THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

MASSIVELY PARALLEL ALGORITHMS FOR CFD SIMULATION AND OPTIMIZATION ON HETEROGENEOUS MANY-CORE ARCHITECTURES

By

AUSTEN C. DUFFY

A Dissertation submitted to the
Department of Mathematics
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2011

The members of the committee approve the dissertation of Austen C. Duffy defended on March 15, 2011.

Mark Sussman
Professor Directing Dissertation

M. Yousuff Hussaini
Professor Co-Directing Dissertation

Robert Van Engelen
University Representative

Nick Cogan
Committee Member

Kyle Gallivan
Committee Member

Approved:

Philip Bowers, Chair, Department of Mathematics

Joseph Travis, Dean, College of Arts and Sciences

The Graduate School has verified and approved the above-named committee members.

This dissertation is dedicated to the memory of my father, Francis J. Duffy, who passed away in September of 2010. Having never graduated from high school himself, and having worked in the bleak Pennsylvania steel industry throughout my early life, he always pushed for and encouraged my continuing education so that I might have a better life than his.

ACKNOWLEDGMENTS

I would first like to thank my advisor Mark Sussman and my co-advisor Yousuff Hussaini from whom I have learned much over the last several years. I would like to thank my other committee members for their help in completing this dissertation and for the valuable insight each has provided in the various topics that have comprised my graduate research. I would like to thank the Florida State University Mathematics Department for supporting me as a teaching assistant in my first few years as a graduate student. I would like to acknowledge the sources of my research assistantship funding, the National Science Foundation under grants DMS-1016381 and DMS-0713256 and the Office of Naval Research under grant N00014-04-1-0709, and I thank both organizations for their vital contributions to the training and education of America's young scientists, engineers and mathematicians like myself. I would like to thank the National Institute of Aerospace and NASA Langley Research Center for supporting me as a visiting researcher, and in particular Dana Hammond and Eric Nielsen of the Computational AeroSciences Branch. Finally, I would like to thank my family, my friends and my soon to be wife Melissa for just always being there for me through these difficult years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this dissertation we provide new numerical algorithms for use in conjunction with simulation based design codes. These algorithms are designed and best suited to run on emerging heterogenous computing architectures which contain a combination of traditional multicore processors and new programmable many-core graphics processing units (GPUs). We have developed the following numerical algorithms (i) a new Multidirectional Search (MDS) method for PDE constrained optimization that utilizes a Multigrid (MG) strategy to accelerate convergence, this algorithm is well suited for use on GPU clusters due to its parallel nature and is more scalable than adjoint methods (ii) a new GPU accelerated point implicit solver for the NASA FUN3D code (unstructured Navier-Stokes) that is written in the Compute Unified Device Architecture (CUDA) language, and which employs a novel GPU sharing model, (iii) novel GPU accelerated smoothers (developed using PGI Fortran with accelerator compiler directives) used to accelerate the multigrid preconditioned conjugate gradient method (MGPCG) on a single rectangular grid, and (iv) an improved pressure projection solver for adaptive meshes that is based on the MGPCG method which requires fewer grid point calculations and has potential for better scalability on hetergeneous clusters. It is shown that a multigrid - multidirectional search (MGMDS) method can run up to 5.5X faster than the MDS method when used on a one dimensional data assimilation problem. It is also shown that the new GPU accelerated point implicit solver of FUN3D is up to 5.5X times faster than the CPU version and that the solver can perform up to 40% faster on a single GPU being shared by four CPU cores. It is found that GPU accelerated smoothers for the MGPCG method on uniform grids can run over 2X faster than the non-accelerated versions for 2D problems, and that the new MGPCG pressure projection solver for adaptive grids is up to 4X faster than the previous MG algorithm.

CHAPTER 1

INTRODUCTION

Traditional high performance computers containing advanced multi-core processors are increasingly being mated with add on specialty hardware such as cell processors, field programmable gate arrays (FPGAs), and graphics processing units (GPUs), creating new breeds of heterogenous computer architectures [23]. The emergence of these heterogeneous many-core computing platforms has allowed for the creation of a new wave of software that exploits an additional level of fine grain parallelism not available to traditional high performance computers. When properly programmed, these systems have given way to drastic reductions in computation time for problems across the spectrum of science and engineering. In this work we develop algorithms designed to exploit the unprecedented levels of mass parallelism [23] these hybrid architectures can provide on the path towards exascale computing. The primary mathematical focus of this dissertation is the development of novel simulation based design (SBD) algorithms that are scalable on heterogeneous platforms. The following items have been developed: (i) a gradient free 'black box' optimizer for PDE constrained optimization that is trivially parallelizable and more scalable than adjoint methods which converge more rapidly. (ii) CFD solvers that exploit GPUs and MPI. (iii) a new MGPCG algorithm for block structured adaptive grids that requires fewer grid point calculations than the previous method and is intuitively more scalable on heterogeneous computer architectures. In addition to heterogenous scalability, another underlying theme found in this work is the use of multigriding strategies which are used to accelerate both a gradient free optimizer and the pressure projection solver of an incompressible multiphase flow solver. Multigrid methods together with block structured data storage enable more efficient use of cache making them well suited for heterogeneous computers. Since this work combines research from distinct areas, we will provide some motivation from each.

1.1 Motivation: Gradient Free Optimization

Numerical optimization is a broad field that can essentially be broken down into two basic categories, derivative and derivative free. Derivative based methods draw directly from calculus where the minimum (or maximum) of a function can be found simply by finding the point where the derivative or gradient is zero. Numerical optimization methods that are founded in locating this point by using approximations to the derivative can be considered rather efficient even in the slowest converging case (steepest descent) when given

the alternative. It is, however, sometimes the case that gradient information is not available, and less efficient alternative methods for numerically finding optimal points becomes necessary. Gradient free methods draw upon various inspirations for locating an optima, from simply checking many random points (Monte Carlo methods) to the physical process of cooling (simulated annealing) to the biological process of evolution (genetic algorithms). Gradient free methods have many redeeming qualities, most notably that they tend to be highly robust and are trivially parallelizable in many cases. Unfortunately, gradient free methods all seem to fall well short of their derivative based counterparts when considering the convergence rate, and so we would consider methods which can significantly reduce the solution time for these methods highly desirable.

1.2 Motivation: Towards Computation at the Exascale

High performance computing systems are undergoing a rapid shift towards the coupling of add-on hardware, such as graphics processing units (GPUs), Cell processors and field programmable gate arrays (FPGAs), with traditional multicore CPU's. Hybrid clusters possess unprecedented price to performance ratios and high energy efficiency, having placed in the top supercomputer rankings [1], and dominated the Green500 list [2]. We note that the Chinese Tianhe-1A, a supercomputer employing Nvidia Fermi architecture GPUs, recently surpassed the Cray based 'Jaguar' of Oak Ridge National Laboratory as the worlds fastest computer [1]. GPU clusters are also being used for research in large scale hybrid computing in the U.S., including the Lincoln Tesla cluster at the National Center for Supercomputing Applications [49]. Plans to build many more of these machines in the very near future are already underway, and while new codes can be developed with this in mind, an emerging challenge in this modern heterogeneous computing landscape is updating old software to make best use of newly available hardware.

An exascale roadmap [23] has been developed by leading computing researchers from academia and industry to address the issues involved with the transition from petascale to exascale computing. While this report discusses numerous issues including the computing infrastrcture, power awareness and fail safe algorithms for many component systems, some important highlights we would like to emphasize from [23] include

- Exascale systems are expected to have a huge number of nodes, with many-core parallelism existing within each node via accelerators such as GPUs, yielding a heterogeneous computing environment.
- Memory heirarchies will be complex, programming models will need to exploit data locality.
- There is a large body of existing scalable applications that need to be migrated towards exascale.
- Exascale computing provides new opportunities for multi-scale, multi-physics and multi-disciplinary applications.

- Algorithms will need to be architecturally aware - adapting to possible heterogeneous environments that they find themselves in.

While it is in general true that a code will see maximum benefit if it is designed from the ground up with available GPUs in mind, there are many widely used legacy codes whose development has spanned multiple decades, and for which this is not an option. In these scenarios, the use of accelerator models in which tasks with large amounts of data parallelism are ported to GPU code may provide the best solution. While some GPU codes can demonstrate multiple orders of magnitude speedup, additional constraints will likely prevent existing high level computational fluid dynamics (CFD) codes from realizing these gains, particularly when they have already been highly optimized for large scale parallel computation, or when the underlying spatial discretization is unstructured. Additionally, architectures with many CPU cores typically need to share GPU resources leading to further limits on the increased performance of a code utilizing an entire cluster. In this work we will examine the acceleration of two CFD codes. We will describe the acceleration of NASA's FUN3D code with a novel GPU distributed sharing model, and discuss the current challenges that face production level unstructured CFD codes in accelerator based computing environments. We will also cover the acceleration of a structured level set - volume of fluid code which will be used with a gradient free optimization algorithm to carry out CFD simulation based design.

1.3 The Bigger Picture: Advancing Simulation Based Design in CFD

The key motivation in advancing simulation based design capabilities is to reduce the time to discovery of new technology. Simulation based design in CFD combines an underlying numerical optimization method with a CFD solver to produce an optimal design. Assuming the CFD simulation is non-trivial, the key bottleneck in the process is always the CFD solver, as it will dominate the computational requirements of the algorithm at every iteration. The other key factor in the process is then the optimization method used since it will determine the number of iterations required for convergence. Considering all other computations negligible to those of the CFD solver, the computational time of the algorithm will equal the CFD solve time multiplied by the number of calls to the solver per iteration multiplied by the number of iterations required for convergence. With this in mind, we find it desirable to develop state of the art techniques which can reduce the CFD solve time and acclerate the convergance rate of the optimizer, leading to multiplied gains in the speed of the overall SBD code.

1.4 Background and Literature Review

In this paper we present a study of some numerical algorithms suitable for parallel architectures employing GPUs as add on hardware, with the ultimate goal of accelerating both simulation times and the solution time of complex simulation based based design problems. This process allows for a reduction in the time to discovery for various problems

of importance in science and engineering. The two key areas of focus in the simulation aspect presented in this dissertation are aerodynamic and incompressible multiphase flows. One such simulation based design application that will directly benefit from this work is the design of microfluidic structures.

1.4.1 Numerical Optimization and Simulation Based Design

Simulation based design for CFD applications is essentially an optimization problem dependend on a flow solver. To give an adequate background on the topic, we will disccuss the closely related field of shape optimization in the context of CFD applications. A general shape optimization problem takes on a form similar to that of a typical constrained minimization problem, however, in shape optimization the cost function will be directly related to the effect of a state variable on the shape of the object undergoing optimization. For example, in a fluid dynamics setting the state variables are given by the governing equations that describe the flow the object is subject to, e.g. Navier Stokes, Euler, etc., and so the constraints are the flow equations themselves. In this work the PDE constraint will be given by the linear advection equation and the nonlinear Burger's equation for 1D data assimilation problems, the Navier-Stokes equations for incompressible multiphase flow, or the Navier-Stokes equations for compressible flow in complex geometries. We will use

$$\begin{aligned} min \quad & J(u, \phi) \\ subject\ to \quad & N(u, \phi) \end{aligned}$$

as the model problem, where J is the cost function, N is the set of governing equations, u is the vector of state variables and ϕ is the design variable which may be a shape (or discretely a set of parameters which make up the shape), or an initial condition as in data assimilation problems. As we will use model problems from the field of data assimilation for developing our numerical optimization strategies, it is also important to discuss this subject here as well. The field of data assimilation can be considered very broad, encompassing all attempts to where the fitting of data to models occurs. We will however narrow this scope to more CFD based topics such as numerical weather prediction and ocean modelling, particularly areas that traditionally use adjoint based methods. A brief discussion is given here, some good overviews of the topic can be found in [33, 66]. The two main ingredients for data assimilation in this realm are observational data and a dynamic model, and so we can view this as a PDE constrained optimization problem. To this end, our cost function will represent a misfit between the observed data and the solution produced by the governing equations, e.g. the fluid dynamic model. The cost function for a problem in this realm in general can be viewed as

$$min \quad J(u) = f(u - u_{obs}).$$

Cost functions in shape optimization can be very similar, leaving data assimilation problems with known solutions as good test problems for developing our optimization strategy. In shape optimization we have a shape that we wish to optimize with respect to some cost function and satisfying a set of constraints. For example, suppose we wanted to find an airfoil that gives an optimal pressure distribution at low speeds, then the shape is the airfoil, the cost function is some measure of the pressure distribution (usually in integral form) and the governing Euler equations with appropriate boundary conditions provide the constraints. Some cost function examples in the context of the optimization of a surface ship hull as provided by Ragab in [64, 65] include

$$J = \int_{BS} p n_x dS \qquad (Wave\ Resistance)$$

$$J = \int_{FS} (\eta - \eta_d)^2 dS \qquad (Target\ Wave\ Pattern)$$

$$J = \frac{1}{2} \int_{BS} (p - p_d)^2 dS \qquad (Target\ Pressure\ Distribution)$$

where BS represents the body surface (ship) and FS represents the free surface (water surface). P is pressure and η is the wave pattern, the d subscript represtents the target for each.

One technique used for optimization in both shape design and data assimilation is the discrete adjoint method. Adjoint methods provide a better way for computing gradients (for use in gradient based optimization methods) than direct computation when there are a large number of design variables, since the computational expense is essentially independent of the number of design variables. Popular in the field of optimal control theory, the use of adjoint methods in aerodynamic shape optimization was first proposed by Jameson and Pironneau who have since contributed numerous works to the field. Ragab has also used continuous adjoints in the optimization of surface ship hulls [64, 65], an area of application for two phase incompressible flow. Though some research utilizing other methods such as genetic algorithms [11, 36] and sequential quadratic programming [74] has been performed along with the work of Ragab, we believe there is area for improvement. Another possible application is the optimization of coastal structures and sea floors to minimize beach erosion, as has been studied in [40] and [41] which claims that no other attempts have been made to apply shape optimization techniques in this area. These works also utilize genetic algorithms which can escape from local minima unlike the gradient based methods.

Aerodynamic shape optimization provides a great knowledge base for methods dedicated to the optimization of shapes subject to fluid flow. There has been extensive research done in many aspects of this field, from 2-D airfoils to complete 3-D aircraft configurations subject to various flows. Some of the earleist numerical optimization for design in an aerodynamic setting was performed using finite difference computations of gradients for 2-D wing profiles in transonic flows [39], from which studies of more complex configurations in

sub and supersonic flows followed. Though there do exist some works still utilizing direct finite difference computation of the gradient (see [35]), the vast majority have moved on to more sophisticated methods. As mentioned earlier, the adjoint based work was pioneered by Jameson and Pironneau who have been joined by numerous others, working with both discrete [12, 55, 57, 59, 60] and continuous [9, 44, 45] versions of the method. Adjoint based data assimilation is popularly used in atmospheric and ocean modeling, where observational data is used to produce predictive simulations. Of interest here is work which has been done involving free surface flows. The two papers by fang et al [24, 25] provide some guidance to this aspect, particularly in the use of adaptive mesh methods with a free surface. A paper that is important to note here is [56] where fluid control simulations are carried out using a level set based discrete adjoint method. We initially found the method contained here attractive for our optimization goals, but these simulations were produced for smoke and we see that the fluid densities will be discontinuous across some multi-phase fluid interfaces (such as liquid-gas) and so computing the derivative required for the adjoint solution becomes problematic. Indeed, in the author's discussion they note that for water their projection and diffusion operations are discontinuous and the results prove to be less accurate. For these reasons, we will seek derivative free methods which will allow us to pursue optimization problems involving sharp interfaces without the need for concern over the possible need for problematic gradient computations that may arise.

1.4.2 Accelerating CFD Codes on Hybrid Many-Core Architectures

There have been an excess of papers demonstrating the impressive performance gains GPU computing can provide to applications across a wide range of science, engineering and finance disciplines. For brevity, we will mention here just those most closely related to accelerating the CFD codes we are interested in, namely a coupled level-set and volume of fluid code (CLSVOF - structured, multiphase, incompressible Navier-Stokes) and the NASA FUN3D code (unstructured, single phase, compressible/incompressible Euler and Navier-Stokes). Some relevant works that have ported CFD codes to GPUs can be found in [22] and [15] who achieved significant speedups up to 40X for the compressible Euler equations, the latter on an unstructured grid. While both demonstrated significantly faster GPU code times, we note that neither solved the full compressible Navier-Stokes equations, and both used explicit Runge-Kutta solvers more suitable for acceleration than the implicit algorithm in FUN3D. In addition, neither of these works considered scaling to multiple processors, giving single-core, single-GPU results against individual CPUs. The report by Jespersen [46] describes the porting of NASA's OVERFLOW code to the GPU; but again only for a single core setup and also in a structured grid environment with more favorable memory access patterns. Jespersen [46] replaced the 64-bit implicit SSOR solver with a 32-bit GPU Jacobi solver, resulting in a 2.5-3X speedup for the solver, and an overall code wall clock time reduction of 40%, representing more realistic results for a high end CFD code. The work by Cohen and Molemaker [13] is an excellent resource for anyone considering the development or porting of CFD code for GPU computation, and includes double precision considerations, an important aspect of CFD in general. We note that several incompressible works using multi-GPU architectures have been reported [32, 43, 77], but none of these

include a treatment of adaptive grids. Reviewing the literature one can see that, until recently, the vast majority of the research in this field has focused solely on utilizing a single CPU core with one or more GPUs as the primary source of computational power. Conversely, only a minor amount of research has considered larger scale computations on many cores with supporting GPUs, or even sharing a GPU among several CPU cores. The latter is due to the fact that until the recent release of Nvidia's latest architecture 'Fermi' cards, concurrent kernel execution was not possible, and hence cores had to wait serially for access to a shared GPU. Gödekke and Strzodka [28–31], among others, have made a substantial contribution to large scale computing on graphics clusters over the last several years, with particular focus to the areas of multigrid and finite element methods. We note some other recent works that have considered using CFD codes in a GPU cluster environment including [63] where a 16 GPU cluster was used to achieve 88X speedup over a single core on a block structured MBFLO solver, and [42] who achieved 130X speedup over 8 CPU cores for incompressible flow with 128 GPUs on the aforementioned Lincoln Tesla cluster. At this time we know of no published works employing any type of GPU sharing model other than our own [21].

CHAPTER 2

PDE CONSTRAINED OPTIMIZATION

The primary contribution of this work is the advancement of current methods involved in the complex optimization of problems governed by PDE systems. While reducing the computational cost of the flow solvers involved is vital to this end, the most important aspect here is in the reduction of the number of iterations required for convergence of the optimization method itself. The physical nature of the majority of the problems we are concerned with necesitate the use of slow converging gradient free methods, and so techniques which can greatly reduce the number of iterations required to reach an optimal design are desirable. While we will primarily focus on derivative free optimization methods, it is important to also discuss gradient based methods which are a better choice for problems where gradient computation is feasible, such as in single phase aerodynamics where adjoint methods dominate. In this chapter we will describe both the continuous and discrete adjoint methods along with the derivative free multidirectional search (MDS) method. To demonstrate the effectiveness of both the discrete adjoint method and MDS, we will develop model data assimilation problems to test both and will also describe a multigriding strategy which we have used to reduce the number of iterations required for convergence of the MDS algorithm when applied to these problems.

2.1 Adjoint Methods

Both continuous and discrete adjoint methods have shared much popularity in simulation based design. In [80] an extensive study of discrete and continous adjoint methods is reported. Some findings there include that the continuous adjoint equations are not unique and in fact they can vary due to choices in the derivation which could affect the optimization procedure, that the continuous adjoint may destroy symmetry properties for optimality conditions required in some optimization procedures, and that continuous adjoints may have degraded convergence properties over the discrete method, or may not converge at all. They also found that continuous adjoints can provide more flexibility and note that the solution provided by the discrete version is no more accurate than the discretization used to solve the state equation. The discrete adjoint will be used here due to the more systematic approach that can be taken to problem solving, however, a brief description of the continuous method is provided below.

2.1.1 Continuous Adjoint Method

In the continuous adjoint method, a constrained optimization problem is transformed into an unconstrained type by forming a Lagrangian, with the Lagrange multipliers posing as the adjoint variables. The adjoint problem can be determined through either a variational formulation or by finding shape derivatives. In the variational formulation, one could represent the Lagrangian, as done in [65], by a sum of integrals of the general form

$$L = \sum_i \int G_i(u, \phi)$$

and compute the variation in the Lagrangian, L, subject to a variation in the state variable u by

$$\delta L = \sum_i \int \left. \frac{\partial G_i}{\partial \phi} \right|_u \delta\phi + \sum_i \int \left. \frac{\partial G_i}{\partial u} \right|_\phi \delta u.$$

The computation of δu is undesireable and so the Lagrange multipliers are defined in order to eliminate the term, i.e. the adjoint problem is determined so that it satisfies

$$\sum_i \int \left. \frac{\partial G_i}{\partial u} \right|_\phi \delta u = 0.$$

After solving the adjoint problem, the adjoint solutions are then used to compute the gradient

$$\frac{dL}{d\phi} = \sum_i \int \left. \frac{\partial G_i}{\partial \phi} \right|_u$$

which is used for the optimization. The procedure for finding shape derivatives is similar but is not discussed here.

2.1.2 Discrete Adjoint Method

In the discrete adjoint method, we essentially solve a set of governing equations forward and then solve the adjoint problem backwards in time in order to acquire the adjoint variables. In the case of linear governing equations, only the final solution data from the forward solve is required, however, in the case of nonlinear governing equations the solution data must be stored at every time step which can become restrictive for large problems. We believe that this cost is not enough to detract from a method which can be systematically computed due to a lack of need for problem by problem analysis as is required in the

continuous case. The treatment of boundary conditions is also an issue with the continuous method, and as [55] notes, it is much easier to obtain these conditions for the adjoint solver in the discrete version. An interesting solution method for the discrete adjoint problem is given in [82] where a Monte Carlo method is used to solve the adjoint problem forward in time for an explicit discretization of a time dependent problem. This allows the adjoint solution to be solved at the same time as the original problem without the need for storing the solution values at each time step.

Consider the general minimization problem

$$\begin{aligned} min \quad & L(u,\phi) = J(u,\phi) \\ subject\ to \quad & N(u,\phi) = 0 \end{aligned}$$

where J is the cost function, N is the governing equation, u is the state variable and ϕ is the design variable. In order to use a gradient based method (like steepest descent) to perform the minimization, one will need to compute

$$\frac{dL}{d\phi} = \frac{\partial J}{\partial u}\frac{du}{d\phi} + \frac{\partial J}{\partial \phi}. \tag{2.1}$$

The $\frac{du}{d\phi}$ term can be determined by looking at the derivative of the governing equation.

$$\frac{dN}{d\phi} = \frac{\partial N}{\partial u}\frac{du}{d\phi} + \frac{\partial N}{\partial \phi} = 0$$

which implies

$$\frac{\partial N}{\partial u}\frac{du}{d\phi} = -\frac{\partial N}{\partial \phi}$$

$$\frac{du}{d\phi} = \left(\frac{\partial N}{\partial u}\right)^{-1}\left(-\frac{\partial N}{\partial \phi}\right). \tag{2.2}$$

So from 2.1 and 2.2 we have

$$\frac{dL}{d\phi} = \frac{\partial J}{\partial \phi} - \frac{\partial J}{\partial u}\left(\frac{\partial N}{\partial u}\right)^{-1}\left(\frac{\partial N}{\partial \phi}\right).$$

Now, we will instead solve the adjoint problem by finding λ where

$$\lambda = \left[\frac{\partial J}{\partial u}\left(\frac{\partial N}{\partial u}\right)^{-1}\right]^T$$

and the problem of finding the gradient breaks down to

$$Solve \qquad \left(\frac{\partial N}{\partial u}\right)^T \lambda = \left(\frac{\partial J}{\partial u}\right)^T \tag{2.3}$$

$$Compute \qquad \frac{dL}{d\phi} = \frac{\partial J}{\partial \phi} - \lambda^T \frac{\partial N}{\partial \phi} \tag{2.4}$$

This abstract formulation can make the actual implementation seem like a bit of a mystery, so to clarify things two examples are now given.

2.1.3 Linear Example Problem

Suppose we wish to solve the one dimensional advection equation on $x \in$ [-1,1] where we do not know the initial condition, but do know the solution data at time T. We can recover the initial condition by treating this as a minimization problem.

$$min \quad J(u,\phi) = \int_{-1}^{1} (u(x,T) - u_d(x,T))^2 dx$$

$$\begin{aligned} subject\ to \quad N(u,\phi) &= u_t + a u_x = 0 \\ u(x,0) &= g(x) \\ u(-1,t) &= c \end{aligned}$$

In this case, the design variable ϕ is the initial condition $u(x,0)$, u_d is the observed solution

data and c is a constant. We will assume a is positive and use a simple forward in time, backward in space discretization of the governing equation

$$N_i^k = \frac{u_i^k - u_i^{k-1}}{\Delta t} + a\left(\frac{u_i^{k-1} - u_{i-1}^{k-1}}{\Delta x}\right) = 0 \tag{2.5}$$

and approximate J at the final time step m by

$$J = \Delta x \sum_{i=0}^{n} \left(u_i^m - u_{di}\right)^2 .$$

Now, in terms of the discretization we have

$$N = [N_{-1}^0, N_0^0, ...N_n^0, N_{-1}^1, ...N_n^m]^T$$

$$u = [u_{-1}^0, u_0^0, ...u_n^0, u_{-1}^1, ...u_n^m]^T$$

$$\lambda = [\lambda_{-1}^0, \lambda_0^0, ...\lambda_n^0, \lambda_{-1}^1, ...\lambda_n^m]^T$$

$$G = [G_{-1}^0, G_0^0, ...G_n^0, G_{-1}^1, ...G_n^m]$$

where the -1 subscript represnts the left 'ghost' boundary point needed, and G represents $\frac{\partial J}{\partial u}$. To solve equation 2.3 we find

$$\frac{\partial N}{\partial u} = \begin{pmatrix} \frac{\partial N_{-1}^0}{\partial u_{-1}^0} & \frac{\partial N_{-1}^0}{\partial u_0^0} & .. & \dots & \frac{\partial N_{-1}^0}{\partial u_n^m} \\ \frac{\partial N_0^0}{\partial u_{-1}^0} & \frac{\partial N_0^0}{\partial u_0^0} & & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot \\ \frac{\partial N_n^m}{\partial u_{-1}^0} & \cdot & \cdot & \cdot & \frac{\partial N_n^m}{\partial u_n^m} \end{pmatrix}$$

and using the initial condition $N_i^0 = u_i^0$ along with the embedded boundary condition of

$$N_{-1}^k = u_{-1}^k - c = 0$$

this becomes

$$\frac{\partial N}{\partial u} = \begin{pmatrix} I & 0 & 0 & \cdot & \cdot & 0 \\ A1^1 & A2^1 & 0 & \cdot & \cdot & 0 \\ 0 & A1^2 & A2^2 & & & \cdot \\ \cdot & & \cdot & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot & 0 \\ 0 & \cdot & & & A1^m & A2^m \end{pmatrix}$$

where $A1^k$ and $A2^k$ are $(n+1) \times (n+1)$ block matrices (with superscripts denoting the time level) given by

$$A1^k = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ -\frac{a}{\Delta x} & -\frac{1}{\Delta t} + \frac{a}{\Delta x} & 0 & & \cdot \\ 0 & -\frac{a}{\Delta x} & -\frac{1}{\Delta t} + \frac{a}{\Delta x} & & \cdot \\ \cdot & \cdot & \cdot & & \\ \cdot & & \cdot & \cdot & \\ \cdot & & & \cdot & \cdot \\ 0 & \cdots & & -\frac{a}{\Delta x} & -\frac{1}{\Delta t} + \frac{a}{\Delta x} \end{pmatrix}$$

$$A2^k = \begin{pmatrix} 1 & \cdots & & 0 \\ 0 & \frac{1}{\Delta t} & & \cdot \\ \cdot & & \frac{1}{\Delta t} & \cdot \\ \cdot & & \cdot & \\ 0 & \cdots & & \frac{1}{\Delta t} \end{pmatrix}$$

.

For this problem, solving the adjoint turns out to require block solves of sparse structrured matrices allowing simple direct solvers to be used.

