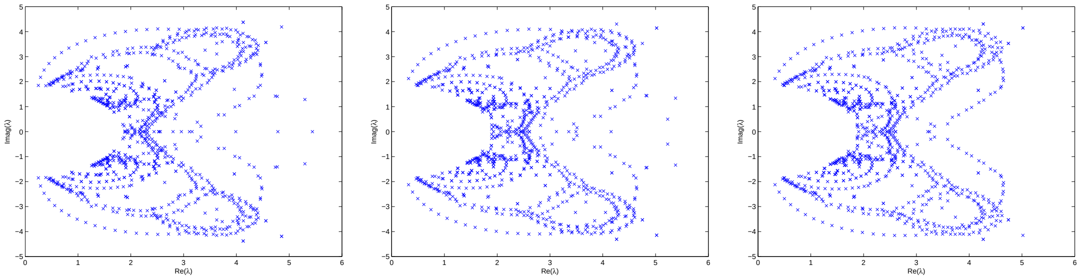
Thus, the coarse level Jacobian is exactly equal to the linearization of the lower order discretization.

Similarly, if $\mathbf{u}_h \in \left(\mathcal{V}_h^{p-1}\right)^{n_s}$, then $\mathbf{u}_h = \mathbf{u}_H$, and

$$A_{H_{kl}} = \frac{\partial \mathcal{R}_H(\mathbf{u}_h, \mathbf{v}_{H_k})}{\partial \mathbf{U}_{H_l}} = \frac{\partial \mathcal{R}_H(\mathbf{U}_H, \mathbf{v}_{H_k})}{\partial \mathbf{U}_{H_l}}, \tag{5.11}$$

so that the coarse level Jacobian based on the Galerkin projection gives the exact linearization of the restricted discretization for a non-linear problem. In general, however, $\mathbf{u}_h \notin \left(\mathcal{V}_h^{p-1}\right)^{n_s}$ and the Galerkin projection of A_h will not result in the linearization $\frac{\partial \mathcal{R}_H(\mathbf{U}_H)}{\partial \mathbf{U}_H}$.

The effectiveness of using the Galerkin projection for the evaluation of the coarse grid Jacobian for the linear multigrid algorithm was verified by performing an eigenvalue analysis of the Jacobian matrix derived from the discretization of a subsonic viscous flow over the NACA0012 airfoil. The Jacobian matrix is evaluated for a $p = 3$ solution of a $M = 0.5$, $Re = 1000$ flow at zero angle of attack using a computational grid with 2432 elements. Figures 5-1(a) and 5-1(b) show the 500 largest magnitude eigenvalues for the $p = 3$ Jacobian and corresponding $p = 2$ Galerkin projection respectively. It can be clearly seen from Figures 5-1(a) and 5-1(b) that the largest eigenvalues of the Galerkin projection closely match those of the $p = 3$ discretization. Figure 5-1(c) shows the eigenvalues of the linearization of a $p = 2$ discretization obtained by restricting the $p = 3$ solution. As predicted by the analysis presented above, the eigenvalues of the Galerkin projection closely match those of the linearization of the restricted flow solution.



(a) $p = 3$ Discretization

(b) $p = 2$ Galerkin Projection

(c) $p = 2$ Discretization

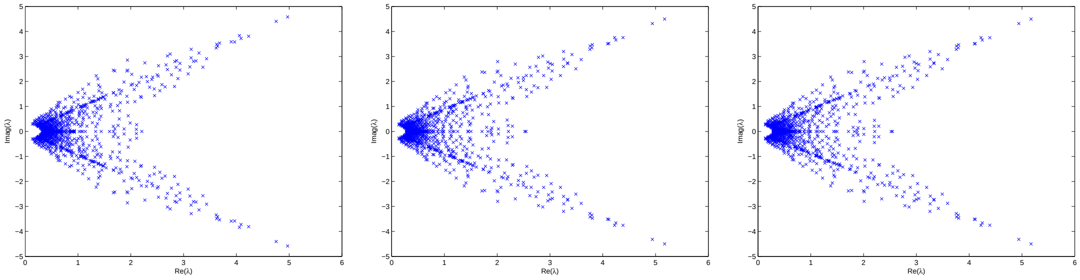
Figure 5-1: Eigenvalue analysis of NACA0012 subsonic viscous test

5.2.1 Stabilization Terms

The previous section demonstrated that the coarse grid Jacobian obtained using a Galerkin projection of the fine grid Jacobian is nearly equivalent to the linearization of the restricted system. These results from the previous section required that the operator $\mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_h)$ be independent of the triangulation $\mathcal{T}_h$ or the solution space $\mathcal{V}_h^p$. Where $\mathcal{R}_h(\mathbf{u}_h, \mathbf{v}_h)$ depends upon the triangulation $\mathcal{T}_h$ or the solution space $\mathcal{V}_h^p$, the restricted system may not be consistent with a lower order the discretization of the governing equations. Okusanya [32, 41], showed that h-dependent stabilization terms, necessary for the streamline upwind/Petrov Galerkin (SUPG) discretization of the Navier-Stokes equations, were improperly scaled when obtaining a coarse grid Jacobian using a Galerkin projection for an h-multigrid scheme. Okusanya also showed that the improper scaling of the stabilization terms can result in the coarse grid system which is inconsistent with the governing PDE resulting in poor solver performance.

Unlike the SUPG discretization, the DG discretization does not require the use of stabilization terms. However, in order to accurately resolve discontinuities in the flow, a discretization-dependent, artificial dissipation is required. In this work, we use the shock capturing scheme presented by Persson and Peraire [35]. Persson and Peraire[35] proposed an artificial dissipation for higher order polynomial approximations that scales as h/p, where h is the mesh spacing and p is the order of the polynomial interpolant within each element. Thus, by increasing p, a shock can be captured within a single element. In the context of p-multigrid, the mesh spacing h is fixed for all multigrid levels, hence the artificial dissipation terms should scale as $1/p$. In order to examine the behaviour of the Galerkin projection linear multigrid algorithm for flows with discontinuities, an eigenvalue analysis is performed on the Jacobian matrices arising from two representative test cases: a transonic flow involving a weak normal shock and a hypersonic flow involving a strong oblique shock.

The first test case is a transonic, $M = 0.75$ flow over the NACA0012 airfoil at an angle of attack of $\alpha = 2°$ discretized on a compuational mesh with 165 elements. Figure 5-2(a) shows the plot of the 500 largest magnitude eigenvalues of the Jacobian matrix corresponding to a $p = 3$ discretization while Figures 5-2(b) and 5-2(c) show the $p = 2$ Jacobians based on the Galerkin projection and $p = 2$ discretization respectively. Since this test case involves only a relatively weak shock, dominant modes of the Jacobian matrix are not significantly affected by the presence of artificial dissipation terms. As a result, the eigenvalue spectra of the Jacobians based on the $p = 2$ Galerkin projection and the $p = 2$ discretization closely match those of the $p = 3$ discretization.



(a) $p = 3$ Discretization

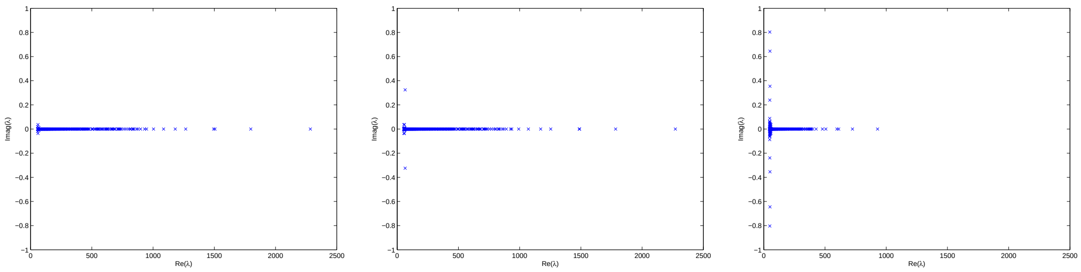
(b) $p = 2$ Galerkin Projection

(c) $p = 2$ Discretization

Figure 5-2: Eigenvalue analysis of the transonic, $M = 0.75$, NACA0012 test case

The second test case is a hypersonic, $M = 11$, flow in a shock ramp involving a strong oblique shock. The eigenvalue spectra of the Jacobian matrix for a $p = 3$ discretization, and the corresponding Jacobians based on the $p = 2$ Galerkin projection and $p = 2$ discretizations, are presented in Figure 5-3. The eigenvalues of the $p = 3$

discretization and the Galerkin projection closely match for the largest magnitude eigenvalues which extend out along the real axis due to the presence of the strong shock. On the otherhand the eigenvalues of the Galerkin projection do not match those of the Jacobian for the $p = 2$ discretization. The presence of larger artificial dissipation terms reduces the spectral radius of the Jacobian for the $p = 2$ discretization, while the corresponding spectral radius of the Galerkin projection matches the spectral radius of the $p = 3$ discretization. The Jacobian based on the Galerkin projection is significantly different from the coarse order discretization, that the coarse level system gives poor solution updates. As a result of these poor solution updates the linear multigrid method is unstable for this problem and cannot be used as an effective preconditioner.



(a) $p = 3$ Discretization

(b) $p = 2$ Galerkin Projection

(c) $p = 2$ Discretization

Figure 5-3: Eigenvalue analysis of the hypersonic, $M = 11$, shock ramp test case

5.3 Memory Considerations

In many aerodynamic applications the size of a problem which may be solved on a given machine is limited by the memory available. Therefore, memory considerations are important in the development of an efficient preconditioner. The memory usage for any Newton-Krylov based solver is dominated by the storage of the Jacobian matrix and Krylov vectors [40]. For a linear multigrid preconditioner significant additional memory is required for the storage of the lower order Jacobians on each multigrid level. Table 5.1 shows the additional memory required for all lower order Jacobians in terms of the fine grid Jacobian for $p = 1 \rightarrow 5$.

Solution Order	2D	3D
$p = 1$	11.1%	6.25%
$p = 2$	27.7%	17.0%
$p = 3$	46.0%	29.3%
$p = 4$	64.9%	42.2%
$p = 5$	84.1%	55.5%

Table 5.1: Additional memory usage for lower order Jacobians for linear multigrid

Several authors [27, 17] have argued that a linear multigrid preconditioner may be unfeasible for large problems due to the additional memory cost of storing these lower order Jacobians. Alternatively, others have advocated for skipping multigrid levels to reduce memory usage. For example, Persson and Peraire [34] employed a multi-level scheme where only $p = 0$ and $p = 1$ corrections were applied. Though the linear multigrid method may require significant additional memory for the storage of the lower order Jacobians, faster convergence of the GMRES method is expected and hence fewer Krylov vectors may be required in order to obtain a converged solution. Hence, in order to provide a memory equivalent comparison between a single- and multi-level preconditioner, the total memory usage for the Jacobians and Krylov vectors must be considered. In the context of a restarted GMRES algorithm this is equivalent to increasing the GMRES restart value for the single level preconditioner so that the total memory used by the single and multi-level preconditioners is the same. Table 5.2 gives the additional memory for the storage of all lower order Jacobians for the linear multigrid solver in terms of the number of solution vectors on the fine grid. Table 5.2 may also be viewed as the additional number of GMRES vectors allocated for the single-level preconditioner to provide a memory equivalent comparison with the multigrid preconditioner.

Solution Order	2D	3D
$p = 1$	5	6
$p = 2$	27	43
$p = 3$	74	146
$p = 4$	156	369
$p = 5$	283	778

Table 5.2: Additional memory usage for lower order Jacobians for linear multigrid solver in terms of solution vectors

5.4 Numerical Results

The performance of the three preconditioners presented in Chapter 3, as well as the linear multigrid preconditioner presented in this chapter are evaluated using two representative test cases: an inviscid transonic flow and a viscous subsonic flow.