Algorithm 1 Discrete Adjoint Solution, 1-D Linear Advection

$A2\lambda^m = G^m$
for i=m-1 to 1 **do**
  $A2\lambda^i = G^i - A1^T\lambda^{i+1}$
end for
$\lambda^0 = G^0 - A1^T\lambda^1$

Once the adjoint variable has been found, the gradient can be found as noted in the previous section by computing 2.4, and the initial condition is updated according to the

minimization method used. It should be noted that this problem, like most inverse problems, is numerically ill-posed. As the effects of dissipation from the discretization become too great (e.g. increasing the spatial step size), much different oscillating solutions are allowed (and hence we have numerical ill-posedness) which can smooth out to match the data at time t when advected forward. This problem can be remedied by the addition of another term to the cost function, for example the term $(\nabla u(x,0))^T(\nabla u(x,0))$ penalizes the cost function when large jumps in the final solution (the recovered IC) are present, thus eliminating the presence of oscillations. A term used by the data assimilation and optimal control communities, known as the background or regularization term respectively, can be given by $(u(x,0) - u_B(x,0))^T(u(x,0) - u_B(x,0))$ where u_B is the 'background', i.e. the previous initial guess. Both of these terms can also benefit from a scaling parameter β.

2.1.4 Nonlinear Example Problem

For the nonlinear example we will solve essentially the same problem, but this time subject to the classical Burgers equation, which possesses many of the features of more complex fluid dynamics models.

$$\begin{aligned} min \quad & J(u,\phi) = \int_{-1}^{1} (u(x,t) - u_d(x,t))^2 dx \\ \\ subject\ to \quad & N(u,\phi) = u_t + uu_x - \mu u_{xx} = 0 \\ & u(x,0) = g(x) \\ & u(-1,t) = c \\ & u_x(1,t) = 0 \end{aligned}$$

Here, we will use a Lax-Freidrichs discretization for N_i^k

$$\frac{u_i^k - \frac{1}{2}(u_{i+1}^{k-1} + u_{i-1}^{k-1})}{\Delta t} + \frac{(u_{i+1}^{k-1})^2 - (u_{i-1}^{k-1})^2}{4\Delta x} - \mu \left(\frac{u_{i+1}^{k-1} - 2u_i^{k-1} + u_{i-1}^{k-1}}{\Delta x^2} \right) = 0 \tag{2.6}$$

The boundary conditions are discretized as

$$\begin{aligned} N_{-1}^k &= u_{-1}^k - c = 0 \\ N_{n+1}^k &= \frac{u_{n+1}^k - u_n^k}{\Delta x} = 0 \end{aligned}$$

For this example, the observational data is produced by solving the problem forward using the above discretization along with $c = 0.5$ and $g(x) = 0.5 + 0.5e^{-10 \cdot x^2}$ and so there is no observational error present, i.e. observational data is exact since it is generated from the same discretization used in our model with the known solution (the IC $g(x)$) we are testing against. For the solution of the inverse problem, poor initial data is chosen so as to demonstrate the ability of the discrete adjoint method, and so we take $g(x) = 0.5$.

2.1.5 A Matrix Free Representation of the Discrete Adjoint Problem

From the previous example, one could imagine that as the number of dimensions increase and the governing equations (along with their discretizations) become more sophisticated, the resulting derivative matrix becomes very large, and (despite its sparseness) storage becomes very undesireable. In the case of realistic computational fluid dynamics problems, the use of traditional sparse solvers may even be out of the question. A few methods have been proposed to combat this issue, for example [82] uses a Monte Carlo method to compute the adjoint, as previously mentioned, without storing the Jacobian or the previous solution states, but here we will look to take advantage of the inherent structure of the Jacobian. For this reason, we want to seek a matrix free representation that will allow us to compute the gradient. It can easily be seen that the matrix $\frac{\partial N}{\partial u}$ will always take on a desireable block lower triangular shape, and in fact it can always (at worst) take the form

$$\frac{\partial N}{\partial u} = \begin{pmatrix} I & 0 & 0 & \cdot & \cdot & 0 \\ J_0^1 & J_1^1 & 0 & \cdot & \cdot & 0 \\ J_0^2 & J_1^2 & J_2^2 & & & \cdot \\ \cdot & & \cdot & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot & 0 \\ J_0^m & \cdot & \cdot & & J_{m-1}^m & J_m^m \end{pmatrix}$$

Where $J_i^k = \frac{\partial N^k}{\partial u^i}$. It is important to note that this notation extends to all dimensions, to be specific, in 1-D

$$J_i^k = \begin{pmatrix} \frac{\partial N_{-1}^k}{\partial u_{-1}^i} & \frac{\partial N_{-1}^k}{\partial u_0^i} & \cdot\cdot & \cdots & \cdot & \frac{\partial N_{-1}^k}{\partial u_n^i} \\ \frac{\partial N_0^k}{\partial u_{-1}^i} & \frac{\partial N_0^k}{\partial u_0^i} & & & \cdot & \\ \cdot & & \cdot & & & \cdot \\ & & & \frac{\partial N_j^k}{\partial u_j^i} & & \\ \cdot & & & & & \\ & & & & \cdot & \cdot \\ \frac{\partial N_n^k}{\partial u_{-1}^i} & \cdot & \cdot & \cdot & \cdot & \frac{\partial N_n^k}{\partial u_n^i} \end{pmatrix}$$

Then in 2-D, each entry of the sub-Jacobian would become a sub-sub-Jacobian matrix

$$\frac{\partial N_j^k}{\partial u_j^i} = \begin{pmatrix} \frac{\partial N_{j,-1}^k}{\partial u_{j,-1}^i} & \frac{\partial N_{j,-1}^k}{\partial u_{j,0}^i} & .. & \dots & . & \frac{\partial N_{j,-1}^k}{\partial u_{j,n}^i} \\ \frac{\partial N_{j,0}^k}{\partial u_{j,-1}^i} & \frac{\partial N_{j,0}^k}{\partial u_{j,0}^i} & & & . & \\ . & & . & & & . \\ . & & & \frac{\partial N_{j,p}^k}{\partial u_{j,p}^i} & & \\ . & & & & . & . \\ \frac{\partial N_{j,n}^k}{\partial u_{j,-1}^i} & . & . & . & . & \frac{\partial N_{j,n}^k}{\partial u_{j,n}^i} \end{pmatrix}$$

and so on. A useful consequence of this setup is that J_i^i can always be made to be at least diagonal 'almost everywhere', if not the identity matrix. This is dependent on the choice of boundary conditions, with any non diagonal elements occuring in either the first or last row. It is also likely that this will be a banded lower triangular block matrix rather than a full lower triangular matrix depending on the time discretization, and the number of time steps. For example, the problem in the previous section resulted in a single lower band while the same problem utilizing a leap frog scheme would have two lower bands. Now, in order to solve 2.3 for λ^k we have a true back solve since the diagonal blocks are themselves diagonal matrices. So here, the solve looks just like a more generalized version of that found in the example problem.

Algorithm 2 Discrete Adjoint Solution, General Explicit Method

$J_m^m \lambda^m = G^m$
for i=m-1 to 1 **do**
  $J_i^i \lambda^i = G^i - \sum_{k=i+1}^m J_i^{k^T} \lambda^k$
end for
$\lambda^0 = G^0 - \sum_{k=1}^m J_0^{k^T} \lambda^k$

At this stage, the setup seems (and is) rather nice for simple problems, as the user can determine the individual sub-Jacobian matrices needed to solve the problem. This can, however, quickly become much more complicated once the governing equations and their associated discretizations become more complex, e.g. higher dimensions, systems of equations, higher order methods, etc. So now, we would like to find an easier way to compute the sub-Jacobians. For this we will define the entries of J_i^k in terms of the approximate Fréchet derivative so that

$$J_i^k e_j = \frac{N^k(u_j^i + \epsilon e_j) - N^k(u_j^i - \epsilon e_j)}{2\epsilon}$$

where e_j is the j^{th} column of the identity matrix. So for example in 1-D, the (j, p) entry

of J_i^k is $\frac{\partial N_j^k}{\partial u_p^i}$ and we would have

$$\frac{\partial N_j^k}{\partial u_p^i} = \frac{N_j^k(u_p^i + \epsilon) - N_j^k(u_p^i - \epsilon)}{2\epsilon}$$

As an example, to compute ${J_i^k}^T$ for the discretization 2.5 we could use algorithm 3.

Algorithm 3 Sub-Jacobian Computation

for p=1 to n **do**
  if i=k or i=k+1 **then**
    set $temp1(u) = u$
    set $temp2(u) = u$
    $temp1(u_p^i) = u_p^i + \epsilon$
    $temp2(u_p^i) = u_p^i - \epsilon$
    for j=1 to n **do**
      $N1 = \frac{temp1(u_j^{k+1}) - temp1(u_j^k)}{\Delta t} + a\left(\frac{temp1(u_j^k) - temp1(u_{j-1}^k)}{\Delta x}\right)$
      $N2 = \frac{temp2(u_j^{k+1}) - temp2(u_j^k)}{\Delta t} + a\left(\frac{temp2(u_j^k) - temp2(u_{j-1}^k)}{\Delta x}\right)$
      ${J_i^k}^T(p,j) = \frac{\partial N_j^k}{\partial u_p^i} = \frac{N1 - N2}{2\epsilon}$
    end for
  else
    ${J_i^k}^T(p,j) = \frac{\partial N_j^k}{\partial u_p^i} = 0$
  end if
end for

So we now have a way to compute the Jacobians required in Algorithm 2 that avoids the error prone and tedious task of explicitly determining them. This allows for an automated procedure capable of solving the adjoint problem for even the most difficult of discretization methods. Note that for linear problems, this matrix could be computed once and stored if desired, while for the nonlinear case one needs to utilize the solution data from the forward solve of the governing equation, and so must be recomputed at every step. Note that unlike automatic differentiation (AD), this method does not require any specific problem by problem coding, and only relies on the solution data itself. Since AD will not be used here, we will not describe it but will note that it is a popular method due to its high accuracy, and good treatments of this can be found in [50], [37] and [19].

One should take note that in most methods, a vast majority of the J_i^k matrices will be 0. Indeed, as previously stated it is dependent upon the time discretization. In general, an n^{th} order explicit time discretization that utilizes data from only n previous time steps will result in

$$J_i^k = 0 \qquad if \quad |i-k| > n.$$

We should also note that in the case of Dirichelet boundary conditions, $J_i^i = I$ and thus will not need to be computed. Also note that in a parallel setting, since we only need the matrix vector products $J\lambda$ in the computation, we do not need to store J, only the resultant vector. So J can be computed, multiplied by λ and be immediately discarded, hence reducing storage and communication costs.

The last bit of information needed is how to choose ϵ. In [61] a similar method is used in the Jacobian-free Newton Krylov method, and they state the simplest choice for epsilon as (in the translated context of this paper)

$$\epsilon = \frac{1}{N} \sum_{c=1}^{N} \beta u_c$$

Where N is the total number of grid points and β is a constant with value within a few orders of magnitude of machine epsilon.

2.2 Parallel Derivative Free Methods

While adjoint methods work very well for optimization problems with many design variables, the gradient computation can be problematic for certain classes of problems. In certain multiphase/multiphysics problems, gradients may be difficult or even impossible to compute, and hence derivative free methods must be utilized in there place. It is well known that derivative free methods, in general, converge very slowly and require a large number of function evaluations. In PDE constrained optimization, this factor can be very limiting if the PDE solution is expensive to compute. For these reasons, we seek a method which can compute the PDE solves in parallel, allowing us to exploit the rapidly increasing number of processor cores available to a given architecture. Luckily, most derivative free methods are parallel in nature and many are embarassingly parallel. We have experimented with the use of Monte Carlo methods (MC) and genetic algorithms (GA) as global optimizers, but to this point in our research we have primarily focused on the local Multidirectional search method (MDS) [78]. Though originally intended for unconstrained optimization, we find this method appealing because it is a descent method and was designed for parallel computers. Descent type methods are desirable because they reduce the cost function at every iteration and are guaranteed to converge in many cases. We note that since MDS is a local method, it can become trapped in local extrema, unlike MC or GA.

2.2.1 The Multidirectional Search Method

Partly inspired by the Nelder-Mead simplex method, the multidirectional search method (MDS) [17,78,79] is a descent method whose search directions are guaranteed to be linearly independent. This method is most appropriate in this setting because it is developed, and ideally suited for parallel computation, something that cannot be said of adjoint methods. A key benefit of MDS is that convergence is guaranteed for smooth cost functions [79] and it is more robust than the Nelder-Mead simplex method [78]. A brief description of the general method will now be described.

The multidirectional search method first requires a set of n+1 vertices $\{v_i; i = 0 : n\}$ which form a nondegenerate n-simplex, this can be viewed as the n-dimensional analogue of a triangle, which is in fact a 2-simplex (see figure 2.1). The first step is to compute the value of the cost function for all n+1 vertices, $f(v_i^0)$, and the vertices are then ordered according to lowest cost, starting with the 0 subscript. The superscript gives the iteration number. The n edges connecting the best vertex to the rest provide the set of search directions, which are linearly independent. A reflection step is then taken, in which we find $v_i^1 = 2v_0^0 - v_i^0$ and then calculate $f(v_i^1)$. If a new minimum value is found, then an expansion step is taken to see if an even better point can be found. This is calculated as $v_{e_i}^0 = (1-\mu)v_0^0 + \mu v_i^1$, and if $f(v_{e_i}^0) < f(v_i^1)$, then $v_i^1 = v_{e_i}^0$. If, however, we do not obtain a better point after the reflection step, then we instead perform a contraction in a similar manner, $v_i^{k+1} = (1+\theta)v_0^k - \theta v_i^{k+1}$. The vertices are then reordered and we move on to the next iteration and repeat the process. The algorithm as described in [79] is given in alg. 2.2.1.

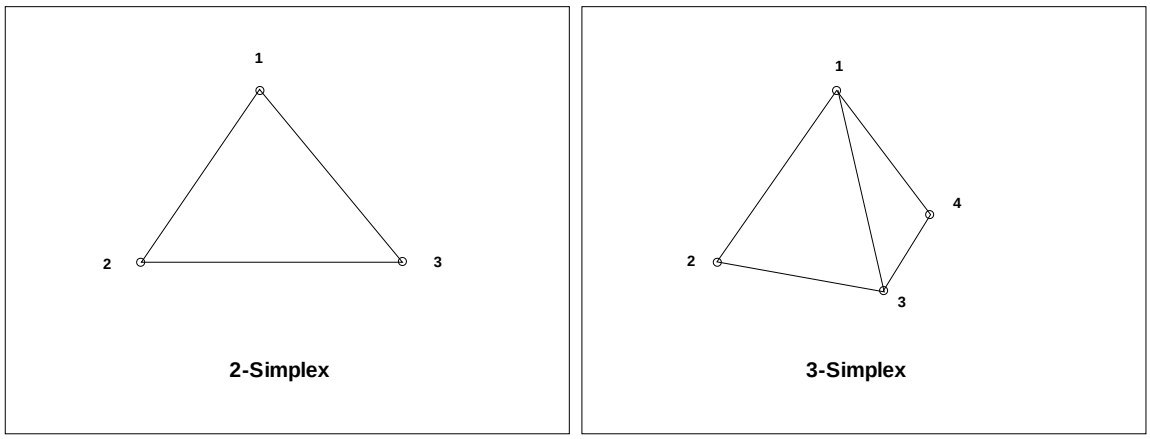

Figure 2.1: 2-Simplex (Left): A two dimensional triangle made by connecting 3 vertices. 3-Simplex (Right): A 3 dimensional tetrahedron made of triangles by connecting 4 vertices

A question arises here of how MDS may be applied to PDE constrained optimization. In the context of the 1-D data assimilation problems of this chapter, suppose we have a

Algorithm 4 The Multidirectional Search Algorithm of Torczon [79].

Given an initial simplex S_0 with vertices $\{u_0^0, u_1^0, ..., u_n^0\}$, $\mu \in (1, +\infty)$ and $\theta \in (0, 1)$
for $i = 0$ to n **do**
  calculate $f(u_i^k)$
end for
$k = 0$
// outer while loop
while stopping criterion is not satisfied **do**
  // find a new best vertex
  $j = min_i\{f(u_i^k) : i = 0, ..., n\}$
  swap u_j^k and u_0^k
  repeat
    Check stopping criterion
    // rotation step
    for $i = 1$ to n **do**
      $r_i^k = u_0^k - (u_i^k - u_0^k)$
      calculate $f(r_i^k)$
    end for
    $e_i^k = u_0^k - \mu(u_i^k - u_0^k)$
    replaced $= (min\{f(r_i^k) : i = 1, ..., n\} < f(u_0^k))$
    if replaced **then**
      // expansion step
      for $i = 1$ to n **do**
        $e_i^k = u_0^k - \mu(u_i^k - u_0^k)$
        calculate $f(e_i^k)$
      end for
      if $min\{f(e_i^k) : i = 1, ..., n\} < min\{f(r_i^k) : i = 1, ..., n\}$ **then**
        // accept expansion
        $u_i^k = e_i^k$ for $i = 1, ..., n$
      else
        // accept rotation
        $u_i^k = r_i^k$ for $i = 1, ..., n$
      end if
    else
      // contraction step
      for $i = 1$ to n **do**
        $c_i^k = u_0^k + \theta(u_i^k - u_0^k)$
        calculate $f(c_i^k)$
      end for
      replaced $= (min\{f(c_i^k) : i = 1, ..., n\} < f(u_0^k))$
      // accept contraction
      $u_i^k = c_i^k$ for $i = 1, ..., n$
    end if
  until replaced
  $k = k + 1$
end while

candidate solution for the recovered IC in vector form. In this scenario we can create a simplex by augmenting the solution with a set of n linearly independent alternate solutions in the form of a scaled identity matrix, and use a multidirectional search on this set in order to find a single optimal solution. We note that simplex-type methods such as MDS are not typically used for problems with a large number of degrees of freedom, or what we would consider here to be design variables. For algorithm 2.2.1, each additional design variable introduced into the problem requires an additional flow solve computation at each cost function evaluation step. For this reason it is important to note here that we intend to use MDS in a design code for problems with a small number of design variables, which will be described later.

2.2.2 A Multigriding Strategy for the Multidirectional Search Method

While we find MDS to be a promising derivative free method with which to carry out our simulation based design goals, we would also like to find ways in which to reduce the number of iterations required to achieve convergence, as each iteration requires a flow solve which can become very costly. One such way to accelerate convergence that has been used on gradient based methods [52, 58] is to use a multigriding strategy. Multigrid methods [10] are well known for accelerating the convergence rates of PDE systems on fine grids when the PDE system's qualitative solution features can be captured using coarse level grids. Multigrid methods should also be effective on nonlinear problems [10] and have a solid research base in CFD applications [20, 38]. Since our proposed MDS algorithm is dependent on such PDE systems, and since success in reducing the number of iterations to convergence for gradient based methods has been achieved by employing multigriding strategies [52, 58], it seems logical to attempt to accelerate our MDS algorithm as well by utilizing a multigriding strategy. In order to test this strategy, we have devolped two MG algorithms for the MDS method (MGMDS) using the MG cycle strategies of figure 2.2. The first uses a V-cycle strategy where one starts with a fine grid approximation to the solution, this method is demonstrated by algorithm 5. As an alternative strategy, we have also used an MGMDS algorithm which starts with a coarse grid approximation using what is refered to as a full multigrid (FMG) cycle, which is described in algorithm 6. We note that while algorithms 5 and 6 are both set up for multiple MG cycles, we currently have no coarse grid correction mechanism and so should be restricted to a single cycle in practice.

2.3 Numerical Results and Discussion

Presented in this section are some numerical findings based on the solution of both a linear and nonlinear test problem formulated using the discrete adjoint method and both the traditional MDS method of Torczon [78] and the novel MGMDS method described in algorithms 5 and 6. For the linear case, the 1-D advection problem is solved, while for the nonlinear case, a similar 1-D problem is posed utilizing Burgers equation in place of the advection equation. Comparisons for the two formulations of each problem are made. We note that in the discrete adjoint case, the solutions utilizing the stored matrix and the computed matrix free versions of the flow Jacobian are identical. We have also found that for

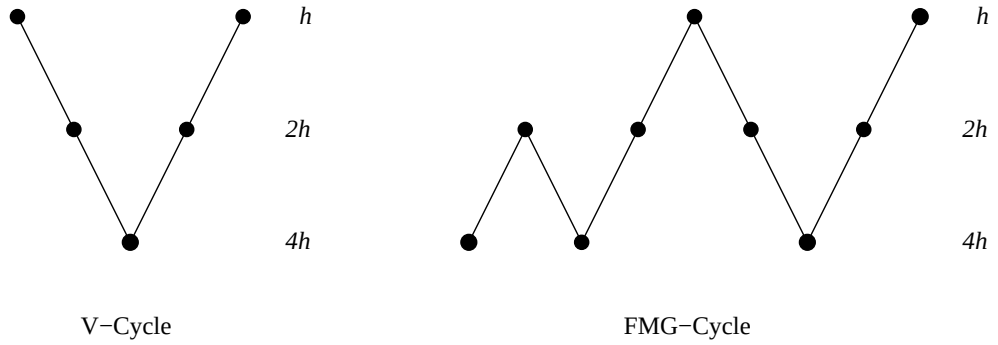

Figure 2.2: Multigrid cycling schemes tested in this work include the V-Cycle and the full multigrid cycle (FMG).

Algorithm 5 A Multigrid-Multidirectional Search Algorithm, V-Cycle Strategy

Given a fine level grid X^N and an initial simplex S^N of solutions on X^N, $\{u_0^0, u_1^0, ..., u_n^0\}$, choose an expansion factor $\mu \in (1, +\infty)$ and a contraction factor $\theta \in (0, 1)$.
while cost $>$ tol and cycles $<$ maxcycles **do**
  $k = N$
  // begin v-cycle
  repeat
    Call MDS($S^k, X^k, \mu, \theta, maxiterates$)
    restrict $S, X, \rightarrow$ k=k-1
  until k=0
  Call MDS($S^0, X^0, \mu, \theta, bottomiterates$)
  repeat
    Call MDS($S^k, X^k, \mu, \theta, maxiterates$)
    prolong $S, X, \rightarrow$ k=k+1
  until $k = N$
  // end v-cycle
end while
if cost $>$ tol **then**
  // MG cycles complete, begin final MDS iterations
  Call MDS($S^N, X^N, \mu, \theta, finaliterates$)
end if

Algorithm 6 A Multigrid-Multidirectional Search Algorithm, FMG Strategy

Given a fine level grid X^N and an initial simplex S^N of solutions on X^N, $\{u_0^0, u_1^0, ..., u_n^0\}$, choose an expansion factor $\mu \in (1, +\infty)$ and a contraction factor $\theta \in (0, 1)$.
while cost $>$ tol and cycles $<$ maxcycles **do**
  Restrict X, S to the coarsest grid $\rightarrow$ k=0
  Call MDS($S^0, X^0, \mu, \theta, maxiterates$)
  $\ell = 1$
  repeat
    for $i = 1$ to ℓ **do**
      prolong $X, S \rightarrow$ k=k+1
      Call MDS($S^k, X^k, \mu, \theta, maxiterates$)
    end for
    for $i = \ell$ to 1 **do**
      restrict $X, S \rightarrow$ k=k-1
      Call MDS($S^k, X^k, \mu, \theta, maxiterates$)
    end for
    $\ell = \ell + 1$
  until $\ell = N$
  for $i = 1$ to N **do**
    prolong $X, S \rightarrow$ k=k+1
    Call MDS($S^k, X^k, \mu, \theta, maxiterates$)
  end for
end while
if cost $>$ tol **then**
  Call MDS($S^N, X^N, \mu, \theta, finaliterates$)
end if

these test cases, the use of the penalty term $(\nabla u)^T(\nabla u)$ and the traditional background, or regularization term in the cost function give essentially the same result, producing solution errors that are consistently within the same order of magnitude.

2.3.1 Linear Advection Problem

For the linear advection example, we demonstrate a recovery of the initial conditions in a 1-D data assimilation problem whose solution possesses a steep gradient, and starting with a poor initial guess. The problem is

$$min \quad J(u,\phi) = \int_{-1}^{1} (u(x,T) - u_d(x,T))^2 dx + \beta(\nabla u(x,0))^T(\nabla u(x,0))$$

$$subject\ to \quad N(u,\phi) = u_t + au_x = 0$$

The data, u_d, is generated by advecting a step function IC forward using the same discretization as for the problem solution. The discretization method is again the forward Euler method, the exact IC we wish to recover is

$$g(x) = \begin{cases} 1.0 & -0.5 < x < 0 \\ 0.5 & Otherwise \end{cases}$$

.

We start with an initial guess of $g(x) = 0$. The plots in figures 2.3 to 2.10 show the actual data, and the solution u after convergence. The plots to the left are at time t=0, and show the exact IC we wish to recover (solid line), and the IC recovered using the discrete adjoint and MDS methods (dash dot line). The plots on the right are at time t=1.0, and show the advected solutions from these starting IC's. These plots all have a value of $\beta = 1.0$ for the cost function.

2.3.2 Nonlinear Burgers Equation Problem

For the Burgers problem, we will again use a single set of final end time observations and a poor initial guess of $g(x) = 0.5$, with the fixed parameters N=96, dt=0.005, beta = 1.0. To demonstrate the effects of nonlinear features on the solution, we have solved this problem for various end times. The first figure 2.11 shows that the IC is recovered to within a pretty close range of the exact IC used, but only after a very short time. As time progresses, however, one can see that the recovered initial condition becomes progressively worse to the point of showing little resemblance to the correct solution as in figure 2.19. To remedy this, additional time observations must be provided. Note that increasing the number of observations in time directly leads to improved solution quality for nonlinear

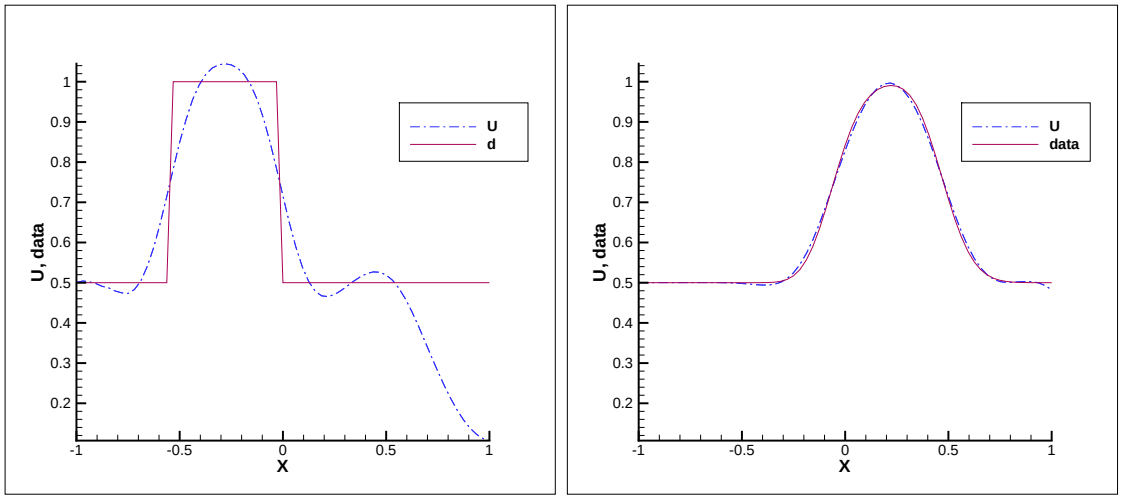

Figure 2.3: Adjoint solution for linear advection problem. Left: Recovered IC for 64 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{32}$, $\Delta t = 0.01$. Requires 200 iterations for convergence and runs in 1.945 seconds on a single CPU core.

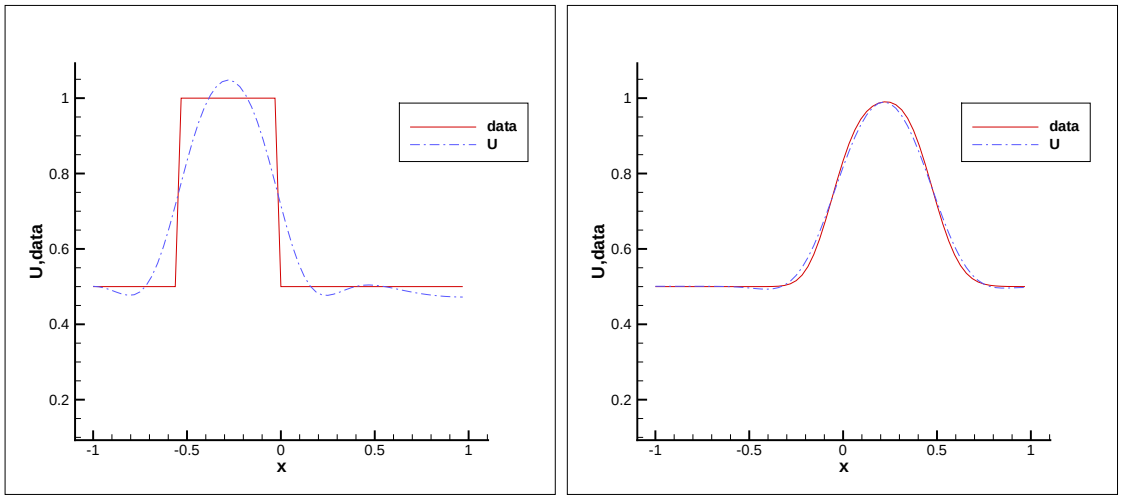

Figure 2.4: MDS solution for linear advection problem. Left: Recovered IC for 64 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{32}$, $\Delta t = 0.01$. Requires 8470 iterations for convergence and runs in 50.183 seconds on four CPU cores.