5.4.1 Inviscid Transonic flow over NACA0012, $M = 0.75$, $\alpha = 2°$

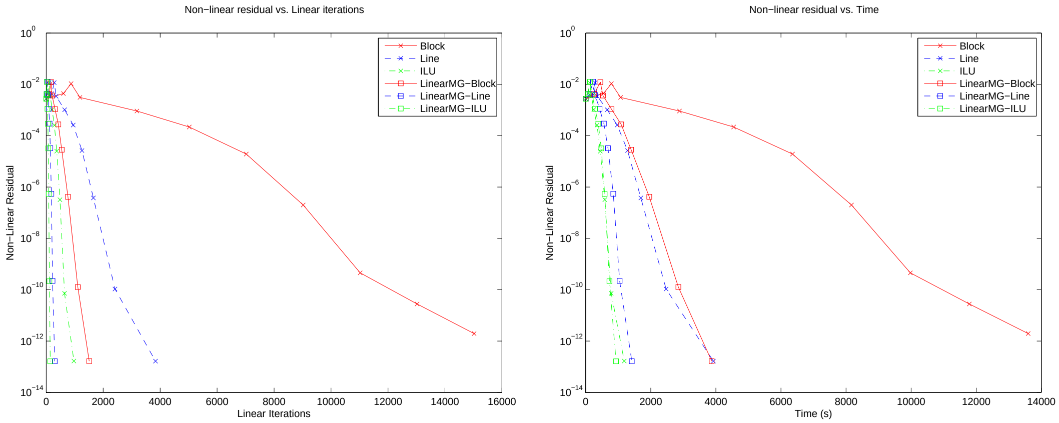
The first test case is an Euler solution of the transonic flow over the NACA0012 airfoil at an angle of attack of $\alpha = 2°$ with a free-stream Mach number of $M = 0.75$. This flow is solved using a $p = 4$ discretization on a computational mesh with 7344 elements. A GMRES restart value of 40 is used for the linear multigrid preconditioner while a memory equivalent GMRES restart value of 200 is used for the single-level preconditioners. The number of linear iterations taken in each Newton step is determined by the tolerance criterion specified in Equation (2.16) up to a maximum of 10 GMRES outer iterations. Table 5.3 shows the convergence results for the different preconditioners in terms on the number of non-linear Newton iterations, linear iterations, GMRES outer iterations and CPU time. The convergence history of the non-linear residual versus linear iterations and CPU time are given in Figures 5-4(a) and 5-4(b) respectively. The residual tolerance criterion developed in Section 2.3 ensures sufficient convergence of the linear system in each Newton step so that quadratic convergence of the non-linear residual is observed for all preconditioners except Block-Jacobi. Additionally, the residual tolerance criterion developed in Section 2.3 ensures that the convergence history of the non-linear residual in terms of non-linear iterations is the same for these preconditioners. The difference in behaviour for the Block-Jacobi preconditioner is due to stalling of the restarted GMRES algorithm, which prevents a sufficient convergence of the linear system to obtain quadratic convergence.

Preconditioner	Newton Iter	Linear Iter	GMRES Outer	Time(s)
Block-Jacobi	10	15024	78	13596
Line-Jacobi	9	3836	23	3925
Block-ILU	9	971	10	1184
LinearMG w/ Block-Jacobi	9	1511	40	3873
LinearMG w/ Line-Jacobi	9	301	11	1417
LinearMG w/ Block-ILU	9	142	9	934

Table 5.3: Convergence results of the inviscid transonic NACA0012 test case

Using the single-level Block-ILU preconditioner significantly reduces the number of linear iterations required to converge compared to the single-level Line-Jacobi and Block-Jacobi preconditioners. This improved convergence using the Block-ILU preconditioner ensures that the GMRES restart value is reached only once. On the other hand, the GMRES restart value is reached in each Newton iteration for the Block-Jacobi preconditioner and all but the first three Newton iteration for the Line-Jacobi preconditioner. The repeated restarting of the GMRES algorithm degrades the convergence rate and leads to the stalling of the GMRES algorithm using the Block-Jacobi preconditioner. While both the preprocessing and the iterative stages of the Block-ILU preconditioner are more expensive than the corresponding stages of the Line-Jacobi or Block-Jacobi preconditioners, the significant reduction in the number of linear iterations ensures that the Block-ILU preconditioner achieves fastest convergence in terms of CPU time.

The linear multigrid preconditioners with Block-Jacobi, Line-Jacobi and Block-ILU smoothing significantly reduce the number of linear iterations required to achieve convergence compared to the corresponding single-level preconditioners. The improved convergence rate in term of the number of linear iterations ensure that the GMRES restart value is not reached as often for the multi-level preconditioners despite the memory equivalent GMRES restart value being 5 times smaller than the



(a) Non-linear residual vs linear iterations

(b) Non-linear residual vs time

Figure 5-4: Convergence plots of the inviscid transonic NACA0012 test case

single-level preconditioners. This ensures that GMRES stall is not seen with the linear multigrid preconditioner using Block-Jacobi smoothing. Additionally, the GMRES restart value is reached only twice for the linear multigrid preconditioner with Line-Jacobi smoothing.

Though the linear multigrid preconditioner significantly reduces the number of linear iterations required to converge this problem, the cost of each application of the linear multigrid preconditioner is more expensive than the single level preconditioner. However, fastest convergence in terms of CPU time is achieved using the linear multigrid preconditioner with Block-ILU smoothing which performs about 20% faster than the single level Block-ILU preconditioner.

5.4.2 Viscous Subsonic flow over NACA0012, $M = 0.5$, $\alpha = 0°$, $Re = 1000$

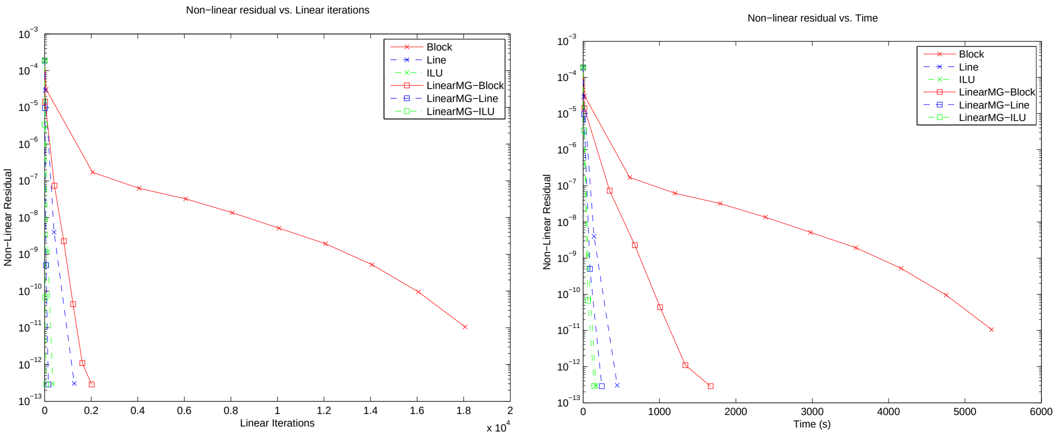
The second test case is a Navier-Stokes solution of a subsonic, $M = 0.5$ flow over the NACA0012 airfoil at zero angle of attack with Reynolds number $Re = 1000$. The flow is discretized using a $p = 4$ solution on a computational grid with 2432 elements. Once again, a GMRES restart value of 40 is used for the linear multigrid preconditioner while a memory equivalent GMRES restart value of 200 is used for the single-level preconditioners. The convergence data for the different preconditioners is summarized in Table 5.4 while the convergence plots are presented in Figure 5-5.