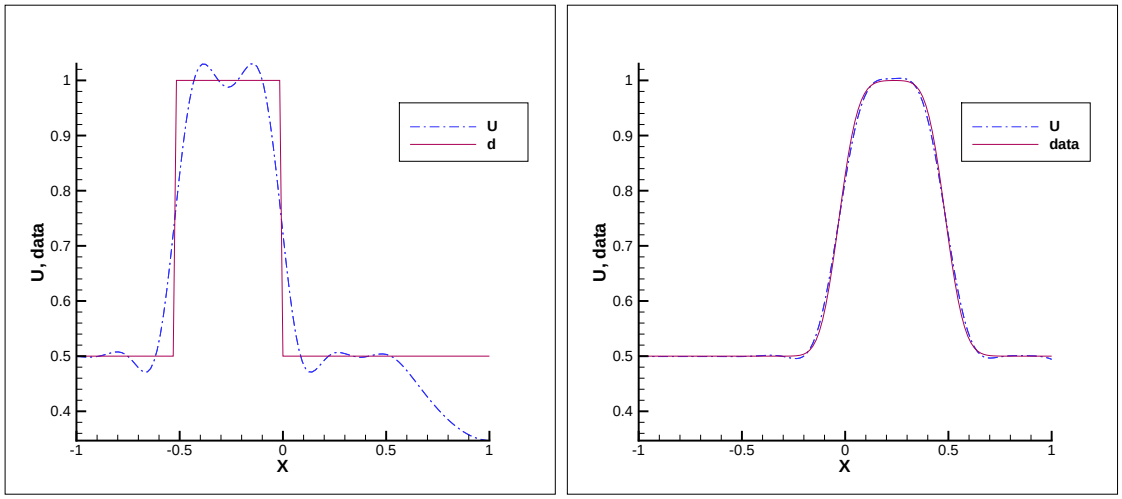

Figure 2.5: Adjoint solution for linear advection problem. Left: Recovered IC for 128 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{64}$, $\Delta t = 0.01$. Requires 700 iterations for convergence and runs in 15.660 seconds on a single CPU core.

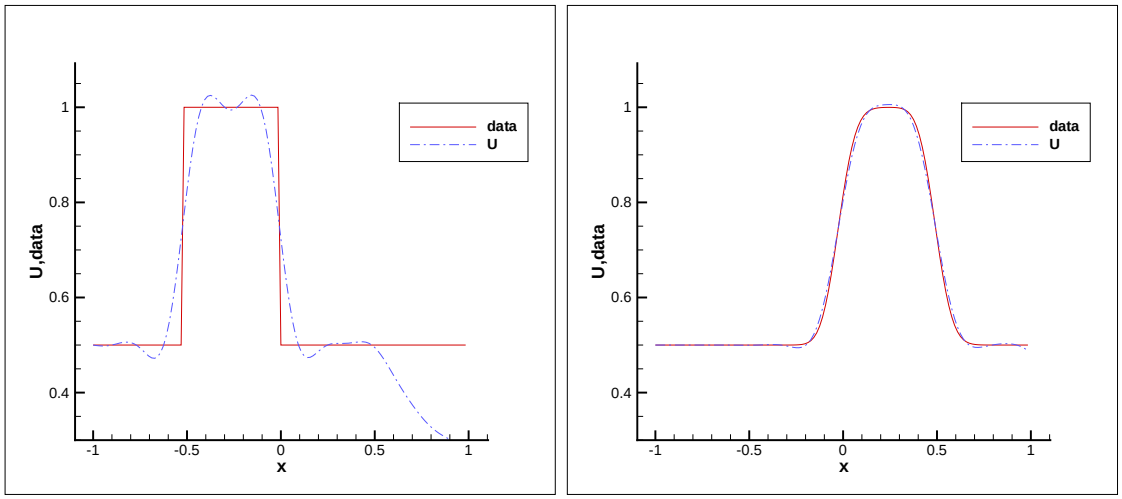

Figure 2.6: MDS solution for linear advection problem. Left: Recovered IC for 128 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{64}$, $\Delta t = 0.01$. Requires 25330 iterations for convergence and runs in 553.261 seconds on four CPU cores.

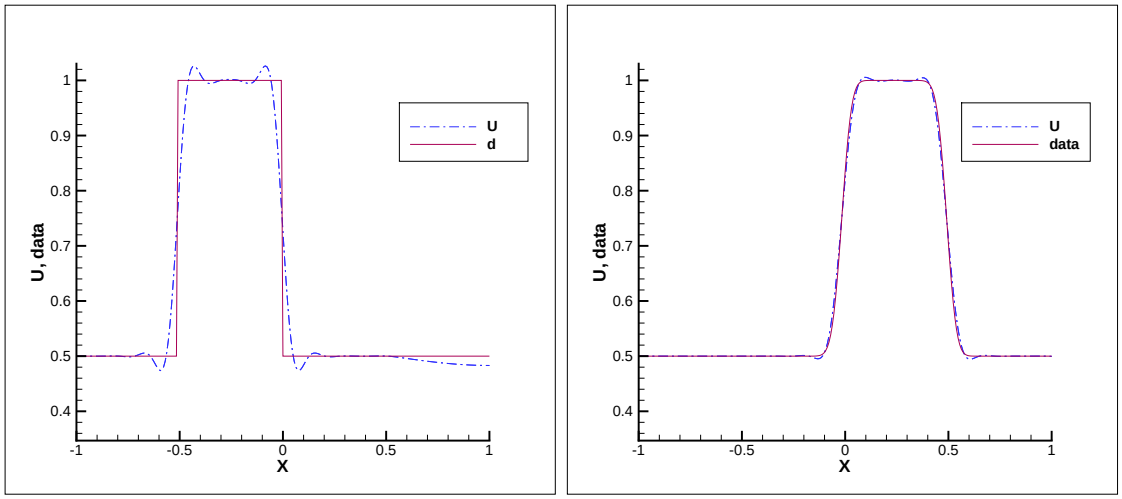

Figure 2.7: Adjoint solution for linear advection problem. Left: Recovered IC for 256 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{128}$, $\Delta t = 0.01$. Requires 5700 iterations for convergence and runs in 476.535 seconds on a single CPU core.

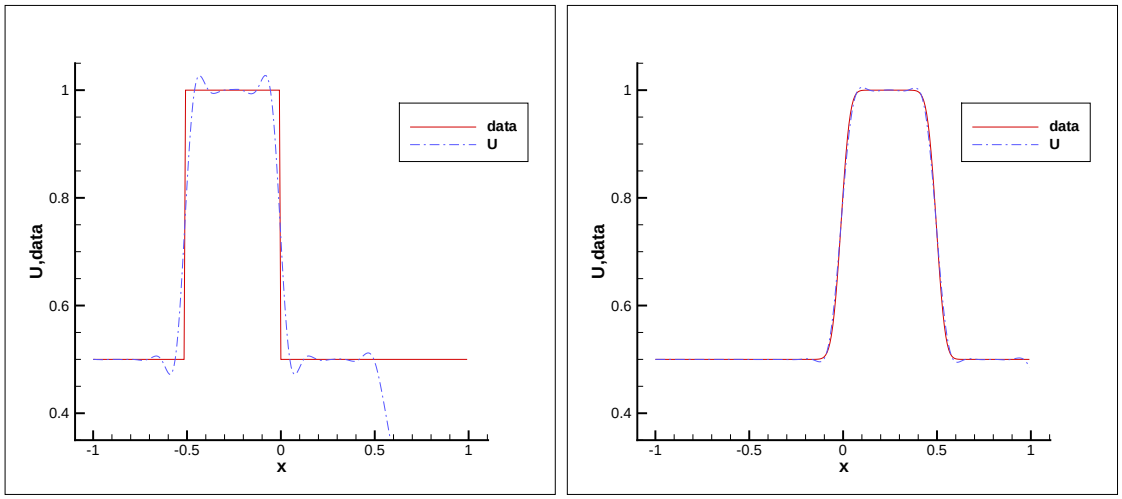

Figure 2.8: MDS solution for linear advection problem. Left: Recovered IC for 256 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{128}$, $\Delta t = 0.01$. Requires 28060 iterations for convergence and runs in 2394.518 seconds on four CPU cores.

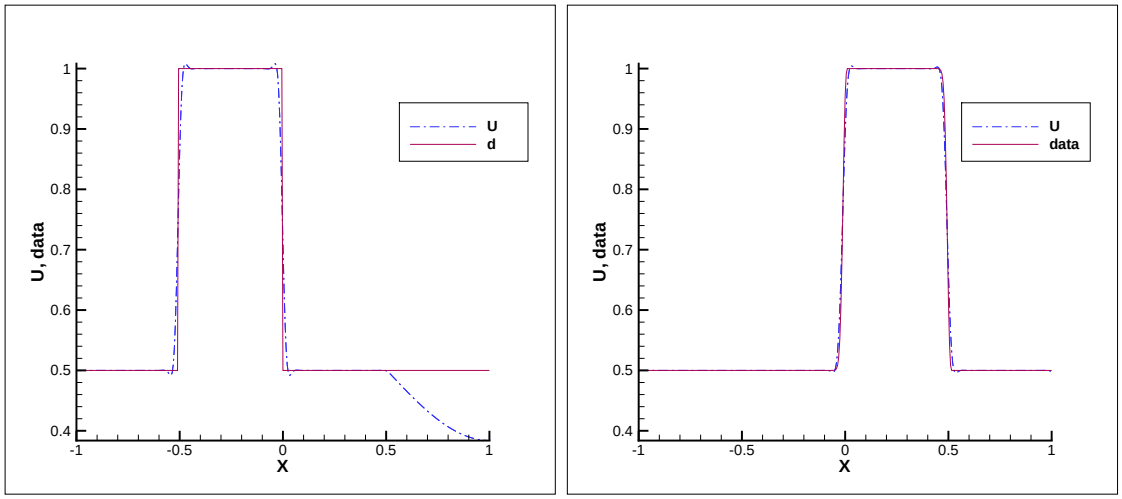

Figure 2.9: Adjoint solution for linear advection problem. Left: Recovered IC for 400 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$. Requires 200 iterations for convergence and runs in 57.158 seconds on a single CPU core.

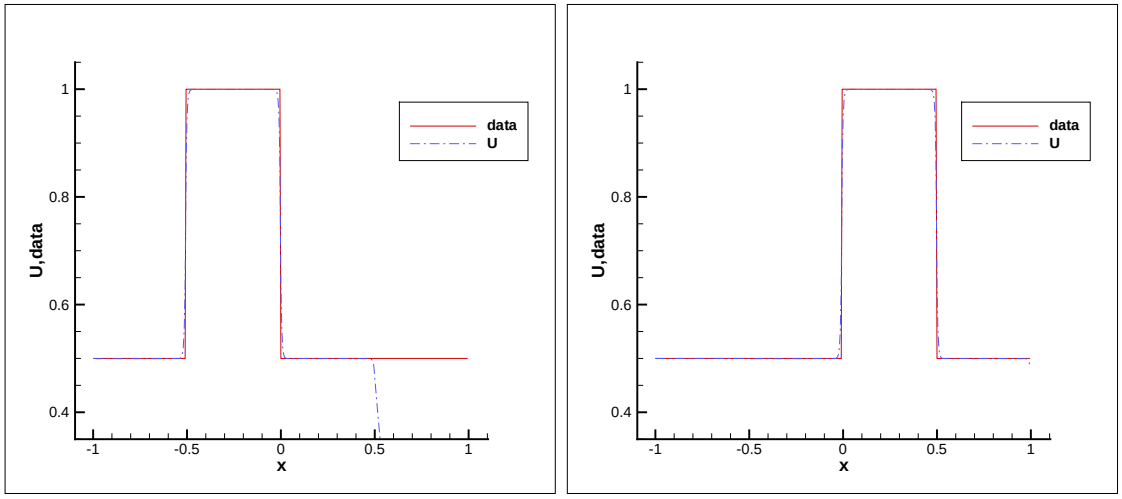

Figure 2.10: MDS solution for linear advection problem. Left: Recovered IC for 400 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$. Requires 20690 iterations for convergence and runs in 4694.7679 seconds on four CPU cores.

problems. This is important, as it is known that increasing the number of observational data points for a single time does not have a major impact [26], but here we see that increasing the number of time observations indeed does.

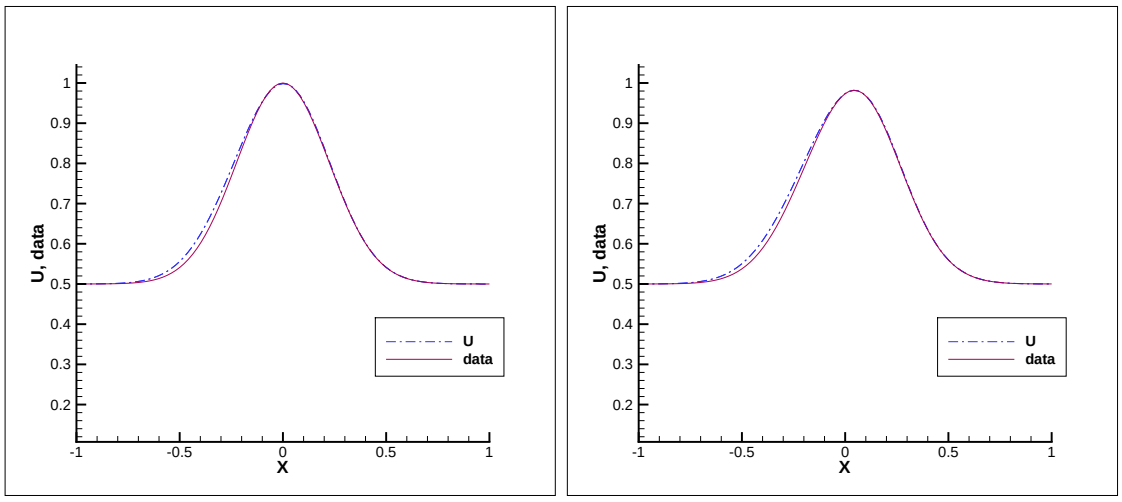

Figure 2.11: Adjoint solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.05. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 200 iterations for convergence and runs in 4.339 seconds on a single CPU core.

2.3.3 Multigrid-Multidirectional Search Strategies

Employing the multigriding strategies of algorithms 5 and 6 have proven effective at substantially reducing the number of iterations required for convergence of both the linear advection and the nonlinear Burger's data assimilation problems in many scenarios. For the linear advection problem, we have found that both the v-cycle and FMG cycles produce near optimal results by using approximately 300 fixed iterations of the MDS solver at each level. While we have seen good speedup by employing multigriding strategies, one can see from the figures that the quality of the converged solution does suffer slightly, and we attribute this to the lack of a correction step in our algorithm. One method we have employed to try and smooth out these small errors that eminate from the use of a multigriding strategy is to use a weighted average of the fine grid solutions. To accomplish this, we take a weighted average of the solution everytime we arrive at a fine grid in the MG cycle, i.e. $U = \theta U + (1-\theta)U_{old}$ for a $\theta \in (0, 1)$. This has shown to be useful in smoothing out the solution of the linear advection problem when more than 1 FMG cycle is used as seen in figure 2.25, but at the cost of added sppedup as seen in table 2.3.

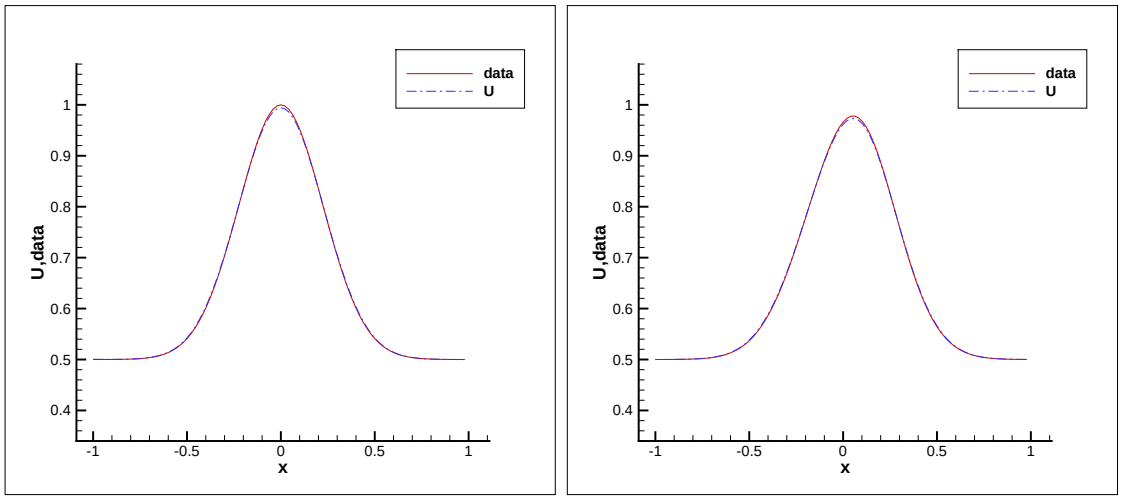

Figure 2.12: MDS solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.05. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 720 iterations for convergence and runs in 2.763 seconds on four CPU cores.

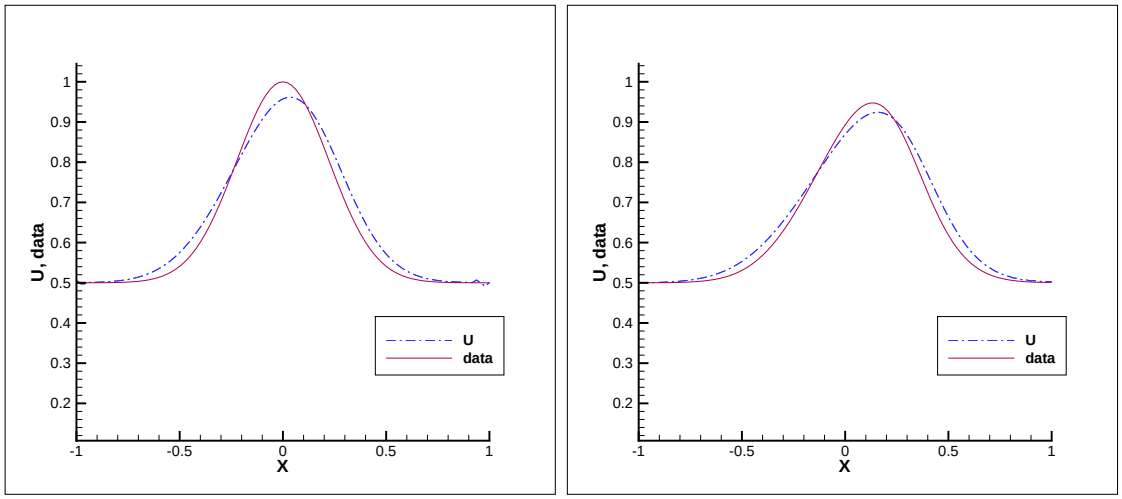

Figure 2.13: Adjoint solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.15. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 200 iterations for convergence and runs in 27.599 seconds on a single CPU core.

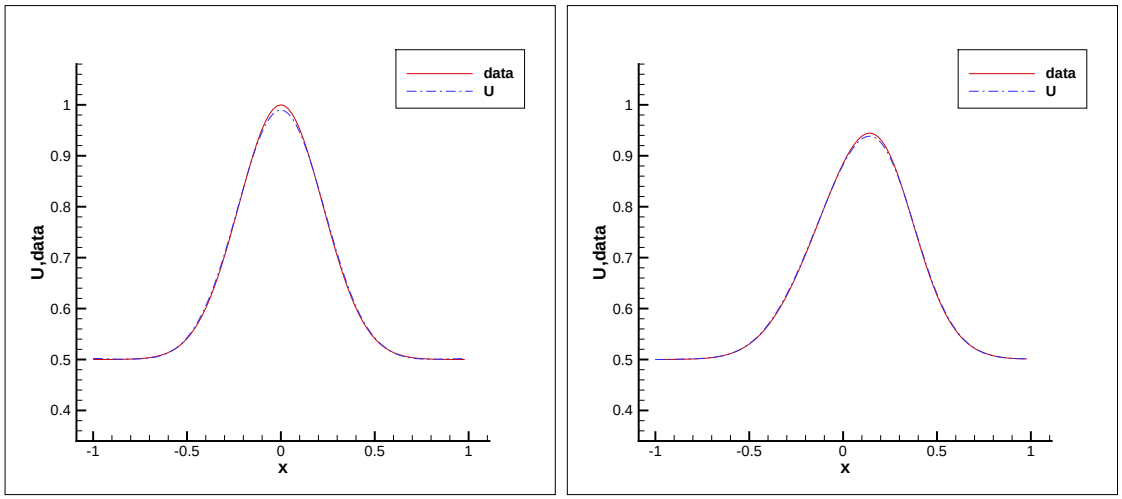

Figure 2.14: MDS solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.15. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 1470 iterations for convergence and runs in 10.354 seconds on four CPU cores.

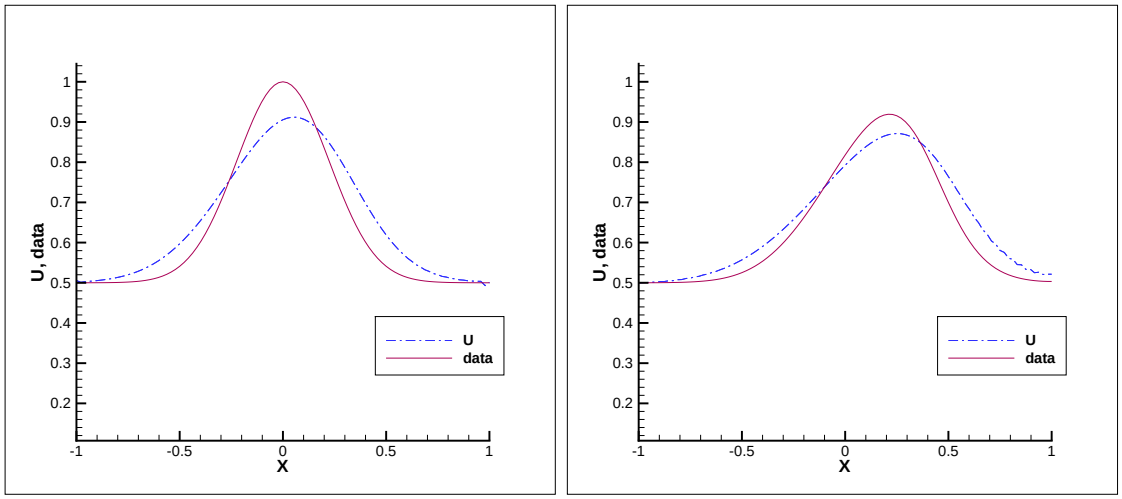

Figure 2.15: Adjoint solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.25. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 200 iterations for convergence and runs in 71.018 seconds on a single CPU core.

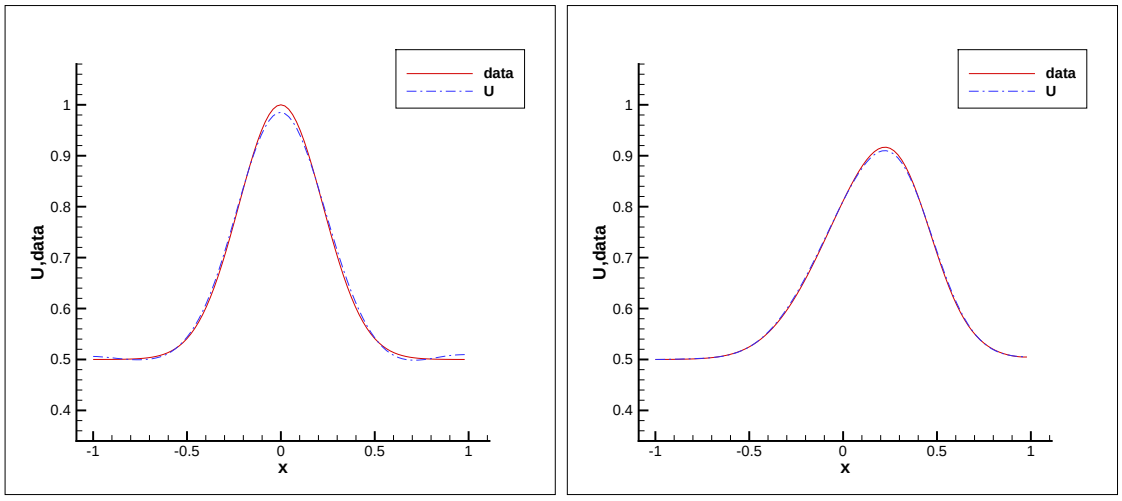

Figure 2.16: MDS solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.25. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 3220 iterations for convergence and runs in 30.021 seconds on four CPU cores.

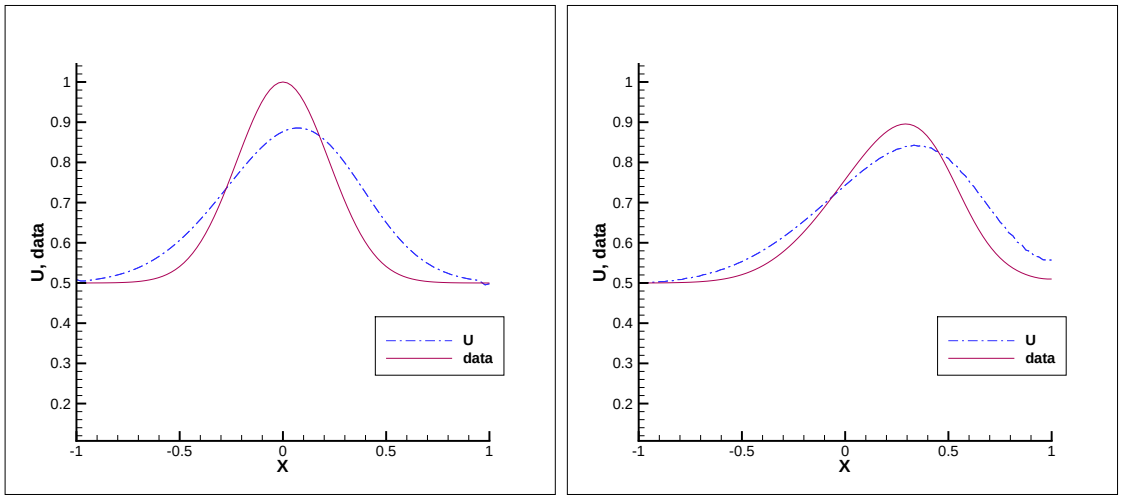

Figure 2.17: Adjoint solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.35. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 200 iterations for convergence and runs in 134.777 seconds on a single CPU core.

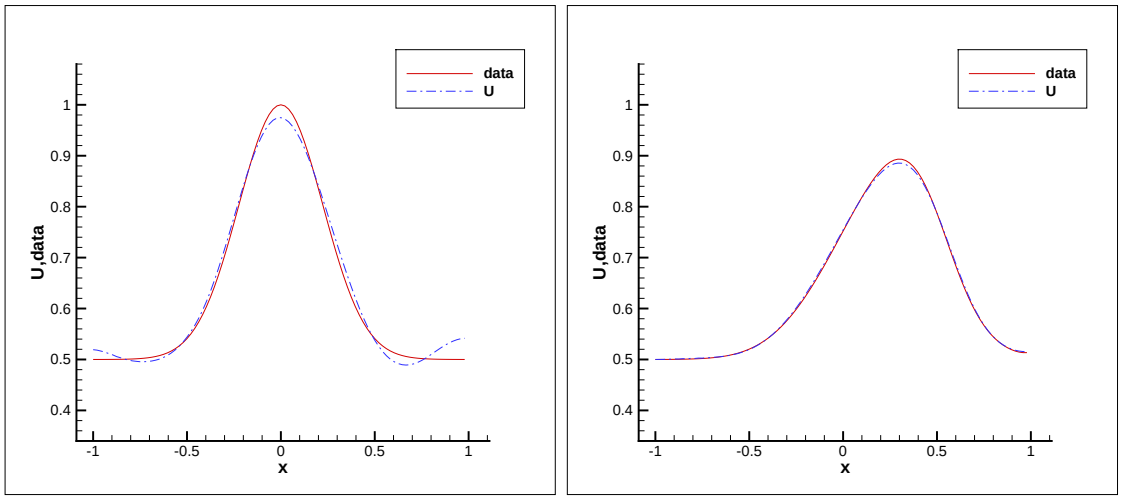

Figure 2.18: MDS solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.35. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 10690 iterations for convergence and runs in 120.272 seconds on four CPU cores.

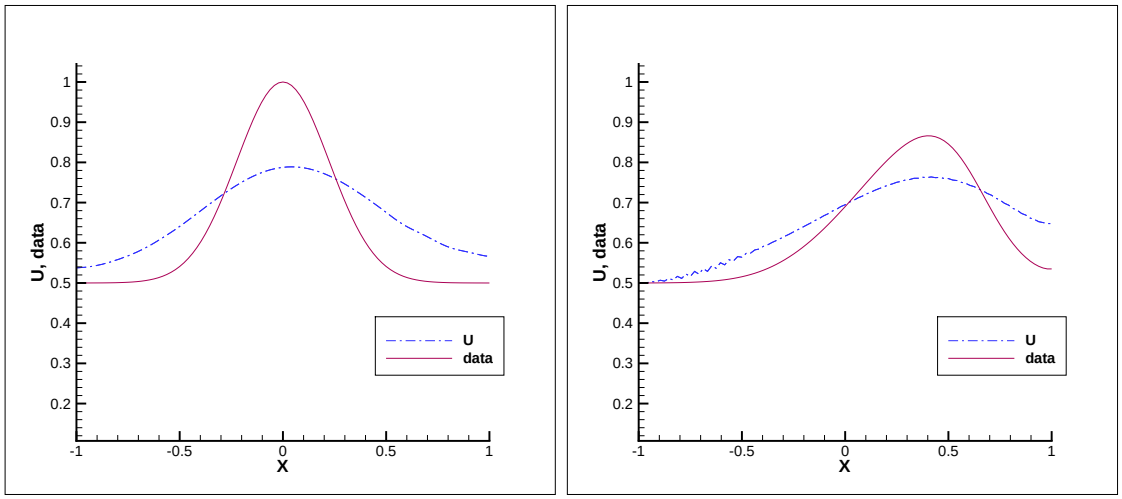

Figure 2.19: Adjoint solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.5. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 300 iterations for convergence and runs in 404.615 seconds on a single CPU core.

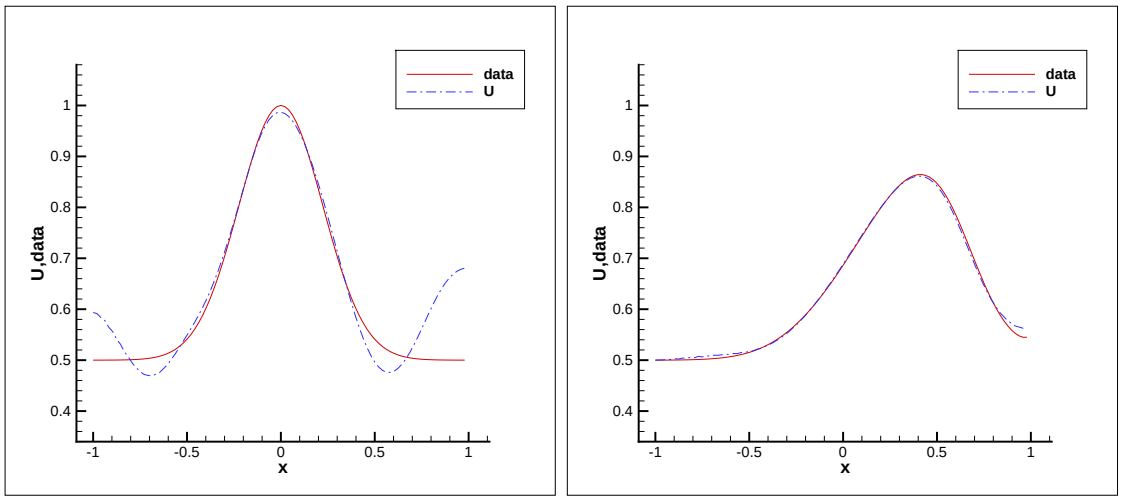

Figure 2.20: MDS solution for non-linear Burger's equation problem. Left: Recovered IC (U) versus exact IC (data), t=0.5. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$. Requires 45620 iterations for convergence and runs in 704.615 seconds on four CPU cores.

Table 2.1: V-cycle multigriding strategy applied to the multidirectional search method on the 1D linear advection data assimiliation problem for N=400 grid points. V-cycles utilize 300 iterations at each grid level, and the final number of MDS iterations required for the solver to converge are also listed. Time represents user time on four processor cores. Baseline time for the original MDS algorithm is 17977 seconds.

V-Cycles (maxcycles)	Final MDS Iterations (finaliterates)	Time (s)	Speedup
1	24740	21968	0.82X
2	10350	10586	1.70X
3	8360	9554	1.88X
4	10100	11691	1.54X

Table 2.2: FMG multigriding strategy applied to the multidirectional search method on the 1D linear advection data assimiliation problem for N=400 grid points. The FMG cycle utilizes between 200 and 400 iterations at each grid level, the final number of MDS iterations required for the solver to converge are also listed. Time represents user time on four processor cores. Baseline time for the original MDS algorithm is 17977 seconds.

maxiterates	Final MDS Iterations (finaliterates)	Time (s)	Speedup
200	9280	8911	2.02X
250	7040	7179	2.50X
300	2260	3279	5.48X
350	5780	6381	2.82X

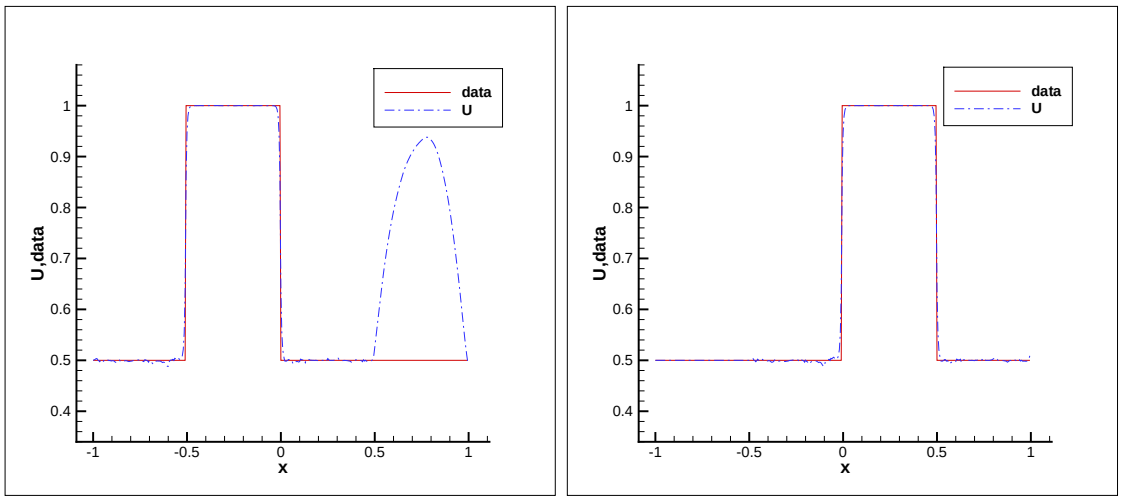

Figure 2.21: MGMDS solution for linear advection problem using an FMG cycle with 200 iterations at each level. Left: Recovered IC for 400 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$.

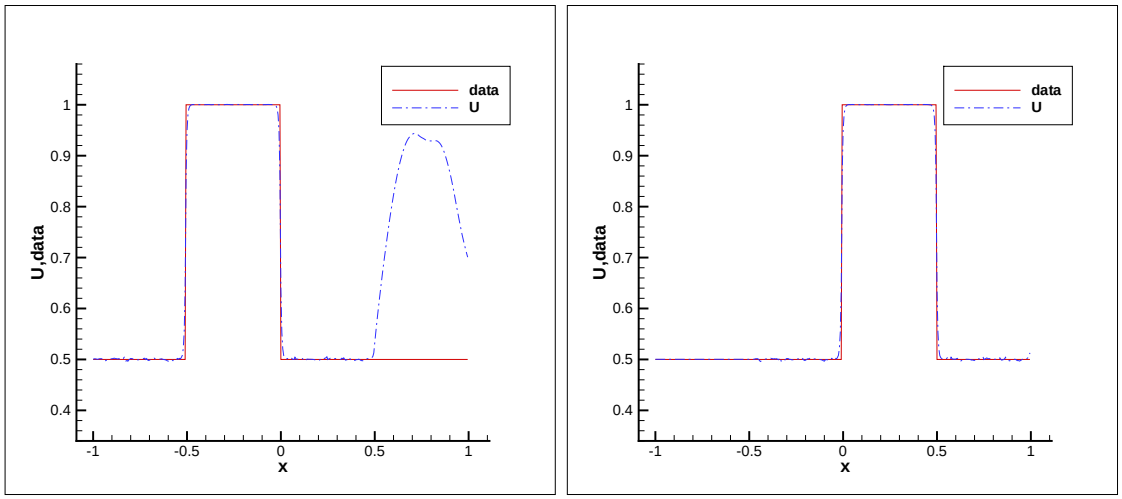

Figure 2.22: MGMDS solution for linear advection problem using an FMG cycle with 300 iterations at each level. Left: Recovered IC for 400 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$.

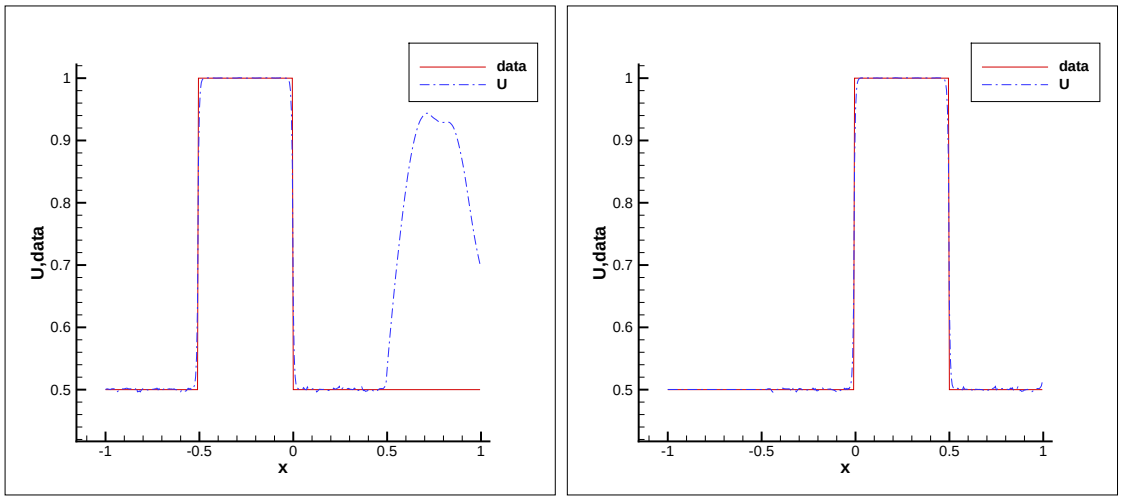

Figure 2.23: MGMDS solution for linear advection problem using 2 V-cycles with 300 iterations at each level. Left: Recovered IC for 400 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$. Requires 2 V-cycles plus 10350 final MDS iterations for convergence and runs in 2515.487 seconds on four CPU cores.

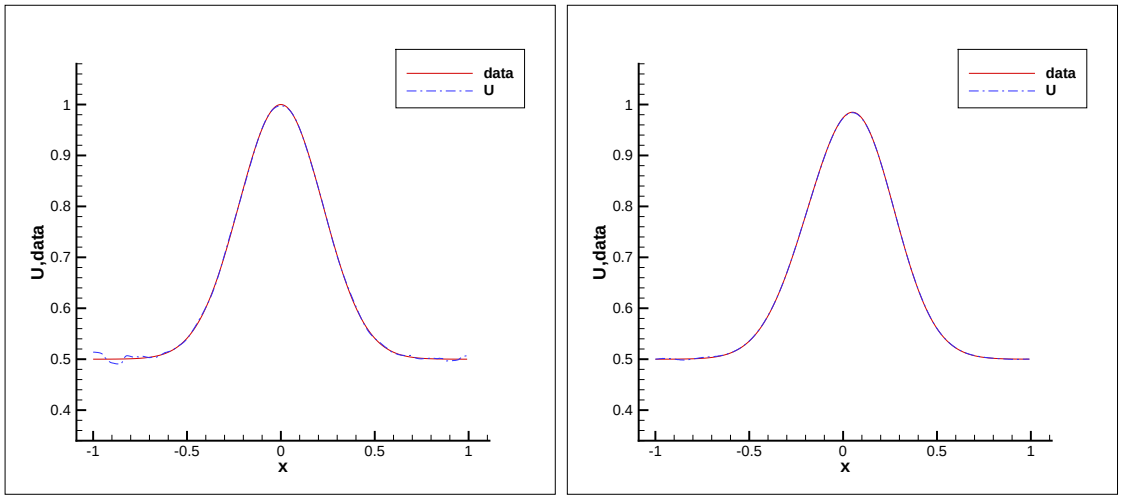

Figure 2.24: MGMDS solution for nonlinear Burger's problem using an FMG cycle. Left: Recovered IC for 256 grid points (U) versus exact IC (data). Right: Comparison of the advected solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$.

Table 2.3: FMG multigriding strategy applied to the multidirectional search method on the 1D linear advection data assimiliation problem for N=400 grid points, and employing a weighted averaging of the fine grid solutions, $\theta = 0.5$. The FMG cycle utilizes 300 iterations at each grid level, the final number of MDS iterations required for the solver to converge are also listed. Time represents user time on four processor cores. Baseline time for the original MDS algorithm is 17977 seconds.

FMG Cycles	Final MDS Iterations	Time (s)	Speedup
1	4350	4863	3.70X
2	4460	5760	3.12X
3	5370	7308	2.46X
4	6630	9152	1.96X

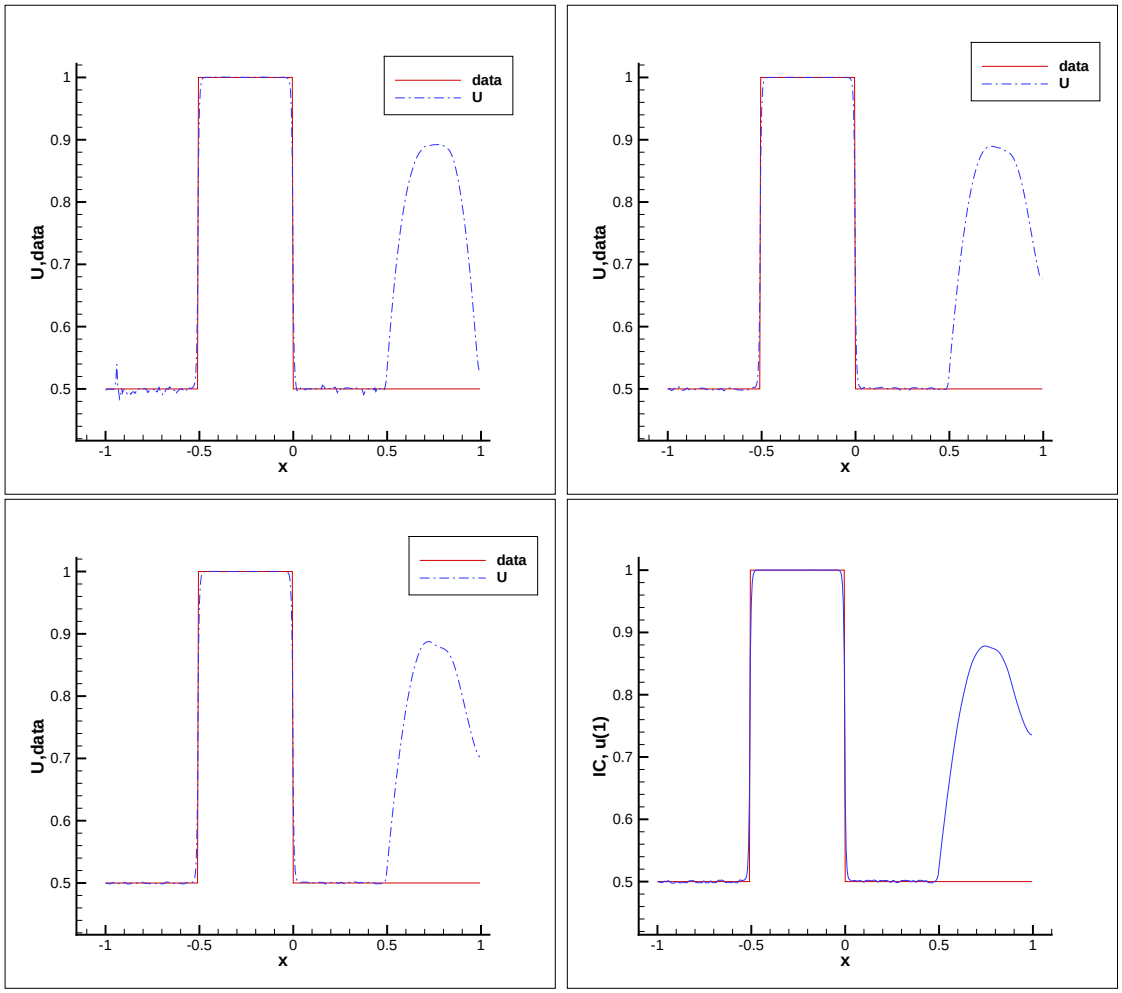

Figure 2.25: MGMDS IC recovery for a 1D linear advection data assimilation problem using an FMG cycle employing a weighted averaging of the fine grid solution. Using more than one FMG cycle results in a smoothing effect on the solution data. Top Left: 1 FMG cycle. Top Right: 2 FMG cycles. Bottom Left: 3 FMG cycles. Bottom Right: 4 FMG cycles. $\beta = 1.0$, $\Delta x = \frac{1}{200}$, $\Delta t = 0.01$ for all figures.

2.3.4 Discussion of Numerical Results

We should note that the numerical experiments carried out here were under conditions far from ideal. Usually, a good starting estimate along with more available data (i.e. data collected at multiple time intervals) is used for the problem solution in data assimilation problems, but here we have purposely used poor initial guesses along with limited data. It can be seen that for linear problems, the adjoint method works very well and is able to recover a difficult initial condition possessing steep gradients with ease using only a single set of observed solutions, and that as the computational grid is resolved the error tends towards zero. The MDS algorithm also works well for the linear advection problem, producing qualitative solutions very similar to those of the adjoint method despite the large number of design variables.

The adjoint method does not work as well on the nonlinear Burgers problem, as poor initial guesses will produce poorer and poorer solutions as the solution is carried out further and further in time. This is due to the nonlinear features, and in this scenario the use of data for multiple times should be used to correct this problem. The figures provided (2.26, 2.27, 2.28) demonstrate how increasing the number of observations leads to improved solutions for nonlinear problems. We note that many observations are typically used in data assimilation problems and that errors resulting from non-linear model features are well known, though those produced by data assimilation tend to be lower than those produced by other predictive modeling methods, see for example [6]. The MDS method works very well on the nonlinear Burger's problem even in the absence of additional observational data as can be seen by figures 2.11 - 2.20.

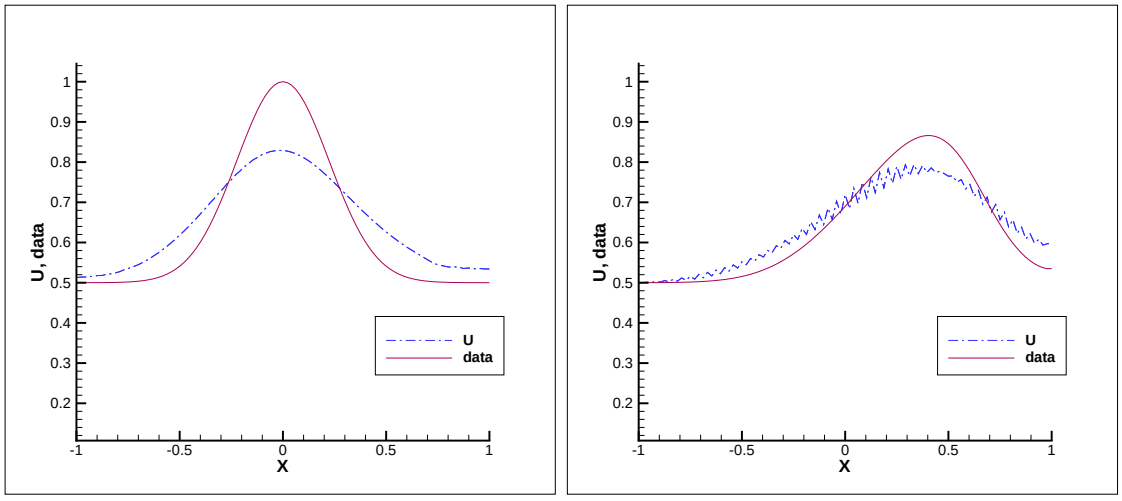

Figure 2.26: Adjoint nonlinear IC Recovery, 5 observations. Left: Recovered IC (U) versus exact IC (data), t=0.5. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$.

We would like to compare how well the multidirectional search with and without a

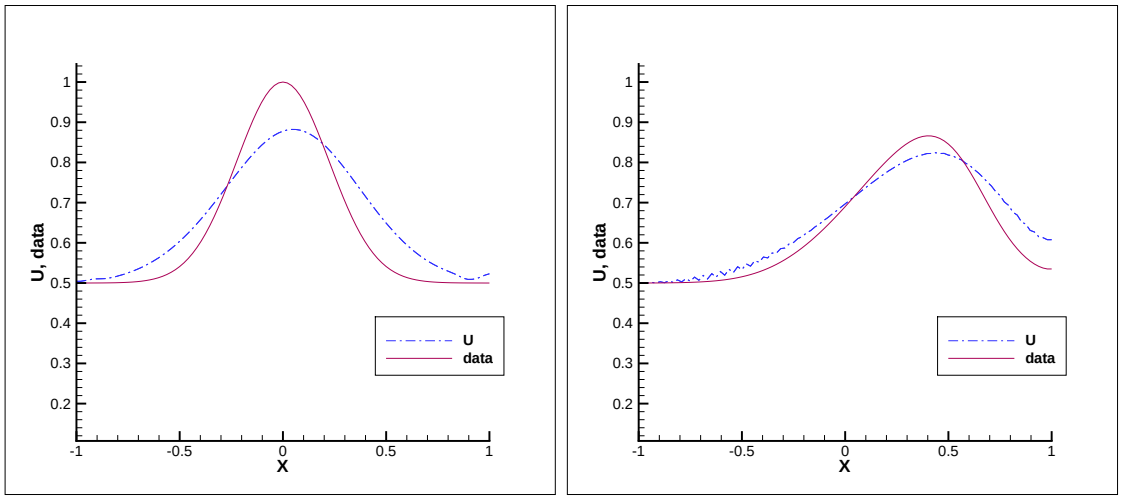

Figure 2.27: Adjoint nonlinear IC Recovery, 10 observations. Left: Recovered IC (U) versus exact IC (data), t=0.5. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$.

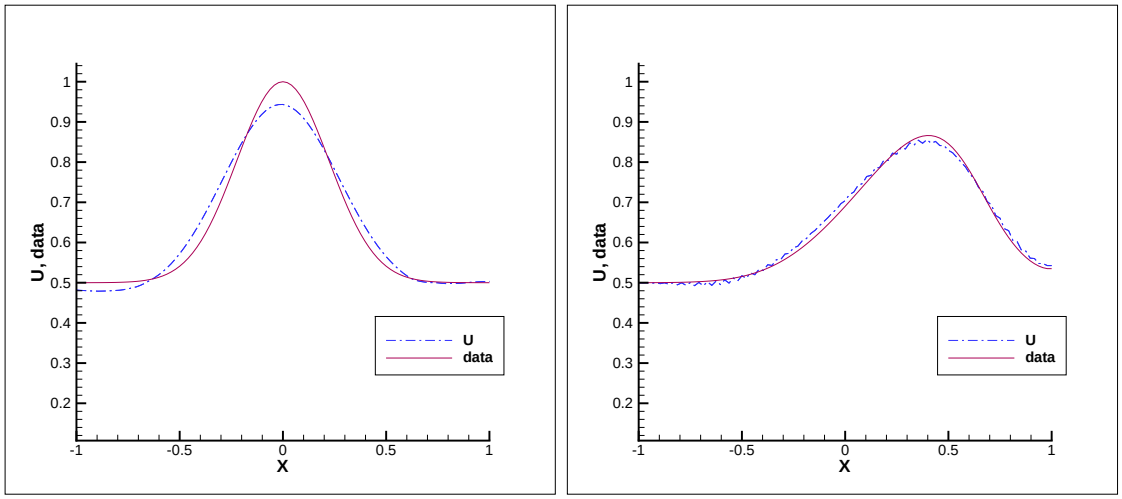

Figure 2.28: Adjoint nonlinear IC Recovery, 20 observations. Left: Recovered IC (U) versus exact IC (data), t=0.5. Right: Comparison of the final solutions. $\beta = 1.0$, $\Delta x = \frac{1}{48}$, $\Delta t = 0.005$, $\mu = 0.001$.

multigriding strategy compares to the adjoint method for the numerical examples given. While it is very clear that the discrete adjoint method is substantially faster than MDS in a sequential setting, we wish to show that in the presence of abundant processors the MDS method can compete on par with the discrete adjoint method on a single core. For this purpose we will compare the linear advection problem with N=400 grid points since it produced the most accurate results for all methods involved here. For this problem, the discrete adjoint code converged in 200 iterations and took 57.158 seconds to complete. In contrast, it took the MDS method 20690 iterations to converge and 16970 seconds on a single core. In parallel, the same MDS code ran in 4694 seconds, giving a parallel efficiency of about 0.9 on a shared memory system. We note that the parallel efficiency was higher in all other cases, and almost linear for the N=256 case, so we feel that 0.9 is a safe estimate to use for this hypothetical exercise. The MDS method can be considered 'embarassingly parallel' on up to N+1 processors since this is how many parallel flow solves can be done at once. So, if we were to theoretically have 401 processors on a shared memory system, then this code would take approximately 47 seconds (=16970.342/(401*0.9)) to run (a little faster than the adjoint code), and using this parallel efficiency we would require approximately 330 cores to match the speed of the adjoint code. The MGMDS code is much more promising. Consider using the FMG cycle with 300 iterations at each grid level, then the estimated parallel efficiency is 0.94 and takes 3090 seconds on a single core. In this scenario we would only need about 58 cores to match the speed of the adjoint method, and with 401 cores would be 7X faster than the adjoint method. We note that our discrete adjoint code could also benefit somewhat from parallelization, but the underlying algorithm is inherently sequential and the parallel efficiency would likely be very low.

CHAPTER 3

CFD CODE ACCELERATION

In this chapter we will discuss some approaches that have been taken in order to accelerate two CFD codes, the NASA FUN3D code and the coupled level-set and volume-of-fluid (CLSVOF) code for incompressible multiphase flows. While we have primarily focused on the use of add on GPU hardware for accelerating CFD codes, we will also discuss an improved pressure projection algorithm for adaptive mesh refinement (AMR) that we have recently developed. This new algorithm based on Tatebe's multigrid preconditioned conjugate gradient method (MGPCG) [76] substantially reduces the computational effort required by the projection step when compared to MG AMR solvers, and also improves the scalability of the CLSVOF code by avoiding a 'domain decomposition penalty'.

3.1 Graphics Processors and Hybrid Computing Architectures

Modern graphics processors along with languages such as the breakthrough 'Compute Unified Device Architecture' (CUDA) [14] and OpenCL to program them has allowed for scientific computations on distributed computers to benefit from an additional level of fine grain parallelism at the node level. When attempting to develop an accelerated version of a large parallel CFD code, several challenges present themselves. To start with, these codes have already been highly optimized for parallel computation and so the biggest possible gains may have already been accounted for. One must look beyond coarse grain parallelism (e.g., dividing a grid into blocks and distributing them over multiple CPU cores) and seek fine grained data parallel tasks which can be ported to the GPU. If these opportunities exist for a large scale code, it has the potential to benefit from an added level of parallelism. Parallel codes also must employ frequent MPI communications, which when used in an accelerator environment will implicitly invoke expensive GPU-CPU data transfers. Scalability is also a big issue for all production level CFD codes. With the rising number of CPU cores per processor coupled with the rising cost and power consumption of an individual GPU, it is not likely that future large scale hybrid architectures will be able to maintain a 1:1 or better ratio of GPUs to CPU cores. The new availability of concurrent kernel execution allows multiple cores to share a single GPU. This feature should allow hybrid codes to scale on systems with a GPU to CPU ratio below 1. This feature does come at a price though, as increasing the number of CPU threads which simultaneously share a single GPU puts added

demand on the available memory and could put additional strains on other resources for large kernels. In addition to these challenges, the unstructured solver of FUN3D is hindered by the unavoidable constraint of out-of-order memory access patterns, which can cause very poor performance on a GPU. In this chapter we will describe the CLSVOF and FUN3D codes, and detail the various strategies we have employed to accelerate them.