Preconditioner	Newton Iter	Linear Iter	GMRES Outer	Time(s)
Block-Jacobi	10	18060	91	5348
Line-Jacobi	3	1271	8	446
Block-ILU	3	351	4	166
LinearMG w/ Block-Jacobi	6	2020	51	1669
LinearMG w/ Line-Jacobi	3	159	6	244
LinearMG w/ Block-ILU	3	66	3	146

Table 5.4: Convergence results of the viscous subsonic NACA0012 test case

In order to achieve fast convergence for this viscous test case, it is necessary that the preconditioner sufficiently resolves the coupling between element in the boundary layer. Since the Block-Jacobi preconditioner ignores all inter-element coupling, the restarted GMRES algorithm stalls and the linear system is not sufficiently solved such that several additional Newton iterations are required to converge the non-linear residual. On the other hand, the Line-Jacobi and Block-ILU preconditioners which make use of the lines of maximum coupling within the flow are able to sufficiently converge the linear system at each Newton step and quadratic convergence of the non-linear residual is observed.

As with the inviscid test case, the use of the linear multigrid preconditioner significantly reduces the number of linear iterations required to converge the linear system at each Newton step. The GMRES restart value is reached less often in the case of the Linear Multigrid preconditioners despite the GMRES restart value being five times larger for the single-level preconditioners. This ensures that the Linear Multigrid preconditioner with Block-Jacobi smoothing is able to solve the linear system sufficiently to converge the non-linear residual in 6 non-linear iterations as opposed to 10 for the corresponding single-level Block-Jacobi preconditioner. Additionally, only the Linear Multigrid preconditioner with Block-ILU smoothing is able to converge the linear system at each Newton step without restarting GMRES. Once again, the fastest convergence in terms of CPU time is achieved using the Linear Multigrid



(a) Non-linear residual vs linear iterations

(b) Non-linear residual vs time

Figure 5-5: Convergence plots of the viscous subsonic NACA0012 test case

preconditioner with Block-ILU smoothing.

Chapter 6

Parallel Performance

The solutions of complex 3D problems necessitates the use of parallel computing. The development of an efficient solver for DG discretizations must therefore necessarily consider the implications of parallel computing. This chapter discusses the parallel implementation of the DG solver, as well as the parallel performance of the linear multigrid preconditioner presented in the previous chapters.

6.1 Parallel Implementation

Parallel implementation involves partitioning the computational grid across multiple processors, where each processor maintains all elements in a partition. In addition, each processor maintains ghosted data, corresponding to neighbouring elements on other partitions that are required for the local computation of the residual and Jacobian matrix. The ghosted states are updated from the appropriate partition at the beginning of each residual evaluation, where communication is performed using the Message Passing Interface (MPI).

6.2 Parallel Preconditioner Implementation

Except for the Block-Jacobi preconditioners, the preconditioners presented in the previous chapters have some inherent serialism as they require elements to be traversed sequentially. Thus, while the Block-Jacobi preconditioners can be trivially parallelized, the Line-Jacobi and Block-ILU methods are more difficult. In our parallel implementation of the Line-Jacobi and Block-ILU preconditioners, the off-partition coupling between elements is ignored. For the Line-Jacobi method this implies lines are cut at partition boundaries with a potential for performance degradation. Similarly, the Block-ILU performance may be degraded by ignoring fill that would occur between elements on different partitions. To decrease the potential degradation of the preconditioners, we utilize a grid partitioning strategy that maintains the lines within a partition.

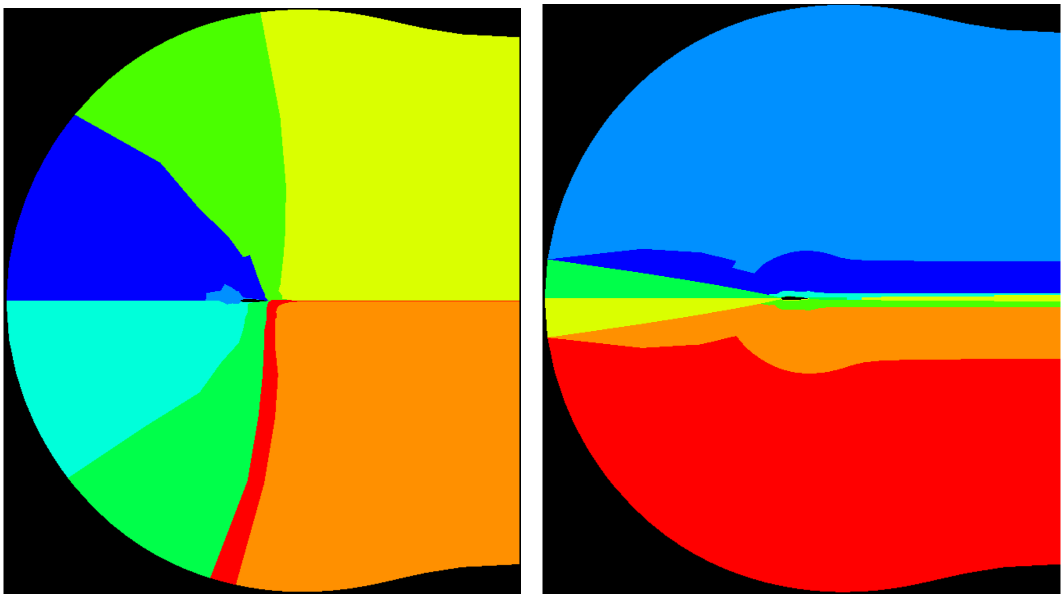
6.3 Grid Repartitioning

The importance of maintaining lines when performing parallel computation of viscous flow using line-implicit solvers has been discussed by several authors [26, 28, 17, 31]. Mavriplis [26] employed different grid partitionings for each level of a directional-implicit agglomeration-multigrid algorithm so as to ensure efficient parallel performace. Mavriplis presented a grid partitioning scheme where all elements in a line were grouped into a single macro element for the partitioning algorithm so as to guarantee no lines were cut. In the context of the Line-Jacobi preconditioner presented in this paper, the lines change every non-linear solution update since the lines are based on the current approximate solution. In order to retain the preconditioner performance seen in the serial case, it would be necessary to repartition the grid throughout the solution procedure to ensure that lines are not cut. While it may be possible to repartition the grid each non-linear iteration, this may be unnecessary since the lines may not significantly change from iteration to iteration. Instead, the grid is repartitioned after a fixed number of non-linear iterations, typically every 10 to 15 non-linear iterations.