3.2 The Coupled Level-Set and Volume-of-Fluid Method

We will consider the following incompressible Navier-Stokes equations for multiphase flow [73]:

$$\rho(\phi)(\vec{u}_t + \vec{u} \cdot \nabla \vec{u}) = -\nabla p + \nabla \cdot \mu(\phi) \nabla D - \gamma \kappa(\phi) \nabla H(\phi) + \rho \vec{g}.$$

$$\nabla \cdot \vec{u} = 0.$$

where $\rho(\phi)$ is the density, $\mu(\phi)$ is the viscosity, $\vec{u}$ is the velocity, D is the rate of deformation tensor, p is the pressure, $\kappa(\phi)$ is the curvature, $\vec{g}$ is the gravitational forces, and $H(\phi)$ is the Heaviside function.

In the level set approach, the set of points,

$$\Gamma = \{(x, y, z) | \phi(x, y, z, t) = 0\},$$

represents the air/water interface Γ at time t. For single-valued surfaces the points satisfying $z = \zeta(x, y, t)$ represents the zero level set of $\phi(x, y, z, t)$. The advantage of using $\phi(x, y, z, t)$ is that multi-valued surfaces are handled naturally. Also, the level set representation of an interface does not break down if the interface changes topology: e.g. if an interface break-up occurs or if surfaces reconnect. The equation governing the level set function is given by,

$$\phi_t + \vec{u} \cdot \nabla \phi = 0.$$

In order to increase the accuracy for simulating multi-phase flows, we have developed adaptive mesh refinement (AMR) level set methodology in order to dynamically place grid points in the vicinity of the air/water interface [18, 70–72].

The key component of this work considers the solution of the pressure projection equation,

$$\nabla \cdot \frac{1}{\rho} \nabla p = \nabla \cdot \vec{u}^*, \tag{3.1}$$

where $\vec{u}^*$ is an intermediate velocity field resulting from a standard "projection method" splitting scheme, and the density ρ is represented by both liquid and gas phases

$$\rho = \rho^L H(\theta) + \rho^G (1 - H(\theta)).$$

The discretization utilizes the level set function such that ρ is a function of ϕ and

$$\rho_{i+1/2,j}(\phi) = \rho^L \theta_{i+1/2,j} + \rho^G (1 - \theta_{i+1/2,j})$$

where $\theta_{i+1/2,j}$ is a height fraction,

$$\theta_{i+1/2,j} = \frac{\phi^+_{i,j} + \phi^+_{i,j+1}}{|\phi_{i,j}| + |\phi_{i,j+1}|}$$

and $\phi^+ = max(\phi, 0)$. The spatial discretization of eqn. 3.1 utilizes cell centered values for p with corresponding values at the cell faces for ρ such as in the example 1D discretization 3.3. We note that the use of first order boundary conditions results in a symmetric discretization matrix [53, 54]. We will primarily be concerned with solving equation 3.1.

3.2.1 Solution of the Pressure Poisson Equation

The numerical simulation of incompressible fluid flows consisting of multiple phases with large density variations (such as in liquid-gas scenarios) can present significant challenges, particularly in the solution of the pressure projection step. The phase density ρ is represented in liquid and gas phases by $\rho = \rho_L H(\phi) + \rho_G(1 - H(\phi))$, where $H(\phi)$ is a Heaviside function equal to 1 in liquid and 0 in gas. The rate of convergence of the iterative solution of the pressure Poisson equation

$$\nabla \cdot \frac{1}{\rho} \nabla p = F \tag{3.2}$$

is retarded if $\frac{\rho_L}{\rho_G} >> 1$ because the resulting discretization matrix is very poorly conditioned. Consider the 1D discretization of eqn. 3.2

$$\frac{\frac{1}{\rho_{i+\frac{1}{2}}} \frac{p_{i+1} - p_i}{\Delta x} - \frac{1}{\rho_{i-\frac{1}{2}}} \frac{p_i - p_{i-1}}{\Delta x}}{\Delta x} = F_i, \tag{3.3}$$

where $\rho_{i+\frac{1}{2}} = 1$ in a first phase and α in a second phase, then the condition number of the corresponding discretization matrix becomes larger as the density ratio increases as seen in table 3.1. Though we have seen that the condition number is sensitive to the density ratio, we have found that the condition number is not necessarily sensitive to the problem geometry as seen in figure 3.2, where all illustrated geometries produce a discretization matrix whose condition number is of the same order of magnitude.

Significant steps in reducing the solution time of the pressure projection step for these problems have come from separate sources. The development of adaptive mesh refinement [7, 8, 34, 72] (AMR) has allowed for a reduction in the number of required computational grid points when used in conjuction with a method such as multigrid (MG). Unstructured grid coarsening strategies have been implemented to allow for MG solutions of the Poisson

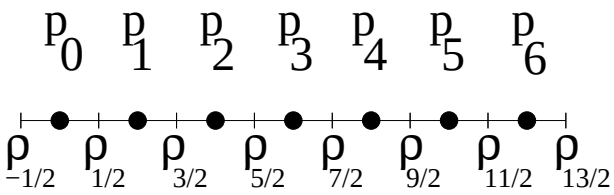

Figure 3.1: Example 1D discretization for the pressure Poisson equation

Table 3.1: In multiphase flows, the condition number of the discretization matrix for eqn. 3.2 grows with the density ratio. In this table, the condition number is calculated for the discretization matrix of a 1D two phase flow in the domain $[0, 1]$ following eqn. 3.3, and with the phase interface occurring at $x = 0.25$. The example flow has a density of 1 in the first phase on the interval $[0, 0.25)$ and α in a second phase on the interval $(0.25, 1]$. Values represent a discretization with 256 grid points and were calculated in MATLAB using the built in condition number function cond().

α	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Density Ratio	1	10	10^2	10^3	10^4	10^5
Condition #	205	1.2 x 10^3	1.2 x 10^4	1.2 x 10^5	1.2 x 10^6	1.2 x 10^7

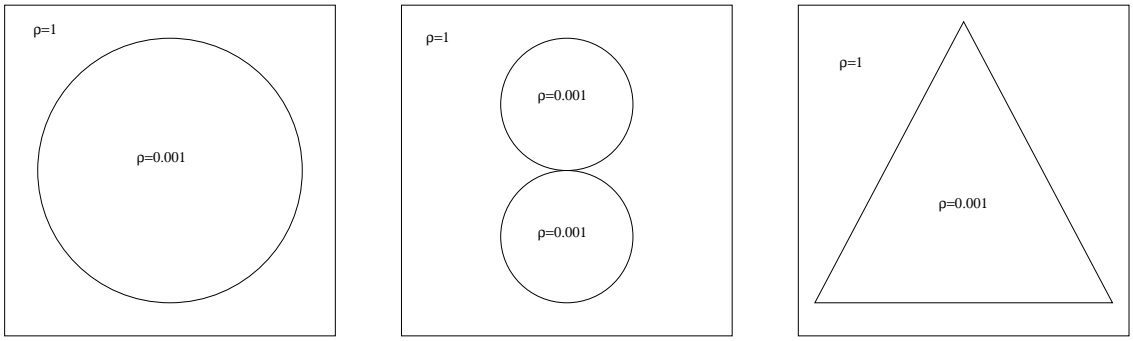

Figure 3.2: The condition number of the discretization matrix is not as sensitive to the problem geometry as it is to the density ratio. The corresponding condition numbers for these figures are 6,132,300 (left), 1,861,000 (middle) and 2,548,900 (right) using a 2D version of discretization eqn. 3.3 on a 64 x 64 grid.

equation in unstructured applications [81]. Alternatively, on uniform grids, MG has been used to form the preconditioning matrix for the preconditioned conjugate gradient (PCG) method. The multigrid preconditioned conjugate gradient method (MGPCG) was first introduced by Tatebe [76] who showed that poorly conditioned problems could be solved 12 times faster using MGPCG instead of MG, and 5 times faster than incomplete LU preconditioned CG (ILU-PCG). In our previous work [71] we developed a pressure solver on an adaptive hierarchy of grids using a combination of MG and MGPCG. The MGPCG "smoother" was applied a single level at a time. We have discovered though, that the performance of the solver developed in [71] is sensitive to the number of smoothing steps of MGPCG. In [71] a fixed number of pre-smoothing and post-smoothing MGPCG iterations were done, but we have found that the convergence rate of the overall MG AMR method was sensitive to the number of smoothing steps. If the number of smoothing steps increases the cpu time increases, but if the number of smoothing steps decreases, the method may not converge at all. In [70] we had modified the algorithm in [71] by taking a number of MGPCG steps necessary in order to insure that the residual on a given level met a prescribed tolerance. The approach in [70] is more robust, but less efficient. Moreover, the performance of the method in [70, 71] is sensitive to the blocking factor.

3.2.2 GPU Acceleration of the Pressure Projection Step on a Uniform Grid

An early attempt at accelerating the pressure projection step on a uniform grid was carried out using the PGI Fortran compiler with accelerator directives. The key to this task is accelerating the smoothing steps used by the algorithm, e.g. Jacobi, Gauss-Siedel Red Black (GSRB) or incomplete LU (ILU). If the method is already parallel, such as with a Jacobi smoother or GSRB, the PGI fortran accelerator makes it easy for one to run on a GPU with little effort since there is minimal code porting. The PGI accelerator makes use of compiler directives on a comment line and so they remain hidden if the acclerator option is not employed, much like OpenMP directives. Algorithm 7 demonstrates a simple example of using this method with a Jacobi smoother for a 1D problem.

Algorithm 7 PGI FORTRAN Accelerator Code for a GPU Accelerated Jacobi Smoother

```
!$acc region
 DO iterates = 1,maxiterates
   DO i = istart,iend
    x(i)=(b(i)-L(i-1)*x_old(i-1)-U(i+1)*x_old(i+1))/D(i))
   END DO
   DO i = istart,iend
    x_old(i)=x(i)
   END DO
 END DO
!$acc  end  region
```

While the Jacobi smoother is easy to implement and works well on a GPU, we would like a better performing solver like that of ILU, but that can be run in parallel on a GPU.

Algorithm 8 Pressure projection solution by PCG on a uniform grid, solving Ax=b.

Given x^0, M
Compute $r^0 = Ax^0 - b$, Solve $Mz^0 = r^0$, Set $p^0 = -z^0$
for $k = 0, Maxiterates$ **do**
  $\alpha^k = (r^k, r^k)/(p^k, Ap^k)$
  $x^{k+1} = x^k + \alpha^k p^k$
  $r^{k+1} = r^k + \alpha^k Ap^k$
  Enfore solvability condition by projecting r^{k+1} into the range of A. $\sum_K r_K^{k+1} = 0$. (Neumann BC on all walls)
  Solve $Mz^{k+1} = r^{k+1}$
  $\beta^{k+1} = (r^{k+1}, z^{k+1})/(r^k, z^k)$
  $p^{k+1} = -z^{k+1} + \beta^{k+1} p^k$
end for

Let us examine the problem. Consider the solution of the matrix system $Ax = b$ by the preconditioned conjugate gradient method of algorithm 8, we want to solve $Mz = r$ where $Q = M^{-1}$ represents our symmetric smoother. We have the following iterative scheme

$$z^{n+1} = z^n + Q(r - Az^n). \tag{3.4}$$

In the case of the Jacobi method, we simply use $M = D$ (the diagonal) and so $Q = D^{-1}$ and we find for our iterative smoother

$$z^{n+1} = z^n + D^{-1}(r - Az^n).$$

For the GSRB smoother, we can break the equation $Mz = r$ into red (R) and black (B) components to find the matrix equation

$$\begin{bmatrix} D_R & C^T \\ C & D_B \end{bmatrix} \begin{bmatrix} z_R \\ z_B \end{bmatrix} = \begin{bmatrix} r_R \\ r_B \end{bmatrix}$$

which can be solved by using the scheme

$$\begin{aligned} z_R^* &= D_R^{-1} r_R \\ z_B &= D_B^{-1}(r_B - C z_r^*) \\ z_R &= D_R^{-1}(r_R - C^T z_B) \end{aligned}$$

which in turn produces the Q matrix

$$Q = \begin{bmatrix} D_R^{-1} + D_R^{-1}C^T D_B^{-1} C D_R^{-1} & -D_R^{-1}C^T D_B^{-1} \\ -D_B^{-1} C D_R^{-1} & D_B^{-1} \end{bmatrix}$$

The iterative solution process is then given by

$$r^* = r - Az^n$$
$$z^{n+1} = z^n + Qr^*$$

Since the best performing smoother is ILU, we would also like a parallel version for the GPU. Ortega [62] provides a red black incomplete Cholesky factorization (ICRB) which can be used here in place of ILU since our matrix A is symmetric positive definite. Using the same red-black form as the GSRB matrix representation, the factorization yields

$$\begin{bmatrix} D_R & C^T \\ C & D_B \end{bmatrix} \rightarrow \begin{bmatrix} I & 0 \\ CD_R^{-1} & I \end{bmatrix} \begin{bmatrix} D_R & 0 \\ 0 & D_B^* \end{bmatrix} \begin{bmatrix} I & D_R^{-1}C^T \\ 0 & I \end{bmatrix}$$

where

$$D_B^* = diagonal(D_B - CD_R^{-1}C^T).$$

It turns out that in this scenario the algorithm is exactly the same as in the GSRB case, but with D_B replaced with D_B^*. This method has provided some modest speedup results for a 2D Dam break problem, particularly when a large number of smoother iterations are performed without any data transfers. Table 3.2 provides some data on our experience with the accelerator on this problem. The findings here show speedup of up to 2.4X, but the performance is penalized by the need to move fixed data to and from the GPU because the PGI accelerator (at the time this work was performed) has no way of allowing one to permanently store data on the GPU in such scenarios. We have since moved on to GPU coding in the compute unified device architecture (CUDA) C language which provides more complete user control.

3.2.3 An Improved Projection Algorithm for Adaptive Meshes

The first algorithm we will discuss here is the currently used MG algorithm for adaptive grids in order to demonstrate why improvement is needed. For the solution of the pressure projection step for incompressible flows in two phases, Tatebe's [76] MGPCG is used as a smoother for each MG level, as described in algorithm 9. The problem with algorithm 9

Table 3.2: Timing results for the GPU accelerated pressure projection solver on a 2D dam break problem with a large aspect ratio

Smoother	Iterations	Cycles	CPU Time	GPU Time	Speedup
GSRB	4	6	2.1 s	1.5	1.40X
GSRB	20	4	4.1 s	1.7	2.41X
ICRB	4	5	2.1 s	1.7	1.24X
ICRB	20	4	4.5 s	1.9	2.37X

is that the MGPCG smoother requires multiple levels of coarsening to occur in order to achieve optimal performance, but on an adaptive mesh these levels may not be available as demonstrated by the 8x2 level 1 grids in figure 3.4. Here, only one coarsening step can be achieved. The MGPCG smoother also requires one to compute on additional grids below the desired level as seen in fig. 3.3. Also, algorithm 9 is sensitive to the number of MGPCG smoothing steps as too many smoothing steps may lead to unecessary increases in computation time, and too few might lead the MG AMR algorithm diverging. To improve this method, we replace the MG algorithm itself with a MGPCG method, and instead use one of three possible MG preconditioning methods; a) block ILU smoother, b) Gauss-Seidel red black (GSRB) smoother, and c) incomplete Cholesky red black (ICRB) smoother. This results in algorithm 10.

We note that the symmetric GSRB smoother that we have employed is not typical. The "GSRB" symmetric smoother employed by Tatebe used a red-black ordering down the MG v-cycle, and a black-red ordering up the v-cycle. Our GSRB smoother differs by using a red-black-black-red (RBBR) ordering at each step of the v-cycle. The ICRB smoother is an adaptation of the parallel incomplete Cholesky factorization of Ortega [62].

Both algorithms 9 and 10 use identical restriction and prolongation operators for the adaptive grid structure. We will define the restriction by operator R and the prolongation by operator P. To clearly define these operators, we will use a numbering system based on the transition from a coarse grid to an adapted fine level grid such as in figure 3.5. Let us assume that we have N coarse grid points at a given level ℓ, $x_1^\ell, x_2^\ell, \cdots, x_N^\ell$, then we will append the subscript with a 1,2,3 or 4 to define it as an adapted grid point at the next finest level. For example, if we adapted the second grid point we would then have $x_1^\ell, x_{21}^{\ell+1}, x_{22}^{\ell+1}, x_{23}^{\ell+1}, x_{24}^{\ell+1}, \cdots, x_N^\ell$, and could continue on adding grid points in this fashion, but here we will just use two levels. Following this numbering system, we find that the prolongation operator can be expressed by the mapping $P : x_i^\ell \rightarrow x_{ij}^{\ell+1}$ for an adapted cell, and $P : x_i^\ell \rightarrow x_i^{\ell+1}$ for a non-adapted cell. If one were to prolong the pressure on the coarse level ℓ, the matrix equation, describing the prolongation operator, would resemble that of eqn. 3.5.

Algorithm 9 Original MG Algorithm for AMR

Given x^0, $r = b - Ax^0$, $x = x^0$, $\delta x = 0$
Repeat until $||r|| < \epsilon$
1. Call `relax`$(\delta x, r, \ell^{max})$ on finest level
2. Let $x = x + \delta x$, $r = r - A(\delta x)$

Recursive Routine `relax`(sol, rhs, ℓ)
if Coarsest Level **then**
  Solve exactly using MGPCG
else
  (a) Presmoothing Step
  for $i = 1$ to presmooth **do**
    Smooth using MGPCG on level ℓ until $||r_\ell|| < \frac{\epsilon}{10}$
  end for
  (b) Restriction Step
    (i) `restrict`(r) to covered level $\ell - 1$ cells and exposed level $\ell - 1$ cells neighboring a covered cell.
    (ii) $cor = 0$
  (c) Relaxation on Next Coarser Level
    Call `relax`$(cor^{coarse}, rhs^{coarse}, \ell - 1)$
  (d) Prolongate the Correction to the present level l cells covering coarse level $\ell - 1$ cells and one layer of "virtual" level ℓ cells.
    $sol = sol + I(cor)$
  (e) Postsmoothing Step
  for $i = 1$ to postsmooth **do**
    Smooth using MGPCG on level ℓ
  end for
end if

Algorithm 10 Improved MGPCG Algorithm for AMR

Given x^0, $r = b - Ax^0$, $x = x^0$, $\delta x = 0$
$z = 0$
Call `relaxAMR`(z, r, ℓ^{max})
$\rho = z \cdot r$
if $n = 1$ **then**
  $p = z$
else
  $\beta = \frac{\rho}{\rho_{old}}$
  $p = z + \beta p$
end if
$\alpha = \frac{\rho}{p \cdot (Ap)}$
$\rho_{old} = \rho$
$x = x + \alpha p$
$r = r - \alpha Ap$

Recursive Routine `relaxAMR`(sol, rhs, ℓ)
if Coarsest Level **then**
  Solve exactly using MGPCG
else
  (a) Presmoothing Step
  for $i = 1$ to presmooth **do**
    Smooth using ILU on level ℓ
  end for
  (b) Restriction Step
    (i) `restrict`(r) to covered level $\ell - 1$ cells and exposed level $\ell - 1$ cells neighboring a covered cell.
    (ii) $cor = 0$
  (c) Relaxation on Next Coarser Level
    Call `relaxAMR`$(cor^{coarse}, rhs^{coarse}, \ell - 1)$
  (d) Prolongate the Correction to the present level ℓ cells covering coarse level $\ell - 1$ cells and one layer of "virtual" level ℓ cells.
    $sol = sol + I(cor)$
  (e) Postsmoothing Step
  for $i = 1$ to postsmooth **do**
    Smooth using ILU on level ℓ
  end for
end if

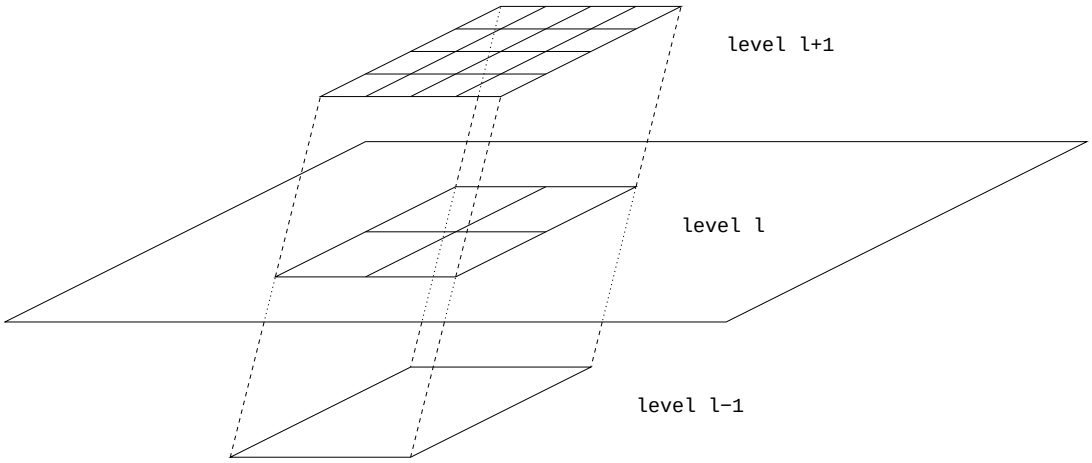

Figure 3.3: Adaptive mesh hierarchy in 2D. To compute the solution at level l+1, the MG AMR algorithm (alg. 9) requires calculations at levels l and l-1.

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & & & & \\ \vdots & & \ddots & & & \\ \vdots & & & 1 & & \\ \vdots & & & 1 & & \\ \vdots & & & 1 & & \\ \vdots & & & 1 & & \\ \vdots & & & & \ddots & \\ 0 & & & & & 1 \end{bmatrix} \begin{bmatrix} p_1^\ell \\ p_2^\ell \\ \vdots \\ p_i^\ell \\ \vdots \\ p_N^\ell \end{bmatrix} = \begin{bmatrix} p_1^{\ell+1} \\ p_2^{\ell+1} \\ \vdots \\ p_{i1}^{\ell+1} \\ p_{i2}^{\ell+1} \\ p_{i3}^{\ell+1} \\ p_{i4}^{\ell+1} \\ \vdots \\ p_N^{\ell+1} \end{bmatrix} \tag{3.5}$$

The restriction operator R can be expressed by the mapping

$$R : p_i^{\ell+1} \rightarrow \begin{cases} \sum_{j=1}^{4} p_{ij}^\ell & \textit{adapted cells to coarse cell} \\ p_i^\ell & \textit{coarse cell to coarse cell} \end{cases}$$

This restriction yields a matrix equation like that of eqn. 3.6 when restricting grid points $x_{i1}, x_{i2}, x_{i3}, x_{i4}$ on a fine level grid to grid point x_i on a coarse level grid.

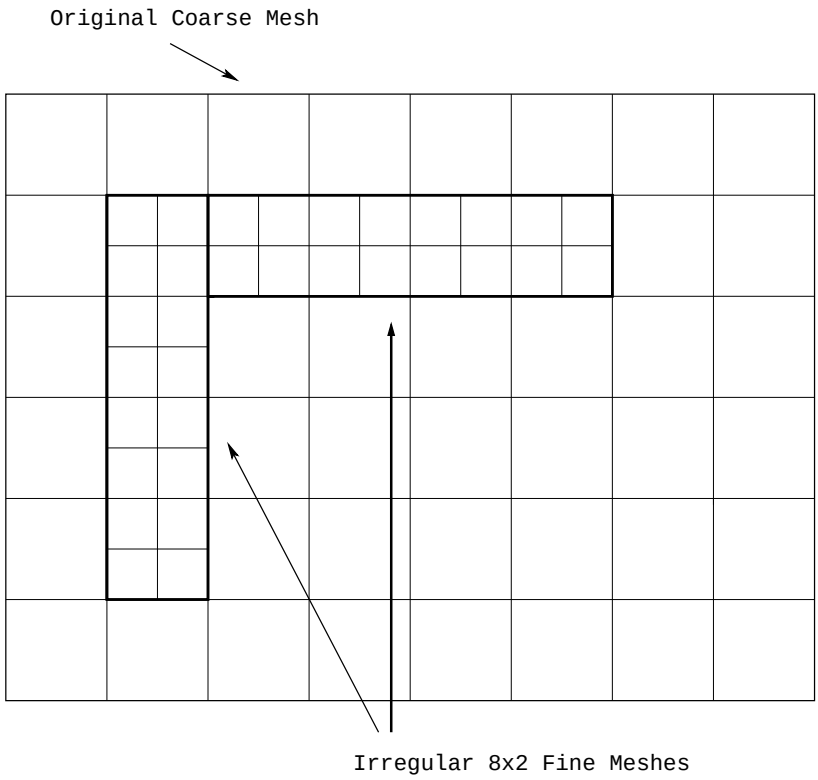

Figure 3.4: An MGPCG smoother can only achieve a single coarsening step on the illustrated fine level; then one must use PCG as a bottom solver on the "irregularly shaped" coarsest domain. On the other hand, our new MG preconditioner coarsens only one level too, but the bottom solver is MGPCG on the whole non-"irregularly shaped" domain. The bottom solver of our new method can coarsen two more times.

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & & & & & & & \\ \vdots & & \ddots & & & & & & \\ \vdots & & & 1 & 1 & 1 & 1 & & \\ \vdots & & & & & & & \ddots & \\ 0 & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} p_1^{\ell+1} \\ p_2^{\ell+1} \\ \vdots \\ p_{i1}^{\ell+1} \\ p_{i2}^{\ell+1} \\ p_{i3}^{\ell+1} \\ p_{i4}^{\ell+1} \\ \vdots \\ p_N^{\ell+1} \end{bmatrix} = \begin{bmatrix} p_1^{\ell} \\ p_2^{\ell} \\ \vdots \\ p_i^{\ell} \\ \vdots \\ p_N^{\ell} \end{bmatrix} \tag{3.6}$$

Matrix eqns. 3.5 and 3.6 show that the restriction and prolongation operators are transposes of each other.

In order to show that algorithm 10 converges, it is sufficient to show that it meets the criteria set forth by Tatebe [76] for convergence of the MGPCG method, i.e. 1) the MG smoother is symmetric, 2) the restriction operator is the transpose of the prolongation operator, and 3) the matrix A in the smoothing step

$$\begin{aligned} x^{k+1} &= x^k + M(b - Ax^k) && \textit{real cells} \\ x^{k+1} &= x^k && \textit{fictitious cells} \end{aligned}$$

is symmetric. One can see that condition 2 is satisfied by viewing the matrix operators for the restriction and prolongation, and A will be symmetric as long as first order boundary conditions are applied [53, 54], so it is sufficient to show that the MG preconditioner produced by `relaxAMR` is symmetric. To show that the MG preconditioner on an adaptive grid is symmetric, we will assume that the grid on each level is a union of 'real' fine cells and 'fictitious' coarse cells such as depicted in figure 3.5. Let M be a symmetric preconditioner produced by ILU on a uniform grid, then we will define the corresponding AMR preconditioner on a given level ℓ by the matrix $\tilde{M}$, which can be defined as

$$\tilde{M} = \begin{cases} M & \textit{real cells} \\ 0 & \textit{fictitious cells} \end{cases}$$

and our equivalent smoothing step is simply $x^{k+1} = x^k + \tilde{M}(b - Ax^k)$. Now, if one were to order the cells in such a manner that the variables corresponding to real cells were listed first, e.g.

$$[x_1, x_2, \cdots, x_N] \rightarrow [x_1^{real}, x_2^{real}, \cdots, x_n^{real}, x_1^{fictitious}, x_2^{fictitious}, \cdots, x_n^{fictitious}]$$

then we find that

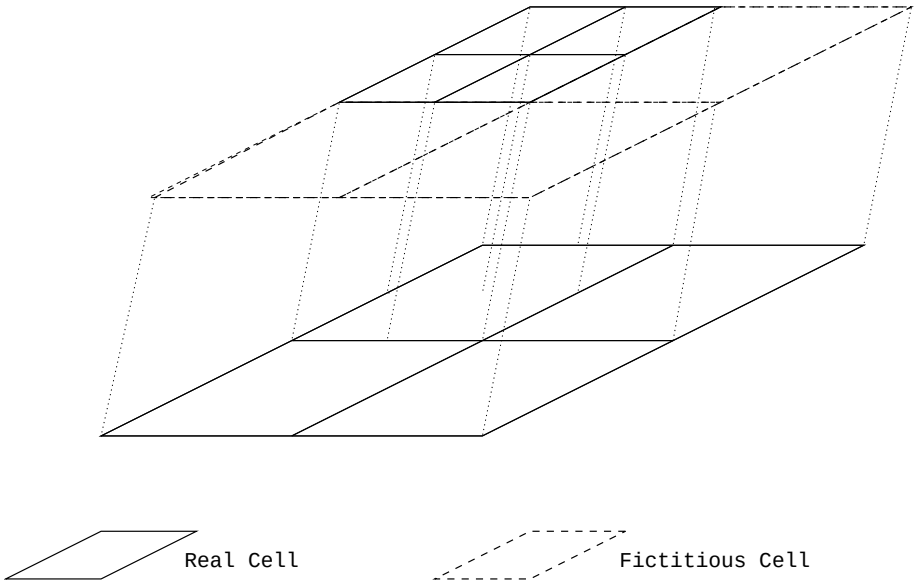

Figure 3.5: Coarse and fine grid levels depicting real and fictitious cells.

$$\tilde{M} = \begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix}$$

which is symmetric and so algorithm 10 is guaranteed to converge.