The grid repartitioning scheme involves five steps. First, the coupling between elements is determined in parallel, where the initial partitioning may be based on the unweighted adjacency graph of the elements in the grid or a previous partitioning based on lines. Second, the lines are formed in parallel allowing connections across partition boundaries. Third, a weighted adjacency graph is created by grouping together all elements in a line into macro-elements. The nodes (i.e. lines) in the adjacency graph are weighted by the number of elements in each line, while the edges are weighted by the number of faces between lines. Fourth, ParMetis is used to partition the grid based on a weighted adjacency graph. Finally, the lines are reformed in parallel on the repartitioned grid, where connections are not permitted across partition boundaries.

The parallel line formation algorithm is modified slightly from the serial version presented by in detail Fidkowski [17]. The serial version of the line formation algorithm involves two stages. In the first stage, lines are formed by connecting elements with maximum coupling such that each element may only be connected across the two faces with the largest coupling. In the second stage, adjacent lines are joined if the coupling across the face between two endpoints is maximum over the free faces of both endpoint. In parallel, the line formation algorithm is performed differently before and after the grid is repartitioned. Prior to repartitioning, the first stage of the line formation algorithm is identical to the first stage of the serial implementation, however, second stage connections are not permitted across partition boundaries. After repartitioning, neither first nor second stage connections are permitted across partition boundaries. However, additional second stage connections are permitted between local line endpoints which may have larger coupling to faces across partition boundaries. This allows for longer lines to be formed where second stage connections across partition boundaries are ignored. Grid repartitioning ensures that the lines formed in parallel are identical to those formed in serial up to first stage connections, while second stage connections may be cut across partition boundaries.

Figure 6-1 shows the sample partitioning of the computational grid used for the viscous NACA0012 test case presented in Section 5.4 for 8 processors. Figure 6-1(a) gives the initial partitioning based on the unweighted adjacency graph of the elements in the grid, while Figure 6-1(b) shows the corresponding repartitioning according to lines. As shown in Figure 6-1(b), grid repartitioning produces partitions which are elongated in the stream-wise direction, as the lines of maximum coupling follow streamlines in the flow.



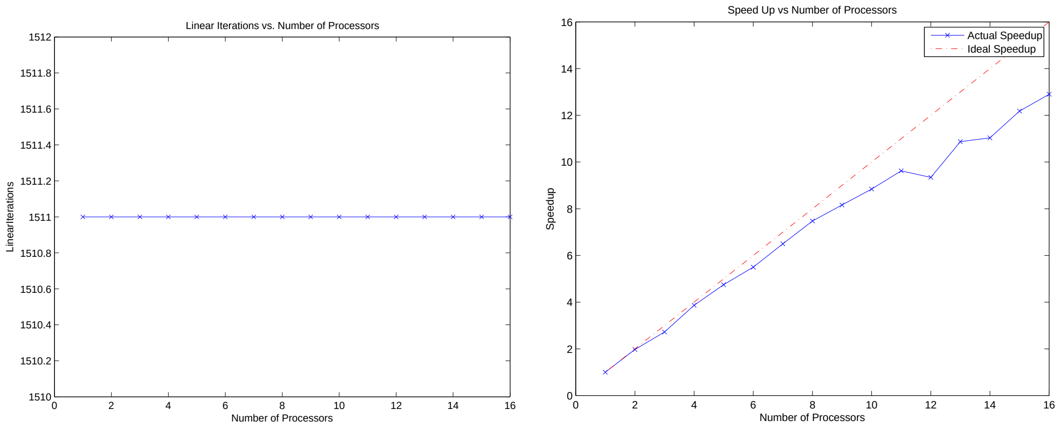
(a) Initial partitioning

(b) Line repartitioning

Figure 6-1: Different grid partitionings for the viscous NACA0012 test case

6.4 Numerical Results

Parallel performance results are presented for the two sample test cases discussed in Section 5.4, namely an inviscid transonic flow and a viscous subsonic flow over the NACA0012 airfoil. The performance results presented in this section give the total wall clock time for solving each problem on a given number of processors so as to ensure that all parts of the flow solution procedure are considered, including the pre- and post- processing steps for grid partitioning and reassembly. The parallel efficiency of an algorithm is affected by communication time, caching effects, and time spent on repeated computations (such as the duplicated evaluation of the fluxes on partition boundaries). In order to isolate these effects the sample timing and parallel speed-up results for the linear multigrid preconditioner with Block-Jacobi smoothing for the inviscid transonic flow test case are plotted in Figure 6-2. Figure 6-2(a) plots the number of linear iterations required to converge for 1 to 16 processors. As discussed previously, the parallel Block-Jacobi preconditioner is the same as in the serial case hence the number of linear iterations required to converge is the same for all number of processors. As shown in Figure 6-2(b) good parallel speed-up is observed for 1-16 processors.



(a) Linear iterations vs. # processors

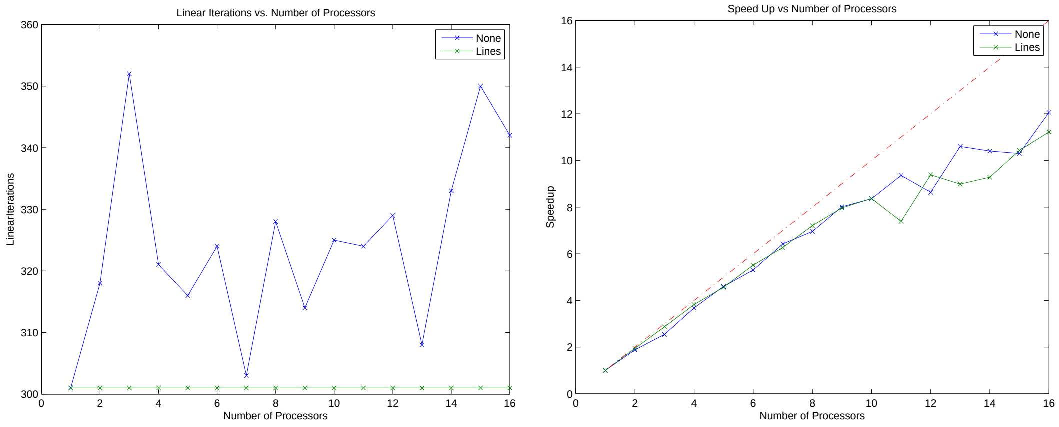
(b) Parallel speed-up vs. # processors

Figure 6-2: Parallel performance of the linear multigrid preconditioner with Block-Jacobi smoothing for the inviscid transonic NACA0012 test case

6.4.1 Linear multigrid preconditioner with Line-Jacobi smoothing

Figure 6-3(b) shows the parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing for the inviscid transonic flow test case. Figure 6-3(a) plots the number of linear iterations required to converge versus the number of processors. Using no grid repartitioning results in degraded preconditioner performance as lines are cut along partition boundaries. On the other hand, for this test case repartitioning according to lines ensures that the same number of linear iterations are used for all processes. Figure 6-3(b) shows the corresponding parallel speed-up.