3.2.4 2D Bubble

The first problem we will discuss is that of a 3D axisymmetric gas bubble (2 computational dimensions) rising through a liquid phase. Figure 3.6 displays representative discretization grids for this problem with increasing numbers of adaptive levels. Timing data for this problem is given in table 3.3 and a table giving the speedup factor of the new MGPCG AMR algorithm over the original MG AMR algorithm is provided in table 3.4. On this problem, the new MGPCG AMR algorithm is typically 2-3X faster than the original MG AMR algorithm, and performs up to 4X faster than the original MG AMR algorithm when using a blocking factor of four along with five adaptive levels. The only case when the new solver underperforms is with a blocking factor of eight and only one adaptive level.

3.2.5 3D Bubble

The second test problem is also a rising gas bubble through a liquid phase, but expanded to three spatial dimensions. Figure 3.7 shows the grid for this problem with two adaptive levels and a blocking factor of four. Timing data for this problem is given in table 3.5 and a table giving the speedup factor of the new MGPCG AMR algorithm over the original MG AMR algorithm is provided in table 3.6. For this test problem, the new MGPCG AMR algorithm shows improvement over the MG AMR algorithm when more than one adaptive

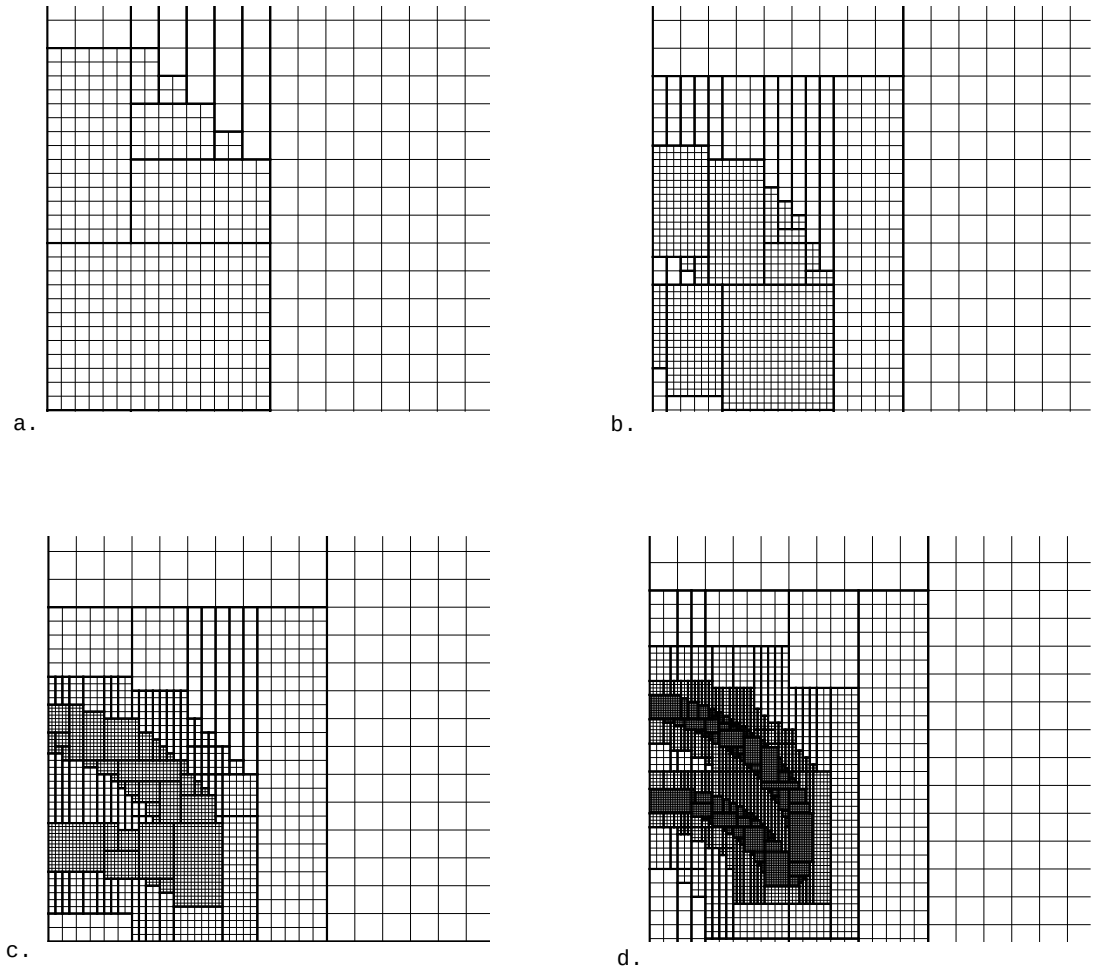

Figure 3.6: AMR grids for a rising 3D axisymmetric bubble. The representative grids all use a blocking factor of two and contain a) one adaptive level, b) two adaptive levels, c) three adaptive levels, and d) four adaptive levels.

Table 3.3: Timing results for a single pressure solve for the new MGPCG AMR algorithm along with the old MG AMR algorithm and the PCG algorithm for the 3D axisymmetric test bubble. The new MGPCG algorithm is faster than the old MG AMR algorithm in nearly every scenario. Grid sizes for a fixed blocking factor and number of adaptive levels are identical for each method.

Blocking Factor	2			4			8		
Adaptive Levels	1	3	5	1	3	5	1	3	5
ILU Smoother									
PCG	0.659	5.424	63.49	0.563	3.359	35.54	0.365	2.875	26.78
MG	0.270	2.181	13.93	0.252	1.349	10.05	0.098	0.751	6.060
MGPCG	0.142	0.636	4.109	0.127	0.439	2.493	0.096	0.382	2.156
ICRB Smoother									
PCG	0.653	5.366	69.49	0.567	3.561	39.02	0.377	3.012	25.53
MG	0.281	2.177	16.54	0.278	1.435	10.96	0.112	0.885	6.634
MGPCG	0.157	0.655	4.511	0.152	0.498	2.732	0.123	0.415	2.402
GSRB Smoother									
PCG	0.641	5.706	65.99	0.567	4.037	37.72	0.364	3.014	29.26
MG	0.284	2.165	13.91	0.266	1.367	10.38	0.108	0.845	5.957
MGPCG	0.153	0.678	4.426	0.145	0.464	2.743	0.118	0.408	2.176

Table 3.4: Speedup factor for the new MGPCG AMR method over the original MG AMR method on the 3D axisymmetric test bubble problem. The new algorithm outperforms the old by an increasing margin as both the blocking factor and the number of adaptive levels are increased. It can be seen that the benefits of the new method are not realized in the case of large blocking factors and a single adaptive level.

Blocking Factor	2			4			8		
Adaptive Levels	1	3	5	1	3	5	1	3	5
ILU	1.91X	3.43X	3.39X	1.99X	3.07X	4.03X	1.01X	1.97X	2.81X
ICRB	1.79X	3.32X	3.67X	1.83X	2.88X	4.01X	0.91X	2.13X	2.76X
GSRB	1.86X	3.19X	3.14X	1.84X	2.95X	3.78X	0.92X	2.07X	2.74X

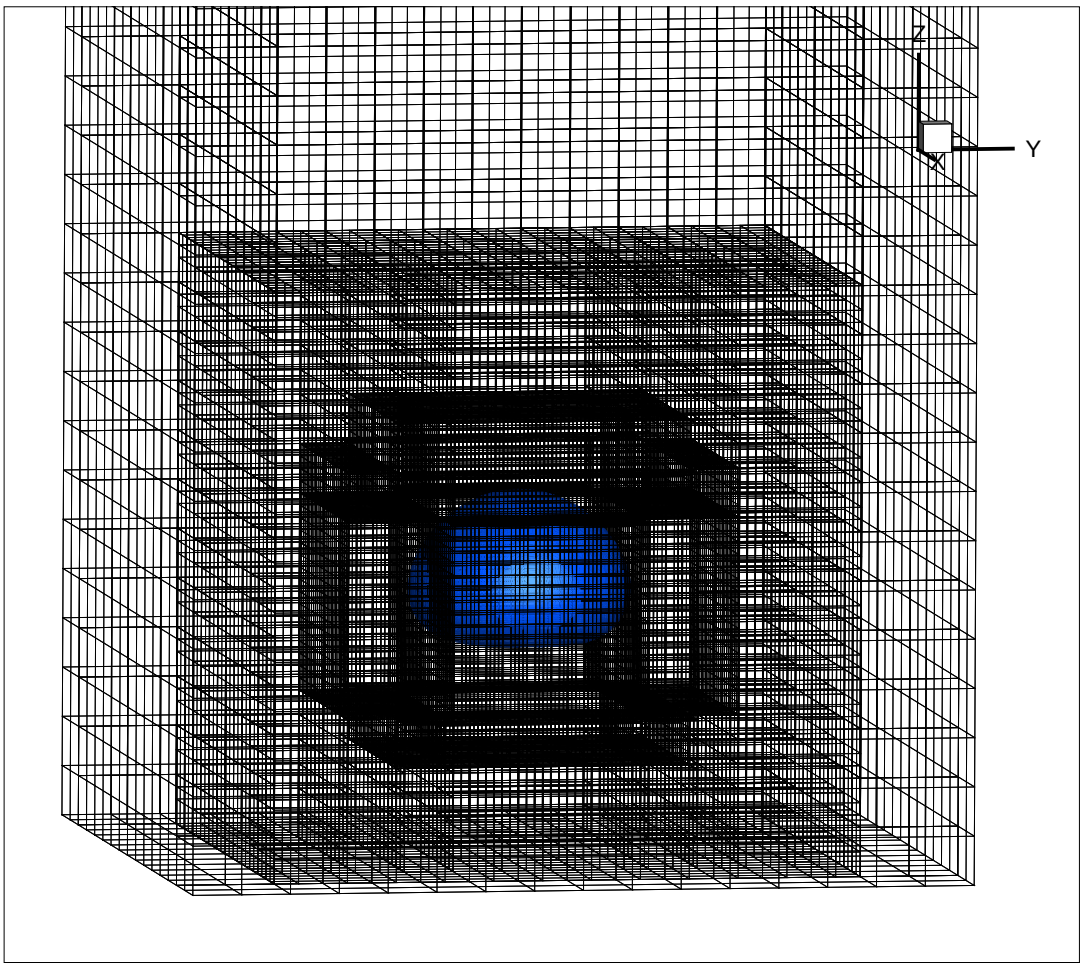

Figure 3.7: Mesh for a 3D rising bubble with 2 adaptive levels and a blocking factor of 4.

level is used and achieves a speedup of 2X in the case of three adaptive levels and a blocking factor of two.

3.2.6 3D Whale

The third problem tested here is simulated flow past a 3D whale body. Figure 3.8 shows a grid with two adaptive levels for this problem, along with the whale body. This test problem shows the best overall speedup of the new method and is 2-4X faster than the old method, even when only solving with one adaptive level.

3.3 FUN3D

The software accelerated in this section is the NASA FUN3D code, which is described in references [5] and [4]. Additionally, the acceleration found here was first described in

Table 3.5: Timing results for a single pressure solve for the new MGPCG AMR algorithm along with the old MG AMR algorithm and the PCG algorithm for a 3D test bubble.

Blocking Factor	2			4			8		
Adaptive Levels	1	2	3	1	2	3	1	2	3
ILU Smoother									
PCG	3.839	18.45	71.11	3.326	14.24	49.79	4.154	22.22	69.70
MG	2.156	9.432	40.46	1.704	6.140	24.02	2.146	7.410	21.68
MGPCG	1.884	6.295	19.05	1.747	5.220	14.63	1.999	6.007	17.53
ICRB Smoother									
PCG	4.158	19.67	75.61	3.682	15.28	56.82	4.770	27.19	84.94
MG	2.227	9.916	41.31	1.910	6.958	30.35	2.490	10.10	28.47
MGPCG	2.354	7.083	19.36	2.263	5.950	16.96	2.686	7.559	20.33
GSRB Smoother									
PCG	4.023	19.99	78.66	3.735	15.43	58.33	4.801	26.19	86.51
MG	2.145	9.708	41.06	1.860	7.086	31.98	2.485	10.10	29.68
MGPCG	2.331	7.149	20.20	2.241	6.191	17.67	2.571	8.001	20.19

Table 3.6: Speedup factor for the new MGPCG AMR method over the original MG AMR method on the 3D test problem of simulating a gas bubble rising through a liquid phase.

Blocking Factor	2			4			8		
Adaptive Levels	1	2	3	1	2	3	1	2	3
ILU	1.14X	1.50X	2.12X	0.98X	1.18X	1.64X	1.07X	1.23X	1.24X
ICRB	0.95X	1.40X	2.13X	0.84X	1.17X	1.79X	0.93X	1.34X	1.40X
GSRB	0.92X	1.36X	2.03X	0.83X	1.14X	1.81X	0.97X	1.26X	1.47X

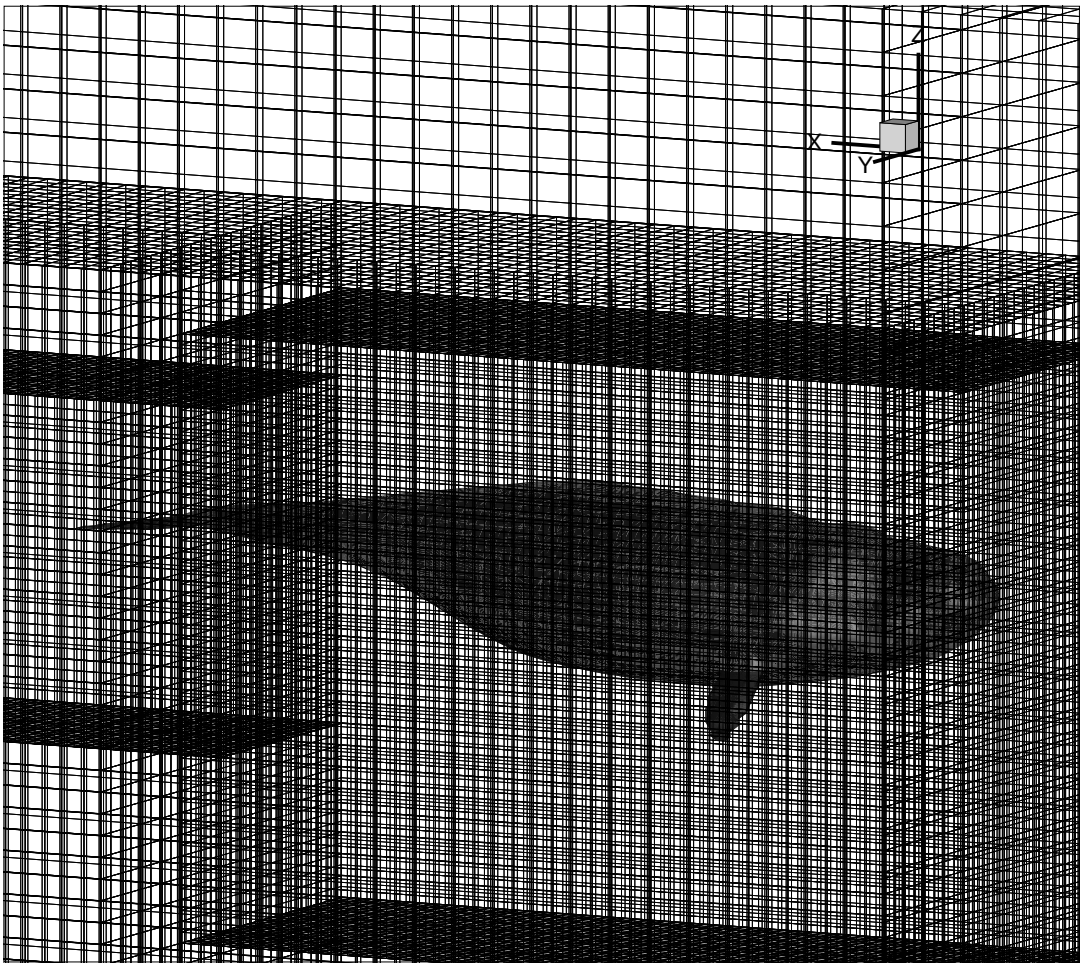

Figure 3.8: Mesh for a 3D whale with 2 adaptive levels.

Table 3.7: Timing results for a single pressure solve for the new MGPCG AMR algorithm along with the old MG AMR algorithm and the PCG algorithm for flow past a 3D whale.

Blocking Factor	2			4			8		
Adaptive Levels	1	2	3	1	2	3	1	2	3
ILU Smoother									
PCG	32.33	126.5	468.1	33.69	122.0	528.6	47.32	198.3	759.9
MG	10.52	41.69	145.6	9.874	35.71	133.4	13.91	57.37	202.2
MGPCG	5.445	15.06	43.66	5.796	14.84	48.74	7.113	21.28	70.07
ICRB Smoother									
PCG	40.15	171.9	636.1	42.80	160.6	705.6	68.93	269.2	1056
MG	14.93	64.01	226.1	14.61	61.08	226.1	24.02	102.5	371.9
MGPCG	7.606	19.93	56.83	8.794	19.45	62.45	10.49	28.93	94.08
GSRB Smoother									
PCG	41.07	179.2	648.8	43.11	167.0	710.2	70.00	276.3	1078
MG	15.99	67.29	230.4	15.42	63.24	234.9	26.03	106.8	395.9
MGPCG	8.063	22.38	57.47	8.451	21.61	65.76	10.39	31.77	94.70

Table 3.8: Speedup factor for the new MGPCG AMR method over the original MG AMR method on the 3D test problem of flow past a 3D whale.

Blocking Factor	2			4			8		
Adaptive Levels	1	2	3	1	2	3	1	2	3
ILU	1.93X	2.77X	3.33X	1.70X	2.41X	2.74X	1.96X	2.70X	2.89X
ICRB	1.96X	3.21X	3.98X	1.66X	3.14X	3.62X	2.29X	3.54X	3.95X
GSRB	1.98X	3.01X	4.01X	1.82X	2.93X	3.57X	2.51X	3.36X	4.18X

[21]. FUN3D is a large production level code consisting of several hundred Fortran 95 modules and approximately 850,000 lines of code. Under development since the late 1980's, FUN3D possesses a large array of aerodynamic simulation and optimization capabilities for both steady and time dependent problems across the speed range. The software has been distributed to hundreds of individuals in academia, industry and other goverment agencies. This work focuses on the porting of a single routine (point implicit colored GS solver) which consumes a substantial portion of the computation.

3.3.1 Mathematical Formulation

The portion of the FUN3D code pertinent to this work is the compressible Reynolds-averaged Navier-Stokes solver. The spatial discretization utilizes an unstructured finite volume scheme with a tetrahedral mesh. The dependent variables are stored at the vertices. Inviscid fluxes at control volume interfaces are computed using the upwind scheme of Roe [67], and viscous fluxes are formed using an approach equivalent to a finite-element Galerkin procedure [5]. For turbulent flows, the eddy viscosity is modeled using the one-equation approach of Spalart and Allmaras [69]. An approximate solution of the linear system of equations formed within each time step is obtained through several iterations of a multicolor point-iterative scheme. The turbulence model is integrated all the way to the wall without the use of wall functions and is solved separately from the mean flow equations at each time step with a time integration and linear system solution scheme identical to that employed for the mean flow equations. In the current implementation, the grid points are numbered using a reverse Cuthill- McKee technique [16] to improve cache performance during flux and jacobian gather/scatter operations. However, the multicolor point-iterative scheme forbids physically adjacent unknowns from residing within the same color group. Therefore, the relaxation scheme is inherently cache unfriendly in its basic form. Since the majority of the floating point operations required to advance the solution take place in the relaxation phase, a mapping is introduced which orders the data in the coefficient matrix by its assigned color as the individual contributions to the jacobian elements are determined. In this manner, memory is accessed in ideal sequential order during the relaxation as would be achieved in a more simplistic scheme such as point-Jacobi, while the substantial algorithmic benefits (improved stability and convergence) of the multicolor scheme are retained.

3.3.2 Parallelization and Scaling

The FUN3D code has demonstrated parallel scalability to tens of thousands of processor cores. This is achieved primarily through efficiient domain decomposition and message passing communication. Pre- and post-processing operations are also performed in parallel using distributed memory, avoiding the need for a single image of the mesh or solution at any time and ultimately yielding a highly efficient end- to-end simulation paradigm. Typical scaling performance for the solver on a range of mesh sizes is shown in Figure 3.9, with meshes in the millions of nodes. A mesh typically contains six times as many tetrahedral cells than the number of nodes, and so the 105 million node mesh equates to 630 million tetrahedral cells. The majority of these results have been generated on a Silicon Graphics ICE platform consisting of dual-socket, quad-core Intel Xeon "Harpertown" processors. The data marked "Westmere" has been generated using dual- socket, hex-core "Westmere"

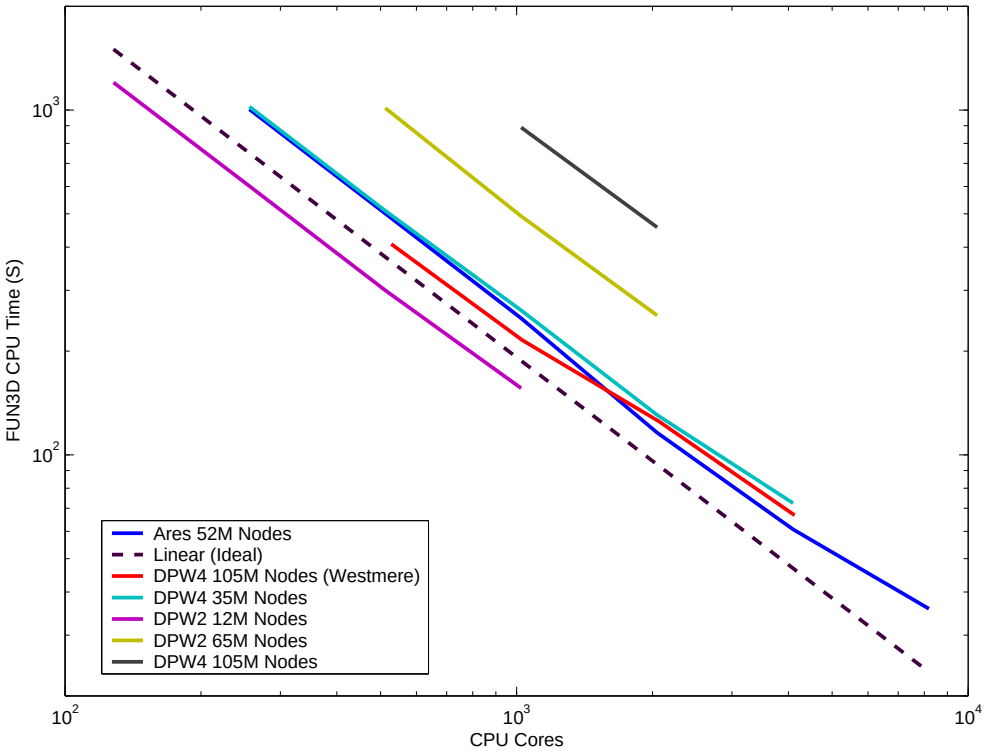

Figure 3.9: Parallel scaling results for FUN3D. Problems are given by node size, with M equating to millions of nodes. There are approximately 6 times as many tetrahedral cells as there are nodes for a given problem size. Ares is a rocket geometry and DPW grids represent aircraft configurations from AIAA Drag Prediction Workshops. The linear dashed line represents ideal scaling.

processors. The implementation scales well across the range of processing cores shown.

One factor in maintaining scalable performance within FUN3D is the load balancing of the computation and communication costs over the processors, which inherently involves the efficient partitioning of the underlying unstructured grid. The mesh partitioning software Metis and ParMetis [47, 48] is used by many CFD codes. Metis' partitioning objectives attempt to evenly distribute the number of nodes (work) to each processor, while minimizing the number of adjacent elements assigned to different processors (communication). In graph theory, these objectives translate to minimizing the edge-cuts and minimizing the total communication volume.

3.3.3 GPU Acceleration

To improve the performance of the FUN3D code, a minimally invasive accelerator model has been implemented in which code portions are ported to the GPU. This allows for an

increase in speed without altering the proven methods of the underlying solvers. The FUN3D code portion targeted for acceleration was the point implicit subroutine used for groups of 5x5 matrix equations which represent the linearized form of the discrete mean flow equations. These correspond to the density, velocity and pressure variables ρ, u, v, w, and p. This routine can account for as much as 70% of the program's total CPU time (such as when solving the Euler equations), but is typically closer to 30% (RANS), depending on the required number of sweeps by the colored Gauss-Seidel (GS) solver for the particular problem. Cache size can play a significant role on this routine and on unstructured codes in general. For this reason along with the need for concurrent kernel execution capabilities, only the latest generation Nvidia Fermi architecture hardware is targeted, since these possesses true L1 and L2 caches which were previously unavailable. On the CPU side, each core performs the colored GS solve sequentially, so a natural mapping to the GPU is created with single threads computing the solutions to the individual 5x5 block solves of a color in parallel. A CUDA C subroutine call has been inserted into the original Fortran `point_solve_5` (PS5) subroutine to carry out the code porting. The computation of the right hand side (RHS) solve between colors has effectively been moved to the GPU when sufficient available memory and double precision capabilities are present. A general outline of PS5 is given in Algorithm 1, noting single precision (SP), double precision (DP) and mixed precision (MP) segments. For the test problems, 10,000's to 100,000's of threads are launched, each computing an entire 5x5 block solve corresponding to an individual equation for the particular color. CUDA C was chosen because it allows for the most explicit user control over the previously mentioned Fermi architecture GPUs that we are employing as accelerators. CUDA C also provides portability as it is is widely used and the compiler is freely available. Given the code, one could (with a little effort) extend this capability to other accelerators with OpenCL, as the language extensions are similar. In general, any accelerator used must have capabilities for both concurrent kernel execution and double precision arithmetic, though there are exceptions that will be discussed further.

Algorithm 11 PS5 Subroutine Outline

```
for color = 1 to max_colors do
  b(:) = residual(:)
  // Begin RHS solve
  for n = start to end do
    Step 1. Compute off diagonal contributions
    for j = istart to iend do
      b_i = b_i − Σ_{i≠j} A_{i,j} * x_icol (SP)
    end for
    Step 2. Compute 5x5 forward solve (DP)
    Step 3. Compute 5x5 back solve (DP)
    Step 4. Compute sum contribution, update RHS (MP)
    Step 5. Accumulate Sum Contributions (GPU Only)
  end for
  // End color, do MPI communication
  Call MPI transfer
end for
```

3.3.4 GPU Distributed Sharing Model

The GPU code has been developed with four different CUDA subroutines; the first dedicated for a single core-single GPU scenario, and the remaining three for multi-core scenarios which require MPI communication transfers at the end of each color sweep. The MPI versions are based on what we call a GPU distributed sharing model. We will define this as an accelerator model where work is efficiently distributed across multiple processors which share a single GPU. This should be done in such a manner that, if necessary, the CPU cores perform some amount of the computational work which would otherwise be done by the GPU in order to ensure that kernel execution remains optimal. This is an important concept since large kernels can place large demands on available GPU resources when called simultaneously, particularly when they use many registers and large chunks of shared and constant memory. In fact, if too many resources are required, kernels from multiple threads could be scheduled to execute sequentially, causing a substantial or complete loss of performance gains. Sharing kernel work with multiple CPU threads should also reduce core idle time, leading to more resource efficient execution. One constraint we have found inherent to this model is a reduction in available GPU memory. As data is distributed to multiple CPU cores, there are more individual structures that are padded when put into GPU memory, leading to a reduction in the available global memory for each additional thread. In all of our test cases on a GTX 480 card, each additional thread sharing the GPU reduces the available global memory predictably within the 63-68 MB range, independent of problem size. This yields an average loss of approximately 66.7 MB of available memory per shared CPU thread, for exactly the same amount of data being stored.