Since cutting lines does not significantly increase the number of linear iterations required to converge the solution, good parallel speed up is observed for both grid partitionings.



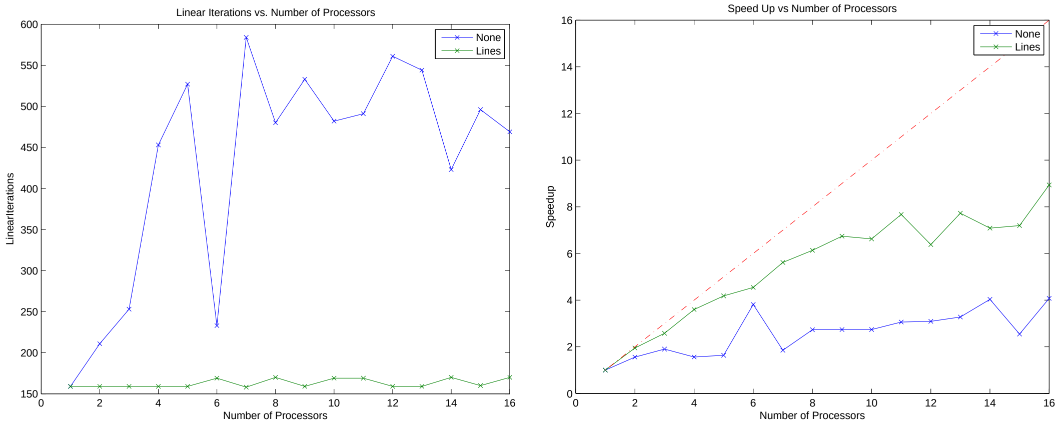
(a) Linear iterations vs. # processors

(b) Parallel speed-up vs. # processors

Figure 6-3: Parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing inviscid transonic NACA0012 test case

Figure 6-4 shows the parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing for the viscous subsonic flow test case. The cutting of lines significantly reduces the performance of the preconditioner, hence significantly more linear iterations are required to reach convergence if no grid repartitioning is employed. Repartitioning according to lines results in superior preconditioning performance, where the number of linear iterations is nearly constant over all processors, with small variations due to the cutting of lines in the second stage of the line formation algorithm.

Figure 6-4(b) shows the corresponding parallel speed-up for both partitioning methods. As expected, significantly better parallel speed-up is observed when a good repartitioning of the grid is employed to ensure that no lines are cut. Unfortunately the parallel speed-up using the repartitioning according to lines is still far from ideal. Since the grid for this test case involves only 2342 elements, there are relatively few lines on each partition, leading to partitions which are long and thin with large numbers of ghosted elements. For example the 16 processor partitions had an average of only 4 lines and 152 elements per partition, while each partition had an average of 32 ghosted elements. On the other hand the base partition required only 14 ghosted elements per partition. In practice, the solution of large aerodynamic problems will involve partitions with thousands of elements and the ratio of ghosted elements to local elements will remain relatively small, resulting in good parallel performance.



(a) Linear iterations vs. # processors

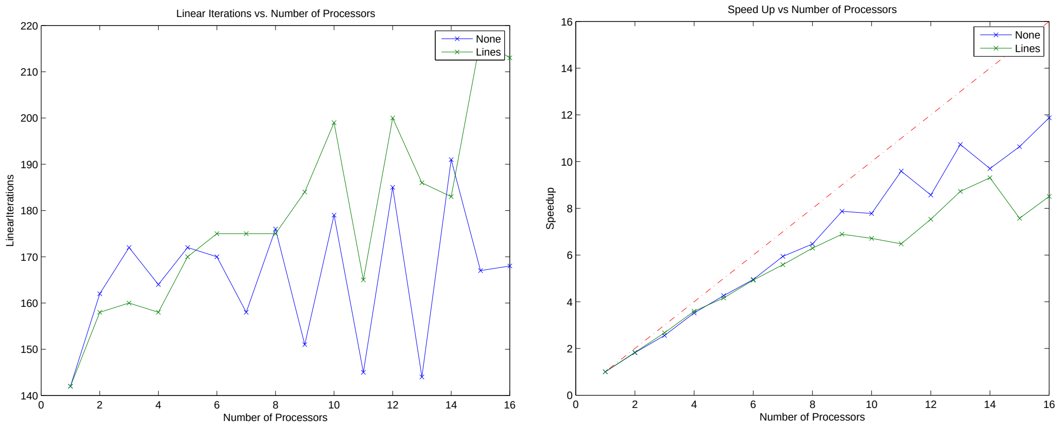
(b) Parallel speed-up vs. # processors

Figure 6-4: Parallel performance of the linear multigrid preconditioner with Line-Jacobi smoothing viscous subsonic NACA0012 case

6.4.2 Linear multigrid preconditioner with Block-ILU smoothing

The parallel performance of the transonic inviscid NACA0012 test case using the linear multigrid preconditioner with Block-ILU smoothing is presented in Figure 6-5. Partitioning the domain degrades the performance of the Block-ILU preconditioner since the coupling between elements across partition boundaries is ignored. The degraded preconditioner performance increases the number of linear iterations required to converge as shown in Figure 6-5(a). For this particular test case, repartitioning according to lines results in more linear iterations, which appears to contradict the result for the linear multigrid preconditioner with Line-Jacobi smoothing. While repartitioning according to lines ensures that the coupling along lines is captured within a partition, more off-partition coupling is ignored due to the larger number of ghosted elements associated with line repartitioning.

Figure 6-5(b) shows corresponding parallel speed-up. Since the number of linear iteration required for convergence does not significantly increase with the number of processors, relatively good parallel performance is observed.

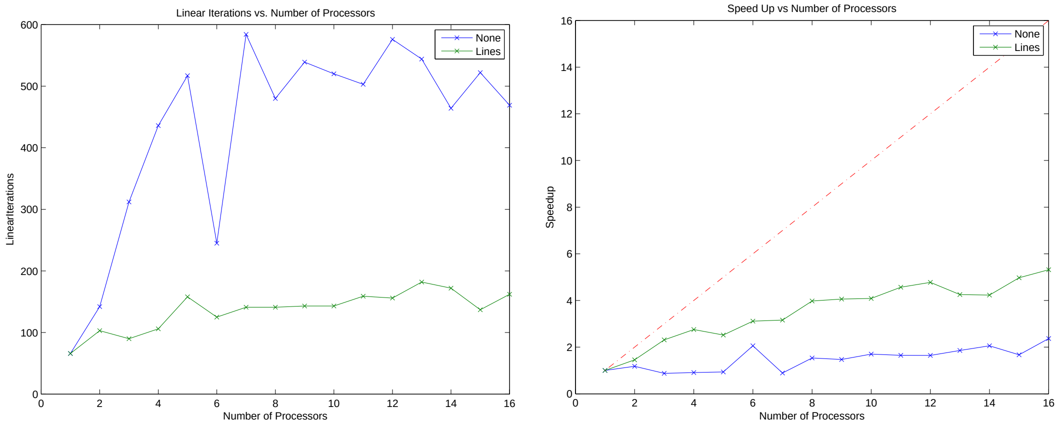


(a) Linear iterations vs. # processors

(b) Parallel speed-up vs. # processors

Figure 6-5: Parallel performance of the linear multigrid preconditioner with Block-ILU smoothing NACA0012 Navier-Stokes test case

Figure 6-6 shows the parallel performance of the linear multigrid preconditioner with Block-ILU smoothing for the subsonic viscous NACA0012 airfoil test case. The lines of maximum coupling are very important for accurately resolving this test case. As a result, cutting the lines across partition boundaries significantly reduces the preconditioner performance. As shown in Figure 6-6(a) the number of linear iterations required to reach converges increases with the number of processors irregardless of the repartitioning used. However, repartitioning according to lines can significantly reduce the number of linear iterations required. Unfortunately the performance of the Block-ILU preconditioner degrades even if no lines are cut, since the coupling between elements across partitions is ignored. The corresponding parallel speed-up is degraded by the increase in the number of linear iterations, as well as the additional repeated calculations for ghosted elements. The corresponding parallel speed-up for both grid partitioning schemes are presented in Figure 6-6(b).

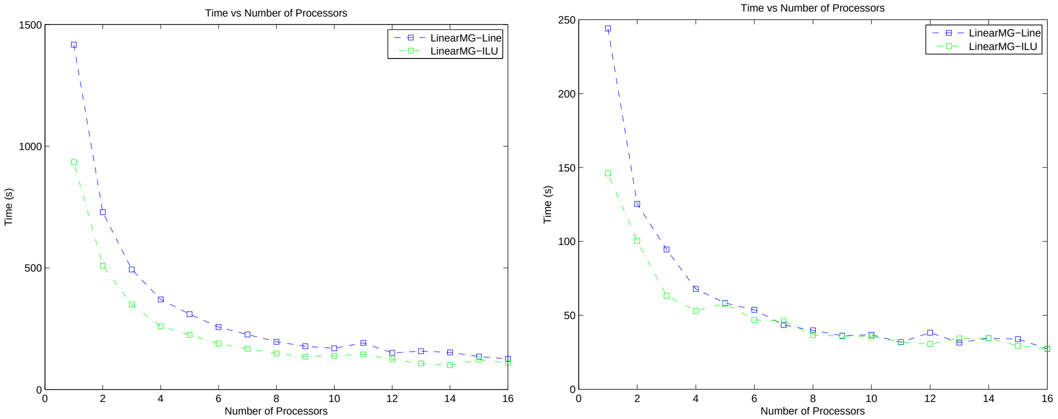


(a) Linear iterations vs. # processors

(b) Parallel speed-up vs. # processors

Figure 6-6: Parallel performance of the linear multigrid preconditioner with Block-ILU smoothing viscous subsonic NACA0012 case

Since the performance of the linear multigrid preconditioner with Block-ILU smoothing is degraded even if lines are not cut across partition boundaries, the parallel speed-up is poor compared to the linear multigrid preconditioner with Line-Jacobi smoothing. Hence, as the number of processors increases the performance of the linear multigrid preconditioner with Line-Jacobi smoothing approaches that with Block-ILU smoothing. Figure 6-7 shows the plot of CPU time versus the number of processors using the linear multigrid preconditioners for the two samples test cases presented. For both test cases, repartitioning based on lines is used to improve parallel performance. Despite poorer parallel speed-up, Block-ILU smoothing outperforms Line-Jacobi smoothing for 1-16 processors for the inviscid transonic test case. On the other hand, for the viscous subsonic test case, while Block-ILU smoothing outperforms Line-Jacobi for 1-4 processors, both Line-Jacobi and Block-ILU smoothing are comparable for larger number of processor.



(a) NACA0012 inviscid transonic case (b) NACA0012 subsonic viscous case

Figure 6-7: Parallel timing results

Chapter 7

Conclusions

An efficient, parallel, solution algorithm has been presented for the Discontinuous Galerkin discretization of the compressible Navier-Stokes equations on unstructured grids. The algorithm is based on a Newton-Krylov approach with a linear p-multigrid preconditioner using a Block-ILU(0) smoother.

An in-place Block-ILU(0) factorization algorithm has been developed, which has been shown to reduce both the memory and computational cost over the traditional dual matrix storage format. A reordering technique for the Block-ILU(0) factorization, based upon lines of maximum coupling in the flow, has also been developed. The results presented show that this reordering technique significantly reduces the number of linear iterations required to converge compared to standard reordering techniques, especially for viscous test cases.

A linear p-multigrid preconditioner has been developed by using a Galerkin projection to obtain coarse level Jacobians. The Galerkin projection is shown to produce nearly the exact linearization of a lower order discretization except in the case of flow solutions with strong shocks, where a multigrid algorithm is not appropriate. The linear multigrid preconditioner is shown to significantly reduce the number of linear iterations required to obtain a converged solution compared to a single-level preconditioner. The linear multigrid preconditioner also results in faster convergence in terms of CPU time for the representative test cases presented. A parallel implementation of the linear multigrid preconditioner has also been presented, which uses a grid repartitioning algorithm to ensure lines of maximum coupling within the flow are not cut across partition boundaries.