We consider GPU sharing to be an important feature of new and future hybrid CFD codes. We recognize that on a large shared hybrid cluster one could simply use one core per GPU, allowing the remaining cores of a node to be allocated to other tasks. However, with a scalable CFD code it is optimal to utilize as many cores as possible, since this should lead to the best overall performance. Consider running a CFD code that easily scales to thousands of cores on a 256 GPU cluster. Clearly, one would produce better times with 1024 cores and 256 GPUs than with only 256 cores and 256 GPUs, even if the speedup over the CPU cores alone was not as dramatic.

We will now present the four new CUDA subroutines introduced into the FUN3D code, and note that only solution data transfers to and from the GPU are required within the subroutine itself. All other necessary data is transferred externally in optimal locations. Also note that these subroutines only replace a single sweep of the colored GS solver, as MPI communications are typically necessary after every sweep.

3.3.5 CUDA Subroutines

- `gs_gpu` : The first subroutine is for a straightforward 1 core - 1 GPU setup where the entire GS solve is done in CUDA. The kernel for this scenario computes the entire RHS solve, and is called repeatedly through the color sweeps without any data transfers. In this case, a single GPU thread maps to a single control volume, computing its own 5x5

block solve (steps one through four in Algorithm 1). After a thread synchronization, the kernel keeps five threads to accumulate the sum contributions for the five variables (step 5 of Algorithm 1) before moving on to the next color. This scenario does not require MPI communication, and hence provides the biggest performance gains, but is limited to a single processor. We will refer to this routine as `gs_gpu`, which does not require concurrent kernel execution. The kernel for this routine allocates 21 registers per thread, and 136 bytes of constant memory per thread block at compile time, as found by the `--ptxas-options=-v` compiler option. It requires one CPU to GPU solution transfer at the beginning of the call, and one GPU to CPU transfer at the end, but none between colors. We note that our unstructured code is unable to effectively use the 48 KB shared memory space, and opt instead to swap this larger section with the 16 KB L1 cache space using the `cudaFuncSetCacheConfig` function. Using the larger L1 cache space provides a noticeable 6% performance increase over the smaller version for this routine.

- `gs_mpi0` : This case is identical to that above in terms of the GPU kernel, however, it assumes that an MPI communication is required at the end of each color. Here, the CUDA routine must copy the solution back from the GPU and return it to the Fortran subroutine to perform the transfer, providing additional overhead. We will refer to this routine as `gs_mpi0`, which uses the same kernel as `gs_gpu` and hence maintains identical GPU resource allocation properties. This routine, along with the other MPI CUDA routines, require two additional data transfers (one CPU to GPU and one GPU to CPU) for each color.

- `gs_mpi1` : This subroutine is the first to consider the effects of GPU sharing, by attempting to reduce the kernel size and distribute a small portion of the workload to the CPU. When developing kernels, one must try to reduce CPU-GPU memory copies at all costs, even if that means performing less than ideal tasks on the GPU side. The kernel of the first two subroutines performs a final accumulation of sum contributions with a mere 5 threads, and so this portion of code does not benefit from mass thread parallelism, but does reduce transfer overhead by keeping sum contributions on the GPU. As more threads share the GPU, this potential overhead can be reduced since multiple CPU cores can copy their respective sum contributions back in parallel. Once there, the more powerful CPU cores should be able to perform the accumulations faster. This concept is implemented in subroutine `gs_mpi1`. Moving this small amount of work to the CPU reduces the constant memory by a mere 8 bytes and does not reduce the kernel register count.

- `gs_mpi2` : The final subroutine was designed to execute with many threads sharing a single GPU. It shares a substantial amount of computation with the CPU, and also eliminates all double precision calculations from the GPU side. This is accomplished by cutting the kernel short and computing the double precision forward and backward solves along with the sum and solution updates on the CPU, only computing step 1 of Algorithm 1 on the GPU. In the process, the need to transfer the diagonal of the LHS matrix A is eliminated, leading to a reduction in data transfer costs and more importantly, freeing up valuable global memory space. The kernel for this subrou-

Table 3.9: FUN3D Grid partitioning over 8 processor cores, showing that node distributions are fairly well balanced.

Cores	1	2	3	4	5	6	7	8
1	339206							
2	168161	171045						
4	85339	84626	84861	84380				
8	42824	40608	42828	42622	42555	42525	42726	42518

tine reduces the register count by 1, while maintaining the same constant memory requirements as the kernel of `gs_mpi1`.

3.4 Test Problems

The GPU accelerated FUN3D code has been tested with a number of problem sizes ranging from approximately 16,000 to 1.1 million grid points (96,000 - 6.6 million elements), but will describe in more detail two particular scenarios.

3.4.1 Test Problem 1

The first test problem is a generic wing-body geometry used in grid generation training at NASA Langley [3]. It is a pure tetrahedral grid, with 37,834 triangular boundary faces, 339,206 Nodes and 1,995,247 Cells. The node partitioning is given in Table 3.9, the surface grid is given in Figure 3.10, along with the GPU computed pressure solution for $\frac{p}{p_\infty}$ at convergence. This setup calls 20 PS5 sweeps per iteration, accounting for approximately 70% of the FUN3D total run time. This problem is inviscid, has a Mach number of 0.3 and an angle of attack of 2.0 degrees.

3.4.2 Test Problem 2

The second problem is a DLR-F6 wing-body configuration used in the second international AIAA Drag Prediction Workshop (DPW-II) [51]. The versions used here include the coarse grid containing 1,121,301 nodes and 6,558,758 tetrahedral cells, and also a reduced grid with approximately 650,000 nodes and 3.9 million tetrahedral cells. In this case, PS5 corresponds to approximately 30% of the total FUN3D wall clock time. The smaller grid along with the corresponding GPU computed pressure solution for $\frac{p}{p_\infty}$ are given in Figure 3.11, corresponding to a turbulent solution with a Mach number of 0.76, an angle of attack of 0.0 degrees and a Reynolds number of 1 million.

3.5 FUN3D Results and Discussion

Two machines were used for timing studies: a dual socket Dell workstation with Intel Xeon 5080 CPUs (2 cores @ 3.73 GHz, 2 MB L2) and a single Nvidia GTX 480 GPU. The second machine is a small Beowulf GPU cluster with a head node and two Fermi equipped

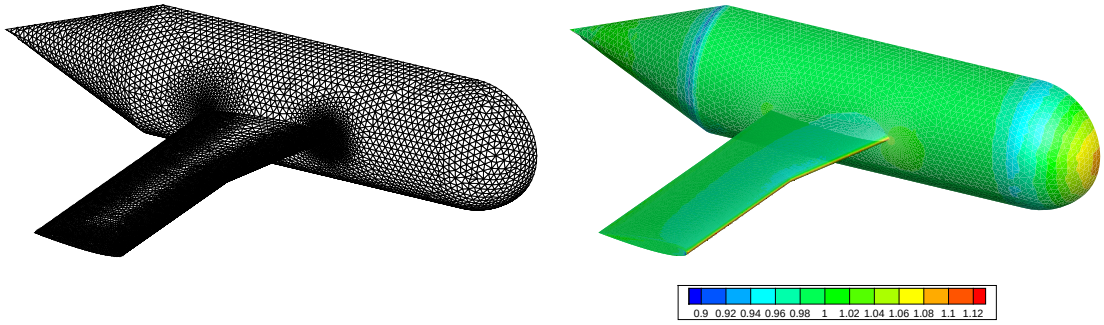

Figure 3.10: Test problem 1 grid (left) and GPU computed pressure solution for $\frac{p}{p_\infty}$ (right). Generic wing-body geometry, contains 339,206 nodes and 1,995,247 tetrahedral cells. Pressure solution obtained at convergence of the FUN3D solver (132 iterations). Inviscid, Mach Number = 0.3, Angle of Attack = 2.0 degrees.

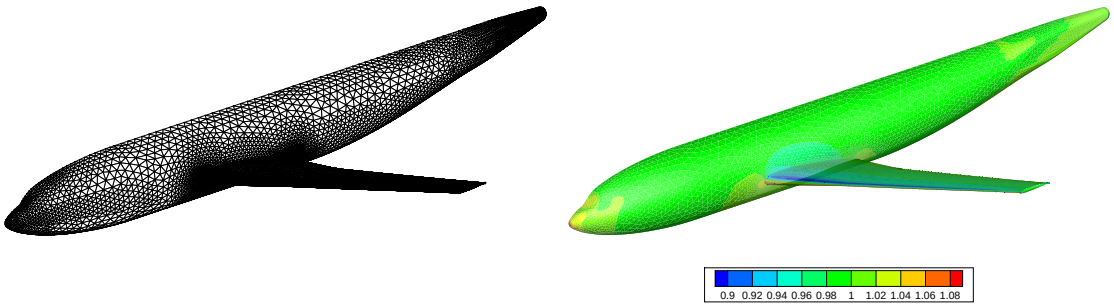

Figure 3.11: Test problem 2 grid (left) and GPU computed pressure solution for $\frac{p}{p_\infty}$ (right). DLR-F6 wing-body configuration, contains approximately 650,000 nodes and 3.9 million tetrahedral cells. Turbulent, Mach Number = 0.76, Angle of Attack = 0.0 degrees, Reynolds Number = $1,000,000$.

Table 3.10: Single core performance results for routine gs_gpu on test problem 1 using the GTX 480 WS.

GTX 480 WS - `gs_gpu`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1732.8 s	794.1 s	2.182X
PS5 Subroutine	1	0.455 s	0.082 s	5.549X

compute nodes connected by a gigabit switch. Each compute node possesses an AMD Athlon II X4 620 processor (4 cores @ 2.6 GHz, 2 MB L2) and a single Nvidia GTX 470 card. We note that we have recently gained access to a GPU research cluster at Nvidia corporation and have begun further testing on a much larger scale, with acces to 96 CPU cores and 16 Tesla Fermi GPUs. We do not include any conclussions from this further study here as it is not yet complete, but note that some initial timing data is included in Appendix A, along with more detailed data using the machines mentioned here.

3.5.1 Single Core Performance

Performance results for a single sweep of the PS5 solver on the GTX 480 workstation for a range of grid point sizes are given in Figure 3.12. The best performing subroutine, `gs_gpu`, achieves a speedup of approximately 5.5X regardless of problem size. Performance for the `gs_mpi0` routine which utilizes the same CUDA kernel is cut in half as a result of CPU-GPU data transfers necessary for MPI communication between colors. We see that version `gs_mpi2` comes closest to the performance of `gs_gpu`, and believe that this is due to the removal of double precision calculations, as the GeForce series cards have been purposely de-tuned for these. We speculate that `gs_mpi1` would achieve the best MPI performance in the presence of higher end Tesla series cards with full double precision capabilities. Single core speedups on the Beowulf cluster are not as good, with a best case of 4.75X. This is due to the fact that the CPU times are slightly better on a single AMD core, and that the GTX 470 is slightly less capable than the 480 model. For the single core scenario on both machines, the overall FUN3D code is accelerated by a factor of 2X or more for test problem 1, which spends about 70% of its total CPU time in the PS5 subroutine. We note that our CUDA subroutines execute the fastest with only 8-16 threads per block, well below the Nvidia minimum recommendation of 64. This is likely due to the additional cache resources available to each thread. While we are primarily concerned with large scale parallel applications, these single core results provide valuable insight. We have learned here that even in the absence of MPI transfers, speedups are limited and well below the par set by so many other GPU acclerated applications. We do note, however, that larger cache sizes could help narrow the gap between this unstructured code and those that benefit from ordered memory access patterns. We also find that in a single core setting our GPU accelerated implicit solver has exceeded the 2.5-3X speedup achieved by that of the OVERFLOW code [46], perhaps a more fitting standard for comparison.

Table 3.11: Single core performance results for routine gs_gpu on test problem 1 using the GTX 470 BC.

GTX 470 BC - `gs_gpu`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1728.9 s	860.5 s	2.001X
PS5 Subroutine	1	0.437 s	0.092 s	4.750X

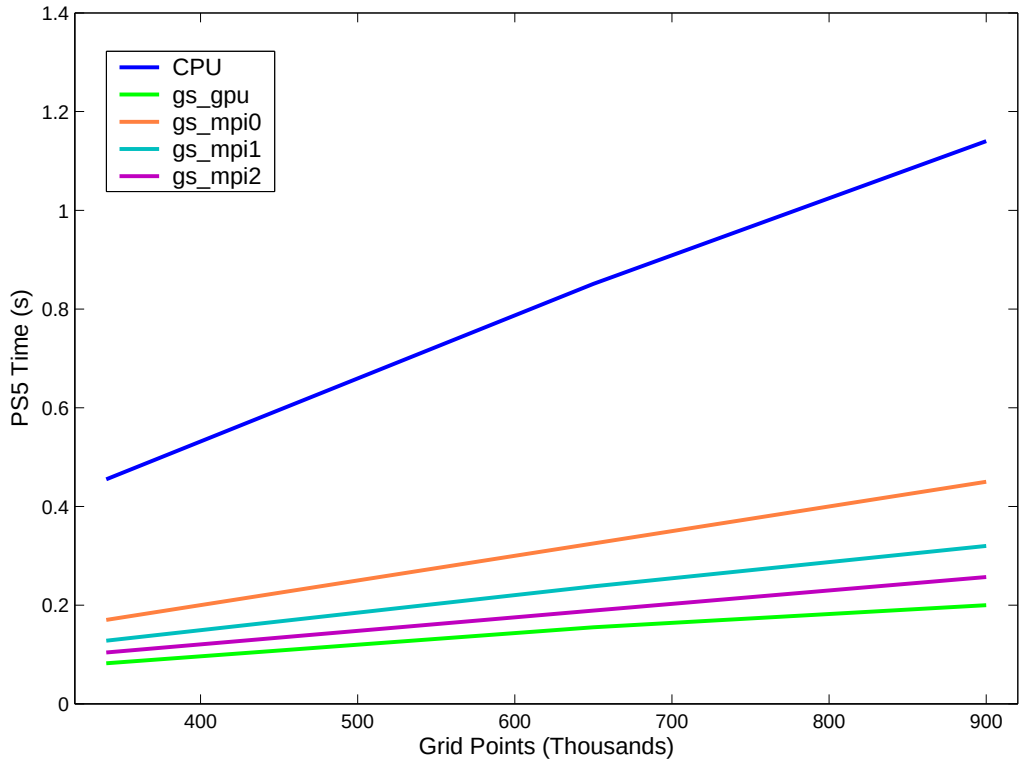

Figure 3.12: CPU and GPU performances for a single PS5 sweep with varying test case sizes. Tests were run on a single core of an Intel Xeon 5080 CPU mated with a single NVIDIA GTX 480 GPU.

Table 3.12: Timing results for routine gs_mpi2 on test problem 1 using the GTX 480 WS.

GTX 480 WS - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1732.8 s	845.7 s	2.049X
PS5 Subroutine	1	0.455 s	0.104 s	4.375X
FUN3D Code	2	850.6 s	543.5 s	1.565X
PS5 Subroutine	2	0.216 s	0.087 s	2.483X
FUN3D Code	4	496.0 s	438.3 s	1.132X
PS5 Subroutine	4	0.125 s	0.092 s	1.359X

Table 3.13: Timing results for routine gsmpi2 on test problem 1 using 1 node of the GTX 470 BC.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1728.9 s	908.1 s	1.904X
PS5 Subroutine	1	0.437 s	0.113 s	3.867X
FUN3D Code	2	963.9 s	651.4 s	1.480X
PS5 Subroutine	2	0.240 s	0.105 s	2.286X
FUN3D Code	4	674.3 s	548.3 s	1.230X
PS5 Subroutine	4	0.172 s	0.109 s	1.578X

3.5.2 Multiple Core Performance

The Beowulf GPU cluster allows us to study scenarios up to 8 cores, 2 GPUs. Figures 3.13 and 3.14 show strong and weak scaling results. Strong scaling within a single node shows the limitations of a GPU distributed sharing model, namely, that if developed properly, the GPU code should execute at approximately the same speed regardless of the number of cores per GPU. Due to this factor, local CPU scaling will ultimately provide the limit on the number of cores that can share a single GPU. We see here from the strong scaling figure that a core limit is not reached for this machine, but it would likely be four with more cores present. Weak scaling results are comparable to the original code on two nodes, but testing on a larger cluster is needed to provide better insight. Viewing the weak scaling results, we find that in a grid point for grid point manner the PS5 subroutine runs about 40% faster when the nodes employ GPUs. This may not seem significant considering the high expectations of GPU accelerated codes, but one must consider that the code is unstructured, there are 20 or more CPU-GPU solution transfers per sweep (one to the GPU and one back to the CPU for each GS color), and this is achieved with four processor cores sharing a single low cost (about $350) gaming card.

3.5.3 Discussion

While the results obtained in this work are limited to a 2X overall code speedup, they do indicate future potential for the use of large GPU clusters with production level unstructured CFD codes. Considering that GPU chips are advancing at a faster pace than CPU versions, it may very well be the case that this code performs substantially better on the next

Table 3.14: Timing results for routine gsmpi2 on test problem 1 using 2 nodes of the GTX 470 BC.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	2	881.1 s	478.8 s	1.840X
PS5 Subroutine	2	0.221 s	0.061 s	3.623X
FUN3D Code	4	498.8 s	348.4 s	1.432X
PS5 Subroutine	4	0.121 s	0.059 s	2.051X
FUN3D Code	8	366.0 s	315.8 s	1.159X
PS5 Subroutine	8	0.088 s	0.063 s	1.397X

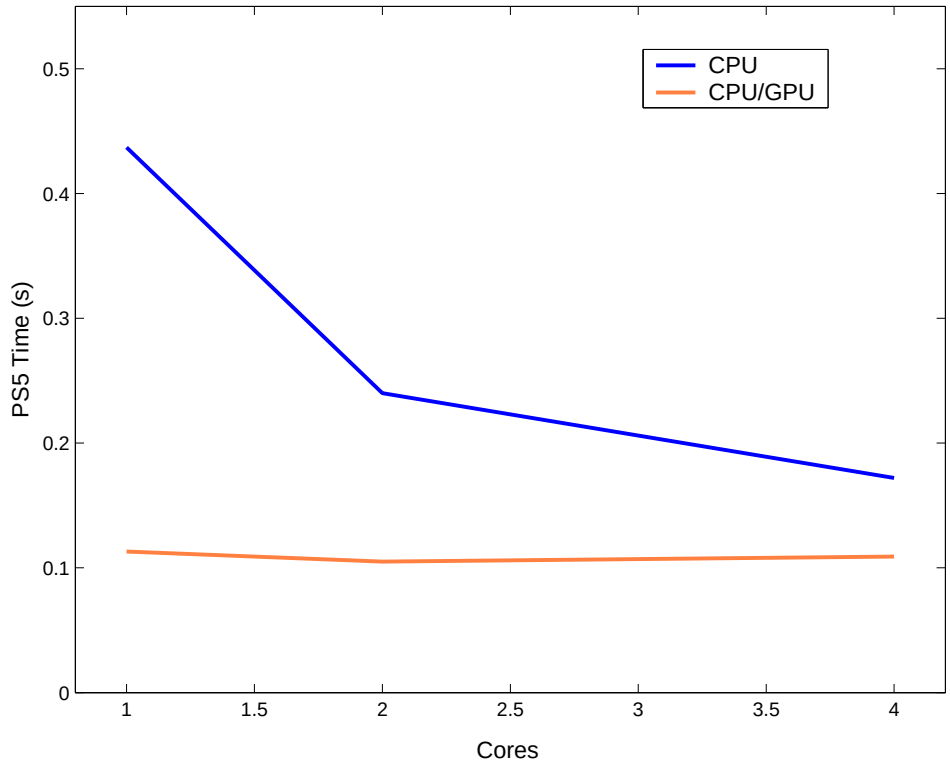

Figure 3.13: Strong scaling within a single node for a single sweep of the PS5 subroutine using CUDA version gs_mpi2. Tests were run on 2 nodes of a Beowulf GPU cluster, each running an AMD Athlon II X4 quad core processor and Nvidia GTX 470 GPU. Strong scaling results use a 340,000 grid point test problem on one node.

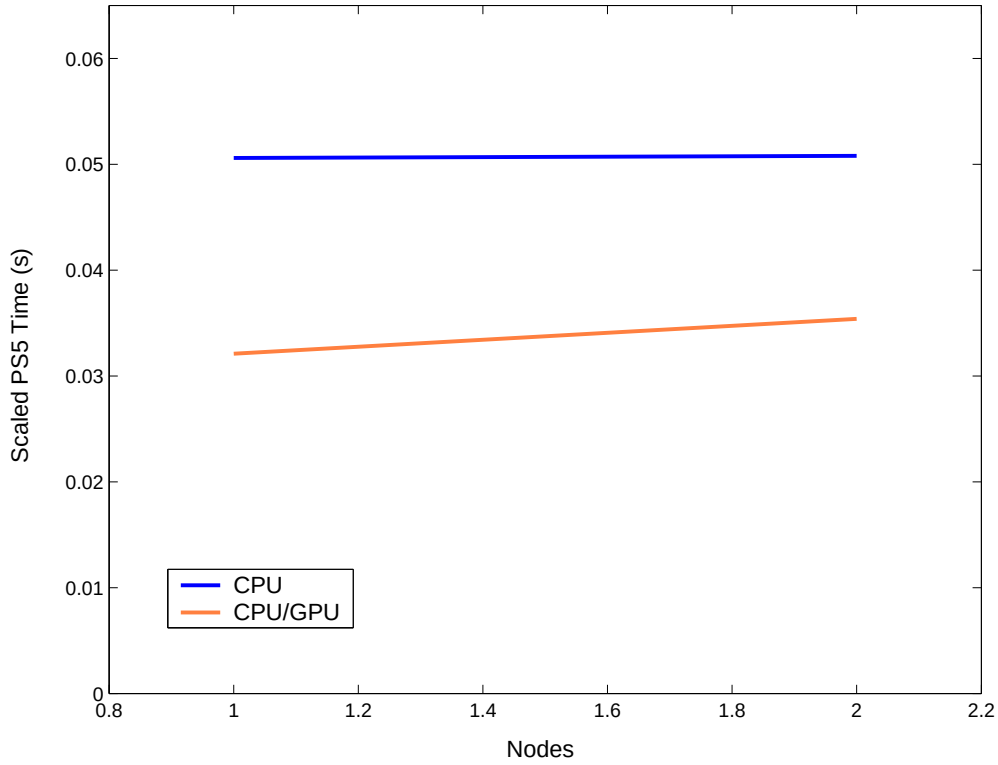

Figure 3.14: Weak scaling across two compute nodes for a single sweep of the PS5 subroutine using CUDA version gs_mpi2. Tests were run on 2 nodes of a Beowulf GPU cluster, each running an AMD Athlon II X4 quad core processor and Nvidia GTX 470 GPU. Weak scaling results use a 340,000 grid point test problem on one node, and a 650,000 grid point problem distributed across 2 nodes. Weak scaling times are per 100,000 grid points.

Table 3.15: GPU architecture evolution from G80, which approximately coincided with the release of Intel's quad core CPUs, to Fermi which coincided with the release of Intel's six core processors. GPU advancements over the last few years have noticeably outpaced those of CPUs. Representative GPUs are: G80-GeForce 8800 GT, GT200-Tesla C1060, Fermi-Tesla C2050. [1]shared memory, [2] texture memory, [3]Configurable L1/shared memory.

Architecture	Year	Cores	L1 Cache	L2 Cache	Memory Access Speed
G80	2006	112	16 KB [1]	128 KB [2]	57.6 GB/s GDDR3
GT200	2008	240	24 KB [1]	256 KB [2]	102 GB/s GDDR3
Fermi	2010	448	48/16 KB [3]	768 KB	144 GB/s GDDR5

generation architecture with larger cache sizes and faster memory access speeds. Table 3.15 provides a look at the rapid advancements occurring in GPU technology, which is beginning to noticeably outpace those of CPUs. Should these trends continue with more available cache and faster memory access speeds, hybrid clusters will become much more amenable to the acceleration of unstructured codes. However, if GPUs are to play an integral role in large scale parallel computations invovling high level codes, data transfer technologies will also need to improve to reduce the performance penalty incurred through CPU-GPU data transfers, as these will provide the biggest bottleneck.

CHAPTER 4

FUTURE RESEARCH

In this chapter we bring together the work on PDE constrained optimization and on CFD code acceleration and look forward to the the development of a simulation based design code for microfluidic applications. While significant contributions to this dissertation come from the study of adjoint methods and the acceleration of the FUN3D code, we note that gradient based aerodynamic design via adjoint methods is well studied and that this is one of FUN3D's current capabilities. The acceleration of the FUN3D point implicit solver should directly provide accelerated adjoint based design capabilities (adjoint solvers essentially require two flow solves per iteration), but we will not pursue this topic any further here. We also note that we are currently working in the early stages of a collaboration with Nvidia Corporation and the Computational AeroSciences Branch at NASA Langley Research Center to further develop FUN3D on GPU based clusters. Since the future direction of this work is uncertain at this time, we can not include any specific discussion here. In addition, we are also developing an improved version of the GPU accelerated CLSVOF projection smoothers which will use CUDA instead of the PGI Fortran Accelerator. We will now move on to discuss the development of a code for the numerical design of microfluidic T-junctions used in lab-on-a-chip devices.

4.1 Towards the Computational Design of Microfluidic Structures

An SBD application we are interested in is the design of midrofluidic T-junctions for lab-on-a-chip designs. This work is a planned collaboration with Micheal Roper's (FSU Chemistry) microfluids lab who will provide experimental data for verification of our numerical models. The main idea of the so called 'Lab on a Chip' setup is to allow for various high throughput assay applications in which the reactions of precisely measured droplets are observed in microfluidic channels. Recent research and applications of such devices are described in the works [27,68,75], which also provide a solid foundation of information relating to the aspect of droplet break up in T-junction geometries. In order to create droplets of a particular size, microfluidic T-junctions are employed; see the left side of Figure 4.1. We can see from the figure that the junction consists of two channels, a horizontal channel which contains a continuous or 'carrier' fluid, and another channel perpendicularto to the first channel which contains the discontinuous fluid which forms the droplets. In [27], a good

discussion of droplet and bubble formation and breakup in both liquid-liquid and gas liquid systems is given along with associated findings on the relationships between various input factors such as fluid pressures and viscosities for particular setups. It is in this work that we find two important conditions in the 'squeezing' mechanism which leads to droplet rupture, first that the width of the main channel should be greater than its height, and second that the width of the inlet channel be at least half the main channel's width. While this work provides some guidance for the design of T-junctions under certain conditions, the models found through experimentation break down once parameters are changed. We feel that this is a problem which can be best tackled by simulation based design utilizing state of the art algorithms for both simulating the fluid flow through these microfluidic channels, and for optimizing the design of the T-junction itself. From Figure 4.1 (left side) one can see that the basic design of a T-junction can be described using the precise height and width of the individual channels. We propose to use varying control points within the T-junction which can be shifted to produce simulations resulting in droplets of varying sizes. The precision in the size of these drops is important as they travel through the trap array since they may not be properly caught if they are not the correct size.. Other important factors which contribute to the control of the droplets include the surfactants used in the process and the flow rates of the individual phases (e.g. oil and water), variables which can easily be incorporated into a non-intrusive derivative free optimization strategy.

Numerical simulations of the T-junction geometries using data from Roper's lab setup are given in figure 4.2. We propose to use our CLSVOF code along with a MGMDS algorithm to carry out the design optimization of the channel dimension, first using only a small number of control points to define the geometry. As the project progresses we intend to include more design variables including surfactant properties. The end result will be a state-of-the-art microfluidic design code capable of producing optimal channel designs for two phase flows at the micro level.