Bibliography

[1] W. Anderson, R. Rausch, and D. Bonhaus. Implicit multigrid algorithms for incompressible turbulent flows on unstructured grids. Number 95-1740-CP. In Proceedings of the 12th AIAA CFD Conference, San Diego CA, 1995.

[2] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. Petsc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[3] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2007. http://www.mcs.anl.gov/petsc.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[5] Timothy Barth. Numerical methods for conservation laws on structured and unstructured meshes. VKI March 2003 Lecture Series, 2003.

[6] F. Bassi and S. Rebay. High-order accurate discontinuous finite element solution of the 2d Euler equations. *Journal of Computational Physics*, 138(2):251–285, 1997.

[7] F. Bassi and S. Rebay. A high-order discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 131:267–279, 1997.

[8] F. Bassi and S. Rebay. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations. In Karniadakis Cockburn and Shu, editors, *Discontinuous Galerkin Methods: Theory, Computation and Applications*, pages 197–208. Springer, Berlin, 2000.

[9] F. Bassi and S. Rebay. Numerical evaluation of two discontinuous Galerkin methods for the compressible Navier-Stokes equations. *International Journal for Numererical Methods in Fluids*, 40:197–207, 2002.

[10] Michele Benzi, Daniel B. Szyld, and Arno van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing*, 20(5):1652–1670, 1999.

[11] Max Blanco and David W. Zingg. A fast solver for the Euler equations on unstructured grids using a Newton-GMRES method. AIAA Paper 1997-0331, January 1997.

[12] X.-C. Cai, W. D. Gropp, D. E. Keyes, and M. D. Tidriri. Newton-Krylov-Schwarz methods in CFD. pages 17–30. In Proceedings of the International Workshop on Numerical Methods for the Navier-Stokes Equations, 1995.

[13] B. Cockburn, G. Karniadakis, and C. Shu. The development of discontinuous Galerkin methods. In *Lecture Notes in Computational Science and Engineering*, volume 11. Springer, 2000.

[14] Bernardo Cockburn and Chi-Wang Shu. Runge-kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, pages 173–261, 2001.

[15] Dimitri J. Mavriplis Cristian R. Nastase. High-order discontinuous Galerkin methods using a spectral multigrid approach. AIAA Paper 2005-1268, January 2005.

[16] V. Dolejší and M. Feistauer. A semi-implicit discontinuous Galerkin finite element method for the numerical solution of inviscid compressible flow. *Journal of Computational Physics*, 198(1):727–746, 2004.

[17] Krzysztof J. Fidkowski. A high-order discontinuous Galerkin multigrid solver for aerodynamic applications. Masters thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2004.

[18] Krzysztof J. Fidkowski and David L. Darmofal. Development of a higher-order solver for aerodynamic applications. AIAA Paper 2004-0436, January 2004.

[19] Krzysztof J. Fidkowski, Todd A. Oliver, James Lu, and David L. Darmofal. p-multigrid solution of high-order discontiunous Galerkin discretizations of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 207(1):92–113, 2005.

[20] Koen Hillewaert, Nicolas Chevaugeon, Philippe Geuzaine, and Jean-Francois Remacle. Hierarchic multigrid iteration strategy for the discontinuous Galerkin solution of the steady Euler equations. *International Journal for Numerical Methods in Fluids*, 51(9):1157–1176, 2005.

[21] C. T. Kelley and David E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal of Numerical Analysis*, 35(2):508–523, 1998.

[22] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(1):357–397, 2004.

[23] Hong Luo, Joseph D. Baum, and Rainald Löhner. A *p*-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids. *Journal of Computational Physics*, 211(1):767–783, 2006.

[24] D. J. Mavriplis. Multigrid strategies for viscous flow solvers on anisotropic unstructured meshes. *Journal of Computational Physics*, 145:141–165, 1998.

[25] D. J. Mavriplis. On convergence acceleration techniques for unstructured meshes. AIAA Paper 1998-2966, 1998.

[26] Dimitri J. Mavriplis. Large-scale parallel viscous flow computations using an unstructured multigrid algorithm. ICASE XSReport TR-99-44, 1999.

[27] Dimitri J. Mavriplis. An assessment of linear versus nolinear multigrid methods for unstructured mesh solvers. *Journal of Computational Physics*, 175(1):302–325, 2002.

[28] Dimitri J. Mavriplis and S. Pirzadeh. Large-scale parallel unstructured mesh computations for 3d high-lift analysis. Technical report, 1999.

[29] Cristian R. Nastase and Dimitri J. Mavriplis. High-order discontinuous Galerkin methods using an *hp*-multigrid approach. *Journal of Computational Physics*, 213(1):330–357, 2006.

[30] Amir Nejat and Carl Ollivier-Gooch. Effect of discretization order on preconditioning and convergence of a higher-order unstructured Newton-Krylov solver for inviscid compressible flows. AIAA Paper 2007-0719, January 2007.

[31] Eric J. Nielsen, James Lu, Michael A. Park, and David L. Darmofal. An implicit, exact dual adjoint solution method for turbulent flows on unstructured grids. Technical report, 2004.

[32] Tolulope O. Okusanya. *Algebraic Multigrid for Stabilized Finite Element Discretizations of the Navier-Stokes Equations.* PhD dissertation, M.I.T., Department of Aeronautics and Astronautics, June 2002.

[33] Todd A. Oliver. Multigrid solution for high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations. Masters thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2004.

[34] Per-Olof Persson and Jaime Peraire. An efficient low memory implicit DG algorithm for time dependent problems. AIAA Paper 2006-0113, January 2006.

[35] Per-Olof Persson and Jaime Peraire. Sub-cell shock capturing for discontinuous Galerkin methods. Aiaa paper, January 2006.

[36] Alberto Pueyo and David W. Zingg. An efficient Newton-GMRES solver for aerodynamic computations. AIAA Paper 1997-1955, 1997.

[37] Patrick Rasetarinera and M. Y. Hussaini. An efficient implicit discontinuous spectral Galerkin method. *Journal of Computational Physics*, 172(1):718–738, 2001.

[38] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981.

[39] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[40] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 1996.

[41] J. Peraire T. Okusanya, D.L. Darmofal. Algebraic multigrid for stabilized finite element discretizations of the Navier-Stokes equations. *Computational Methods in Applied Mechanics and Engineering*, 193(1):3667–3686, 2004.

[42] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.