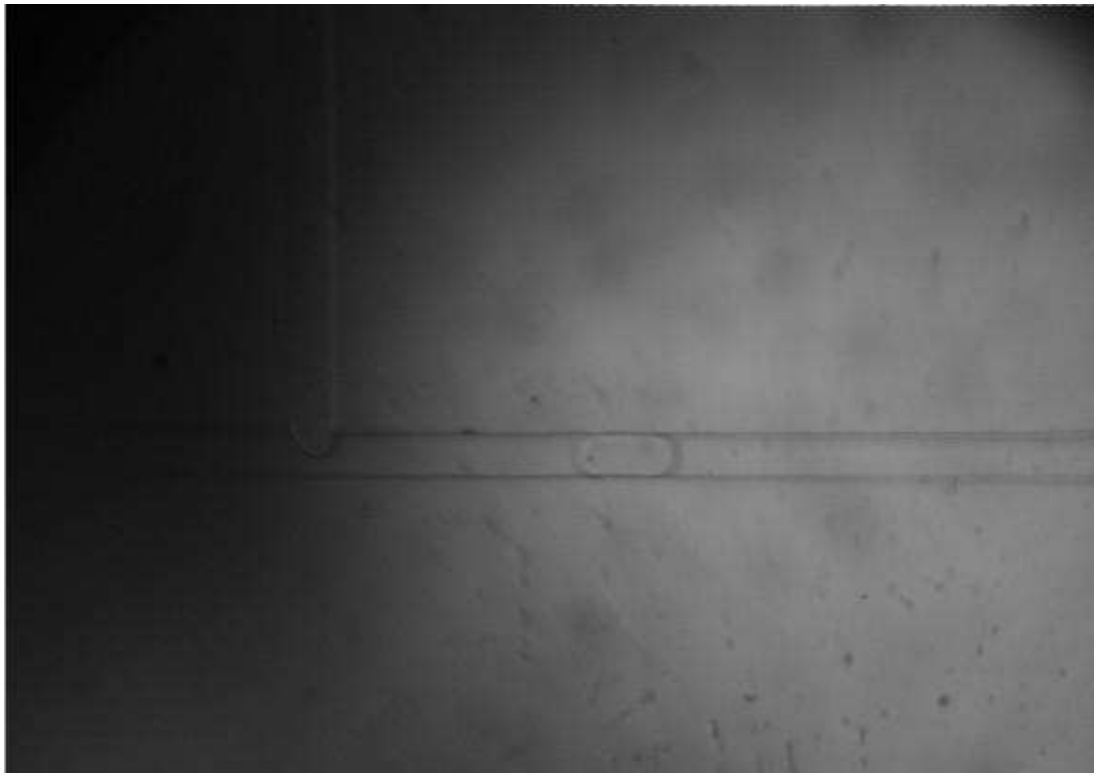

Figure 4.1: Microfluidic T-junction experiment from Roper's laboratory (FSU Chemistry) showing a T-junction geometry with a droplet traveling via a carrier fluid.

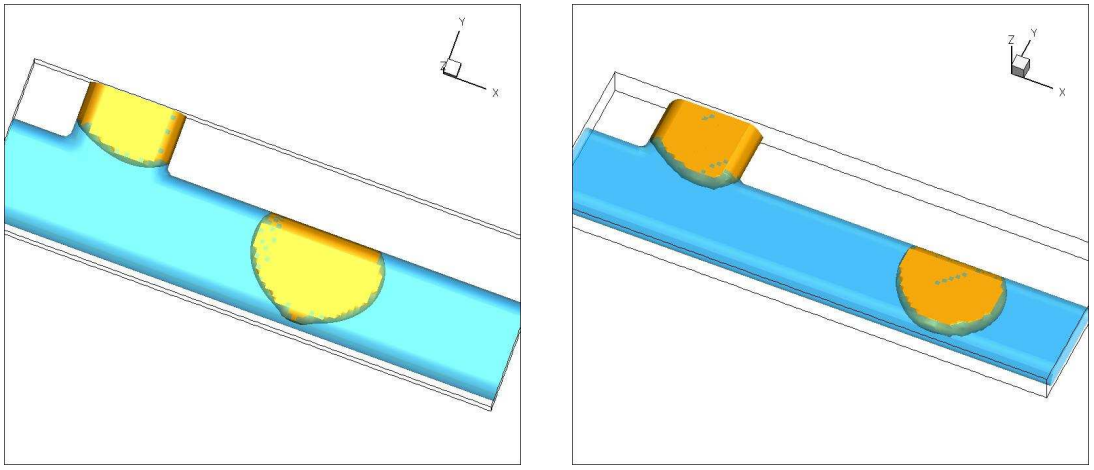

Figure 4.2: Numerical simulations of two fluids in a microfluidic T-junction using the CLSVOF code and experimental setups from Roper's Lab.

CHAPTER 5

CONCLUSION

In this dissertation we have provided valuable knowledge to the fields of PDE constrained optimization, multigrid methods, incompressible multiphase flow simulations on adaptive meshes and GPU computing methods for CFD codes. Furthermore, we have provided a framework with which to move forward towards the the development of a sophisticated design code for microfluidic structures. To this end we have provided the following specific advances to applied and computational mathematics:

- We have demonstrated that the gradient free multidirectional search method (MDS) is a viable alternative to the discrete adjoint method for problems in PDE constrained optimization when one only requires a local method, and when gradient computation is not feasible. We estimate that in the case that there are as many processor cores available for computation as the number of design variables plus one (each processor core maps in parallel to a PDE solve) that the MDS method will perform on par with the discrete adjoint method run on a single processor core. In addition, we have found that the MDS method is not as sensitive to non-linear model features as is the discrete adjoint method for the 1D data assimilation problems tested.

- We have shown that multigrid schemes can significantly accelerate the convergence rate of the multidirectional search method when employed in the solution of PDE constrained optimization methods. Using an FMG cycle, we have demonstrated a 5.5X speedup on a linear 1D data assimilation problem. We have also shown for this problem that the use of weighted averaging of the fine grid solution through the multigrid cycles can produce a more accurate final solution, but at the cost of faster solution times.

- We have described the GPU acceleration of the smoothing steps required by a pre-conditioned conjugate gradient solver used in the solution of the pressure projection step of a coupled level set and volume of fluid code (CLSVOF). This GPU accelerated solver utilizes PGI Fortran compiler with accelerator directives. We have shown that for a large aspect ratio 2D problem, the new accelerated solver is up to 2.4X faster than the CPU version. We have concluded that a new solver should be developed in the compute unified device architecture (CUDA) language which allows for more direct user control over data allocation, and which will allow us to permanently store

data on the GPU. This should substantially reduce communication overhead between the GPU and CPU and lead to a much faster code.

- We have described an improved pressure projection solver for incombressible multi-phase flow simulations on adaptive grids which requires substantially fewer grid point calculations than current methods. The new MGPCG AMR method is up to 4X faster than the previous MG AMR algorithm for the problems tested.

- We have ported the `point_solve_5` subroutine of the unstructured NASA FUN3D code for computation on GPU based clusters, achieving a speedup of up to 5.5X for the subroutine and up to 2X for the overall code. The new subroutine utilizes a novel GPU sharing model that allows multiple processor cores to run on a single GPU simultaneously, and provides for increased scalability. We have shown that this model scales in parallel up to 8 cores - 2 GPUs, and that a 40% performance increase for the accelerated subroutine has been achieved with 4 cores sharing a single GPU.

APPENDIX A

FUN3D TIMING DATA

This appendix contains timing data from the FUN3D acceleration described in chapter 3. Three machines were used for this study. The first is a Dell workstation with dual socket Intel Xeon dual core 5080 processors and a single Nvidia GTX 480 GPU, we refer to this machine as GTX 480 WS. The second machine is a personal Beowulf cluster with two compute nodes each running a single AMD Athlon II quad core processor and a single Nvidia GTX 470 GPU which we will label GTX 470 BC. Finally, we have been aloted 8 nodes on a research cluster at Nvidia corporation, each with dual socket Xeon 6 core processors and 2 Nvidia Tesla M2050 GPUs. We refer to this machine as TESLA RC. The details of these machines are provided below, along with details of the test problems used.

GTX 480 WS: Intel Xeon 5080 Workstation, NVIDIA GTX 480

- CPU: 2 x 2 core @ 3.73 GHz, 2 MB L2
- GPU: 1 x 480 CUDA core @ 1.4 GHz, 1.6 GB Global Memory

GTX 470 BC: AMD Athlon II Beowulf Cluster, NVIDIA GTX 470

- CPU: 2 x 4 core @ 2.6 GHz, 2 MB L2
- GPU: 2 x 448 CUDA core @ 1.25 GHz, 1.26 GB Global Memory

TESLA RC: Nvidia Tesla M2050 Research Cluster

- CPU: 16 x 6 core @ 2.93 GHz, 12 MB L2
- GPU: 16 x 448 Cuda cores @ 1.15 GHz, 3 GB Global Memory

Table A.1: Test problems used in this study. * approximate.

Problem	Configuration	Type	Cells	Nodes
Test Problem 1	Simple Wing/Body	Inviscid	339,206	1,995,247
Test Problem 2	DLR-F6 Aircraft	Turbulent	650,000*	3,900,000*
Test Problem 3	Wing Leading-Edge	Turbulent	900,000*	5,500,000*
Test Problem 4	DLR-F6 Aircraft -	Turbulent	1,121,301	6,558,758

Table A.2: Timing results for routine gs_gpu on test problem 1 using the TESLA RC.

TESLA RC - `gs_gpu`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.17497 s	0.096 s	1.822X

Table A.3: Timing results for routine gs_mpi0 on test problem 1 using the GTX 480 WS.

GTX 480 WS - `gs_mpi0`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1732.8 s	1026.5 s	1.688X
PS5 Subroutine	1	0.455 s	0.170 s	2.676X
FUN3D Code	2	850.6 s	744.5 s	1.143X
PS5 Subroutine	2	0.216 s	0.164 s	1.317X
FUN3D Code	4	496.0 s	648.1 s	0.758X
PS5 Subroutine	4	0.125 s	0.167 s	0.727X

Table A.4: Timing results for routine gs_mpi0 on test problem 1 using 1 node of the GTX 470 BC.

GTX 470 BC - `gs_mpi0`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1728.9 s	1137.1 s	1.568X
PS5 Subroutine	1	0.437 s	0.197 s	2.218X
FUN3D Code	2	963.9 s	882.15 s	1.093X
PS5 Subroutine	2	0.240 s	0.193 s	1.244X
FUN3D Code	4	674.3 s	786.5 s	0.857X
PS5 Subroutine	4	0.172 s	0.197 s	0.873X

Table A.5: Timing results for routine gs_mpi0 on test problem 1 using 2 nodes of the GTX 470 BC.

GTX 470 BC - `gs_mpi0`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	2	888.1 s	596.2 s	1.490X
PS5 Subroutine	2	0.221 s	0.104 s	2.125X
FUN3D Code	4	498.8 s	482.9 s	1.033X
PS5 Subroutine	4	0.121 s	0.104 s	1.163X
FUN3D Code	8	366.0 s	441.6 s	0.829X
PS5 Subroutine	8	0.088 s	0.109 s	0.807X

Table A.6: Timing results for routine gs_mpi1 on test problem 1 using the GTX 480 WS.

GTX 480 WS - `gs_mpi1`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1732.8 s	912.0 s	1.900X
PS5 Subroutine	1	0.455 s	0.128 s	3.555X
FUN3D Code	2	850.6 s	626.4 s	1.358X
PS5 Subroutine	2	0.216 s	0.118 s	1.831X
FUN3D Code	4	496.0 s	528.1 s	0.939X
PS5 Subroutine	4	0.125 s	0.123 s	1.016X

Table A.7: Timing results for routine gs_mpi1 on test problem 1 using 1 node of the GTX 470 BC.

GTX 470 BC - `gs_mpi1`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	1	1728.9 s	1011.7 s	1.709X
PS5 Subroutine	1	0.437 s	0.149 s	2.933X
FUN3D Code	2	963.9 s	754.7 s	1.277X
PS5 Subroutine	2	0.240 s	0.142 s	1.690X
FUN3D Code	4	674.3 s	669.0 s	1.008X
PS5 Subroutine	4	0.172 s	0.148 s	1.162X

Table A.8: Timing results for routine gs_mpi1 on test problem 1 using 2 nodes of the GTX 470 BC.

GTX 470 BC - `gs_mpi1`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D Code	2	881.1 s	522.2 s	1.687X
PS5 Subroutine	2	0.221 s	0.080 s	2.763X
FUN3D Code	4	498.8 s	405.2 s	1.231X
PS5 Subroutine	4	0.121 s	0.079 s	1.532X
FUN3D Code	8	366.0 s	369.3 s	0.991X
PS5 Subroutine	8	0.088 s	0.082 s	1.073X

Table A.9: Timing results for routine gs_mpi2 on test problem 1 using 1 node of the TESLA RC.

GTX 470 BC - `gs_mpi1`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.175 s	0.072 s	2.431X
PS5 Subroutine	2	0.091 s	0.055 s	1.655X

Table A.10: Timing results for routine gs_gpu on test problem 2 using the GTX 480 WS.

GTX 480 WS `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.851 s	0.155 s	5.490X

Table A.11: Timing results for routine gs_gpu on test problem 2 using the GTX 470 BC.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.822 s	0.174 s	4.724X

Table A.12: Timing results for routine gs_mpi2 on test problem 2 using the GTX 480 WS.

GTX 480 WS - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.851 s	0.189 s	4.503X
PS5 Subroutine	2	0.402 s	0.152 s	2.645X
PS5 Subroutine	4	0.237 s	0.155 s	1.529X

Table A.13: Timing results for routine gs_mpi2 on test problem 2 using two nodes of the GTX 470 BC machine.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	0.822 s	0.215 s	3.823X
PS5 Subroutine	2	0.449 s	0.198 s	2.268X
PS5 Subroutine	4	0.332 s	0.197 s	1.685X

Table A.14: Timing results for routine gs_mpi2 on test problem 2 using two nodes of the GTX 470 BC machine.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	2	0.407 s	0.120 s	3.392X
PS5 Subroutine	4	0.223 s	0.112 s	1.991X
PS5 Subroutine	8	0.165 s	0.115 s	1.435X

Table A.15: Timing results for routine gs_mpi2 on test problem 3 using the GTX 480 WS machine.

GTX 480 WS - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	1	1.140 s	0.257 s	4.436X
PS5 Subroutine	2	0.558 s	0.213 s	2.620X

Table A.16: Timing results for routine gs_mpi2 on test problem 3 using two nodes of the GTX 470 BC machine. GPU Memory is insufficient for sharing among 4 threads, the problem must be split across two GPUs.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	2	0.564 s	0.213 s	2.648X
PS5 Subroutine	4	0.316 s	0.179 s	1.765X
PS5 Subroutine	8	0.230 s	0.168 s	1.369X

Table A.17: Timing results for routine gs_mpi2 on test problem 4 using two nodes of the GTX 470 BC machine.

GTX 470 BC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	4	0.411 s	0.195 s	2.108X
PS5 Subroutine	8	0.299 s	0.189 s	1.582X

Table A.18: Timing results for routine gs_mpi2 on test problem 4 using eight nodes of the TESLA RC machine.

TESLA RC - `gs_mpi2`	CPU Cores	CPU Time	CPU/GPU Time	Speedup
PS5 Subroutine	8	0.080 s	0.035 s	2.286X

BIBLIOGRAPHY

[1] www.top500.org, last accessed Aug. 5, 2010.

[2] http://www.green500.org, last accessed Aug. 16th, 2010.

[3] http://tetruss.larc.nasa.gov, last accessed Aug. 13, 2010.

[4] http://fun3d.larc.nasa.gov, last accessed Jan. 2, 2011.

[5] W.K. Anderson and D.L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Comput. Fluids*, 23(1):1–21, 1994.

[6] D. Aurox and J. Blum. Data assimilation methods for an oceanographic problem. volume Lecture Notes XVI of *Mathematics in Industry*. Springer-Verlag, 2004.

[7] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.

[8] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.

[9] J. Brezillon and N.R. Gauger. 2D and 3D aerodynamic shape optimisation using the adjoint approach. *Aerospace Science and Technology*, 8:715–727, 2004.

[10] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, Philadelphia, PA, USA, second edition, 2000.

[11] E. F. Campana, D. Peria, Y. Tahara, and F. Stern. Shape optimization in ship hydrodynamics using computational fluid dynamics. *Comput. Meth. Appl. Mech. Eng.*, 196(3):634–651, 2006.

[12] G. Carpenteiri, B. Koren, and M.J.L. van Tooren. Adjoint-based shape optimization on unstructured meshes. *Journal of Computational Physics*, 224:267–287, 2007.

[13] J. M. Cohen and M. J. Molemaker. A fast double precision CFD code using cuda. In *Proceedings of Parallel CFD 2009*, 2009.

[14] NVIDIA Corporation. *NVIDIA CUDA Programming Guide, version 2.2*, April 2009.

[15] A. Corrigan, F. Camelli, R. Lohner, and J. Wallin. Running unstructured grid based CFD solvers on modern graphics hardware. In *19th AIAA CFD Conference*, San Antonio, Texas, USA, June 2009.

[16] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th Nat. Conf. ACM*, pages 157–172, 1969.

[17] J.E. Dennis and V. Torczon. Direct search methods on parallel machines. *SIAM Journal of Optimization*, 1(4):448–474, 1991.

[18] D. Dommermuth, M. Sussman, R. Beck, T. O'Shea, and D. Wyatt. The numerical simulation of ship waves using cartesian grid methods with adaptive mesh refinement. In *Proccedings of the Twenty Fifth Symposium on Naval Hydr.*, St. Johns, New Foundland and Labrador, Canada, 2004.

[19] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *The Sourcebook of Parallel Computing*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier, November 2002.

[20] D. Drikakis, O. P. Iliev, and D. P. Vassileva. A nonlinear multigrid method for the three dimensional incompressible Navier-Stokes equations. *J. Comput. Phys.*, 146(1):301–321, 1998.

[21] A. C. Duffy, D. P. Hammond, and E. J. Nielsen. Production level CFD code acceleration for hybrid many-core architechtures. *Parallel Comput.*, 2011. Accepted, pending revisions.

[22] E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *J. Comput. Phys.*, 227:10148–10161, 2008.

[23] J. Dongarra et. al. Exascale roadmap 1.0, 2010.

[24] F. Fang, C.C. Pain, M.D. Piggott, G.J. Gorman, and A.J.H Goddard. An adaptive mesh adjoint data assimilation method applied to free surface flows. *Int. J. Numer. Methods Fluids*, 47:995–1001, 2005.

[25] F. Fang, M.D. Piggott, C.C. Pain, G.J. Gorman, and A.J.H Goddard. An adaptive mesh adjoint data assimilation method. *Ocean Model.*, 15:39–55, 2006.

[26] D. Furbish, M.Y. Hussaini, F.-X. Le Dimet, and P. Ngnepieba. On discretization error and its control in varational data assimilation. *Tellus A*, 60:979–991, 2008.

[27] P. Garstecki, M. Fuerstman, H. Stone, and G. Whitesides. Formation of droplets and bubbles in a microfluidic t-junction–scaling and mechanism of break-up. *Lab Chip*, 6:437–446.

[28] D. Göddeke, S. H. Buijssen, H. Wobker, and S. Turek. GPU acceleration of an unmodified parallel finite element navier-stokes solver. In *High Performance Computing and Simulation 2009*, Leipzig, Germany, June 2009.

[29] D. Göddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE T. Parall. Distr.*, March 2010. Special Issue: High Performance Computing with Accelerators.

[30] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU enhanced cluster. *Parallel Comput.*, 33(10-11):685–699, 2007.

[31] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *Int. J. Comput. Sci. Eng.*, 4(1):36–55, 2008.

[32] M. Griebel and P. Zaspel. A multi-gpu accelerated solver for the three-dimensional two-phase incompressible navier-stokes equations. *Comput Sci Res Dev*, 25:65–73, April 2010.

[33] A. K. Griffith and N.K. Nichols. Data assimilation using optimal control theory. Numerical analysis report, The University of Reading, October 1994.

[34] B. E. Griffith, R. D. Hornung, D. M. McQueen, and C. S. Peskin. An adaptive, formally second order accurate version of the immersed boundary method. *J. Comput. Phys.*, 223:10–49, 2007.

[35] Y. Ha and S. Cho. Shape sensitivity analysis for incompressible fluid using SPH projection method. In *8th. World Congress on Computational Mechanics, 5th. European Congress on Computational Methods in Applied Sciences and Engineering*, Venice, Italy, June 2008.

[36] J. P. Hämäläinen, R. E. Mäkinen, and P. Tarvainen. Evolutionary shape optimization in CFD with industrial applications. In *European Congress on Computational Methods in Science and Engineering*, 2000.

[37] J. Haslinger and R.A.E. Mäkinen. *Introduction to Shape Optimization: Theory, Aprximation and Computation.* Advances in Design and Control. SIAM, Philadelphia, PA, USA, 2003.

[38] P. W. Hemker, B. Koren, and S. P. Spekreijse. A nonlinear multigrid method for the efficient solution of the steady Euler equations. *Lect. Notes Phys.*, 264:308–313, 1986.

[39] R. M. Hicks, E. M. Murman, and G. N. Vanderplaats. An assessment of airfoil design by numerical optimization. Technical Report TM X-3092, NASA, Ames Research Center, Moffett Field, CA, USA, July 1974.

[40] D. Isebe, P. Azerad, F. Bouchette, B. Ivorra, and B. Mohammadi. Shape optimization of geotextile tubes for sandy beach protection. *Int. J. Numer. Methods Eng.*, 2007.

[41] D. Isebe, P. Azerad, B. Mohammadi, and F. Bouchette. Optimal shape design of defense structures for minimizing short wave impact. *Coastal Eng.*, 55:35–46, 2008.

[42] D.A. Jacobsen, J. Thibault, and I. Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2010.

[43] D.A. Jacobsen, J.C. Thibault, and I. Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on Multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting*, Orlando, FL U.S.A., January 2010.

[44] A. Jameson and L. Martinelli. A continuous adjoint method for unstructured grids. In *AIAA Paper 03-3955, 16th AIAA CFD Conference*, pages 23–26, 2003.

[45] A. Jameson, S. Shakaran, and L. Martinelli. Continuous adjoint method for unstructured grids. *AIAA J.*, 46(5):1226–1239, 2008.

[46] D. C. Jesperson. Acceleration of a cfd code with a gpu. NAS Technical Report NAS-09-003, NASA Ames Research Center, November 2009.

[47] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE SC98 Conference*, 1998.

[48] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[49] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. Hwu. GPU clusters for high-performance computing. In *IEEE Cluster 2009*, New Orleans, Louisiana, USA, Aug. - Sept. 2009.

[50] E. Laporte and P. Le Tallec. *Numerical Methods in Sensitivity Analysis and Shape Optimization*. Modeling and Simulation in Science, Engineering and Technology. Birkhäuser, 2003.

[51] E.M. Lee-Rausch, N.T. Frink, D.J. Mavriplis, R.D. Rausch, and W.E. Milholen. Transonic drag prediction on a DLR-F6 transport configuration using unstructured grid solvers. *Comput. Fluids*, 38:511–532, 2009.

[52] R. M. Lewis and S. G. Nash. Model problems for the multigrid optimization of systems governed by differential equations. *SIAM J. Sci. Comp.*, 26(6):1811–1837, 2005.

[53] F. Losasso, R. Fedkiw, and S. Osher. Spatially adaptive techniques for level set methods and incompressible flow. *Comput. Fluids*, 35(10):995–1010, 2006.

[54] F. Lossaso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.*, 23:457–462, 2004.

[55] C. A. Mader, J. R. R. A. Martins, J. J. Alonso, and E. Van Der Weide. ADjoint: An approach for the rapid developement of discrete adjoint solvers. *AIAA J.*, 46(4):863–873, 2008.

[56] A. McNamara, A. Treuille, Z. Popović, and J. Stam. Fluid control using the adjoint method. *ACM Trans. Graphics*, 23:449–446, 2004.

[57] S. K. Nadarajah and A. Jameson. Optimum shape design for unsteady three-dimensional viscous flows using a non-linear frequency domain method. In *AIAA paper 2006-3455, 24th Applied Aerodynamics Conference*, San Francisco, CA, USA, 2006.

[58] S. G. Nash. A multigrid approach to discretized optimization problems. *Optim. Meth. Softw.*, 14:99–116, 2000.

[59] M. Nemec and M. J. Aftsomis. Adjoint formulation for an embedded-boundary cartesian method. In *AIAA paper 2005-4987, 17th Computational Fluid Dynamics Conference*, Toronto, ON, CAN, 2005.

[60] M. Nemec, M. J. Aftsomis, S. M. Murman, and T. H. Pulliam. Adjoint formulation for an embedded-boundary cartesian method. In *43rd AIAA Aerospace Sciences Meeting*, Reno, NV, USA, 2005.

[61] R. Nourgaliev, S. Kadioglu, and V. Mousseau. Marker redistancing/level set method for high-fidelity implicit interface tracking. *SIAM J. Sci. Comput.*, 32(1):320–348, 2010.

[62] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, New York, NY, 1988.

[63] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2009.

[64] S. A. Ragab. Shape optimization in free surface potential flow using an adjoint formulation. Hydrodynamics Directorate Technical Report NSWCCD-50-TR-2003/33, Naval Surface Warfare Center, Carderock Division, West Bethesda, MD, USA, August 2003.

[65] S. A. Ragab. Shape optimization of surface ships in potential flow using an adjoint formulation. *AIAA J.*, 42(2):296–304, 2004.

[66] A. R. Robinson and P. F.J. Lermusiaux. Overview of data assimilation. Harvard Reports in Physical/Interdisciplinary Ocean Science 62, Harvard University, 2000.

[67] P.L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *J. Comp. Phys.*, 43(2):357–372, 1981.

[68] W. Shi, J. Qin, N. Ye, and B. Lin. Droplet-based microfluidic system for individual *Caenorhabditis elegans* assay. *Lab Chip*, 8:1432–1435, 2008.

[69] P.R. Spalart and S.R. Allmaras. A one-equation turbulence model for aerodynamic flows. *Rech. Aerospatiale*, 1:5–21, 1994.

[70] P. Stewart, N. Lay, M. Sussman, and M. Ohta. An improved sharp interface method for viscoelastic and viscous two-phase flows. *J. Sci. Comput.*, 35(1):43–61, 2008.

[71] M. Sussman. A parallelized, adaptive algorithm for multiphase flow in general geometries. *Comput. Struct.*, 83:435–444, 2005.

[72] M. Sussman, A. Almgren, J. Bell, P. Colella, L. Howell, and M. Welcome. An adaptive level set approach for incompressible two-phase flows. *J. Comput. Phys.*, 148:81–124, 1999.

[73] M. Sussman, K. Smith, M. Hussaini, M. Ohta, and R. Zhi-Wei. A sharp interface method for incompressible two-phase flows. *J. Comput. Phys.*, 221(2):469–505, 2007.

[74] Y. Tahara, F. Stern, and Y. Himeno. Computational fluid dynamics-based optimization of a surface combatant. *J. Ship Res.*, 48(4):273–287, 2004.

[75] W.-H. Tan and S. Takeuchi. A trap-and-release integrated microfluidic system for dynamic microarray applications. *PNAS*, 104(4):1146–1151, 2007.

[76] O. Tatebe. The multigrid preconditioned conjugate gradient method. In *6th Copper Mountain Conference on Multigrid Methods*, Copper Mountain, CO, USA, April 1993.

[77] J. C. Thibault and I. Senocak. Cuda implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *47th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2009.

[78] V. Torczon. *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*. PhD thesis, Rice University, May 1989.

[79] V. Torczon. On the convergence of the multidirectional search algorithm. *SIAM J. Optim.*, 1(1):123–145, 1991.

[80] B. G. van Bloemen Waanders, R. A. Bartlett, S. S. Collis, E. R. Keiter, C. C. Ober, T. M. Smith, V. Akcelik, O. Ghattas, J. C. Hill, M. Berggren, M. Heinkenschloss, and L. C. Wilcox. Sensitivity technologies for large scale simulation. Technical Report SAND2004-6574, Sandia National Laboratories, January 2005.

[81] J. Waltz and R. Löhner. A grid coarsening algorithm for unstructured multigrid applications. In *38th AIAA Aerospace Sciences Meeting and Exhibit*, AIAA-00-0925, Reno, NV, USA, 2000.

[82] Q. Wang, D. Gleich, A. Saberi, N. Etemadi, and P. Moin. A Monte Carlo method for solving unsteady adjoint equations. *J. Comput. Phys.*, 227:6184–6205, 2008.

BIOGRAPHICAL SKETCH

Austen Duffy was born in East Stroudsburg, Pennsylvania on August 16, 1980 to parents Francis J. and Bonnie L. Duffy. He grew up in the Pocono Mountains region of northeastern Pennsylvania, graduating from Pocono Mountain Senior High School in 1999. In 2003, Austen recieved his B.S. in Mathematics from York College of Pennsylvania, a small private liberal arts school in southcentral Pennsylvania. Upon recieving his bachelors degree, Austen began his graduate education at Indiana University of Pennsylvania, and recieved a terminal M.S. in Applied Mathematics in 2005 before entering the Ph.D. program in Applied and Computational Mathematics at Florida State University. Austen's research interests are primarily computational in nature, with an emphasis towards real world applications in science and engineering. In addition to his time as a teaching and research assistant, Austen has worked as a visiting researcher at the National Institute of Aerospace and the NASA Langley Research Center.